The Dissertation Committee for Pengyu Nie
certifies that this is the approved version of the following dissertation:

Machine Learning for Executable Code in Software Testing and Verification

Committee:

Milos Gligoric, Supervisor

Junyi Jessy Li

Raymond J. Mooney

Alexandros G. Dimakis

August Shi

Machine Learning for Executable Code in Software Testing and Verification

by

Pengyu Nie

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2023

Acknowledgments

First of all, I would like to sincerely thank my advisor Milos Gligoric for his innumerous supports in the past six years. I started my PhD journey almost from scratch, clueless of how the academic world works, and it was Milos who patiently taught me everything—from coding in Bash (after which I learned Python myself :)), managing research projects, to applying for academic jobs—and brought me to where I am now. We had so much fun hacking and catching up the deadlines together. I hope to be a good advisor as he is in return for his help, although I could never do enough to repay.

I would like to thank Junyi Jessy Li, Raymond J. Mooney, Alexandros G. Dimakis, and August Shi for serving on my dissertation committee and providing constructive feedback on my research. Jessy and Ray have been supervising my interdisciplinary research from the natural language processing side, and our (long-time) collaborations have always been fruitful.

I also want to thank all my collaborators: Rahul Banerjee, Jonathan Bell, Ahmet Celik, Matthew Coley, Alan Han, Sarfraz Khurshid, Jaeseong Lee, Owolabi Legunsen, Yu Liu, Aleksandar Milicevic, Karl Palmskog, Sheena Panthaplackel, Marinela Parovic, Rishabh Rai, Christopher J. Rossbach, Zachary Thurston, Zhiqiang Zang, and Jiyang Zhang. It was fun to work with them and I look forward to future collaboration opportunities.

I could not survive the long PhD journey without hanging out with my lab mates: Nader Al Awar, Ahmet Celik, Yu Liu, Marinela Parovic, Aditya Thimmaiah, Kaiyuan Wang, Wenxi Wang, Zhiqiang Zang, Jiyang Zhang, and Chenguang Zhu. I was honored to work with several exceptional undergraduate students: Rahul Banerjee, Kush Jain, Rishabh Rai from UT, and Alan Han and Zachary Thurston from Cornell. I wish them success in their careers.

My two internships at Meta (called Facebook at that time) were eye-opening, and I enjoyed working with my mentors Georgios Gousios and Vijayaraghavan Murali, as well as my colleagues Lee Gross, Vivek Nair, and Edward Yao.

Throughout my PhD and especially during my academic job search, I received lots of advice and feedback from many professors, including Jonathan Aldrich, Greg Durrett, Mattan Erez, Wing Lam, Owolabi Legunsen, Darko Marinov, Sasa Misailovic, August Shi, Weihang Wang, Xiaoyin Wang, Wei Yang, and Lingming Zhang. I really appreciate their advice, and I will carry on the mentorship to the next generations in return.

I was also well supported by the excellent staff at UT. I would like to thank Cayetana Garcia, Melanie Gulick, Barry Levitch, and Melody Singleton, who helped me with paperwork so that I can focus on research.

Most of my research projects were inspired by and built for the open-source projects. I appreciate the help from the maintainers of these projects, in particular, Anton Trunov (the maintainer of the PCM library) and Emilio Jesús Gallego Arias (the maintainer of SerAPI) who contributed to parts of Roosterize.

Parts of this dissertation were published at ICSE 2023 [146] (Chapter 2); IJCAR 2020 [143], and ICSE Demo 2020 [144] (Chapter 3). I would like to thank the anonymous reviewers and audience at the conferences of all my papers for their comments. I am honored that I won an ACM SIGSOFT Distinguished Paper Award for my FSE 2019 paper on TrigIt [141], which was the first full paper I published.

My research was funded by the University of Texas at Austin Graduate School Continuing Fellowship, Runtime Verification, Google Faculty Research Award, and the US National Science Foundation under Grant Nos. CCF-1566363, CCF-1652517, CCF-1704790, and CCF-2107291.

Last but not least, I would like to thank my mother Yingying Peng, my father Lei Nie, and my significant other Yu Liu for their unconditional love and support.

Abstract

Machine Learning for Executable Code in Software Testing and Verification

Pengyu Nie, PhD
The University of Texas at Austin, 2023

SUPERVISOR: Milos Gligoric

Software testing and verification are essential for keeping software systems reliable and safe to use. However, it requires significant manual effort to write and maintain code artifacts needed for testing and verification, i.e., tests and proofs. With the pressure for developing software in limited time, developers usually write tests and proofs much later than the code under test/verification, which leaves room for software bugs in unchecked code.

Recent advances in machine learning (ML) models, especially large language models (LLMs), can help reduce manual effort for testing and verification. Namely, developers can benefit from ML models' predictions to write tests and proofs faster. However, existing models understand and generate software code as natural language text, ignoring the unique property of software being *executable*. Software *execution* is the process of a computer reading and acting on software code. Our insight is that ML models can greatly benefit from software execution, e.g., by inspecting and simulating the execution process, to generate more accurate predictions. Integrating execution with ML models is important for generating tests and proofs because ML models using only syntax-level information do not perform well on these tasks.

This dissertation presents the design and implementation of two execution-guided ML models to improve developers' productivity in writing testing and verification code: TeCo for test completion and Roosterize for lemma naming.

First, this dissertation introduces TeCo to aid developers in completing next statements when writing tests, a task we formalized as *test completion*. TeCo exploits code semantics extracted from test execution results (e.g., local variable types) and execution context (e.g., last called method). TeCo also reranks ML model's predictions by executing the predicted statements, to prioritize functionally correct predictions. Compared to existing code completion models that use only syntax-level information (including LLMs trained on massive code dataset), TeCo improves the accuracy of test completion by 29%.

Second, this dissertation introduces Roosterize to *suggest lemma names* when developers write proofs using proof assistants, such as Coq. Consistent coding conventions are important as verification projects based on proof assistants become larger, but manually enforcing the conventions can be costly. Existing ML models for method naming, a similar task in other programming languages, extract and summarize information extracted from code tokens, which is not suitable for Coq where the lemma names should exhibit semantic meanings that are not explicit in code tokens. Roosterize leverages the execution representations of the lemma from various phases of the proof assistant's execution, including syntax trees from the parser and elaborated terms from the kernel. Roosterize improves the accuracy of lemma naming by 39% compared to baselines.

Our findings in this dissertation support that the integration with execution can effectively improve the accuracy of ML models for testing and verification, which enables developing trustworthy software with high-quality tests and proofs with less manual effort.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Software plays an important role in our everyday lives, and it is the foundation of many important technologies nowadays, including artificial intelligence, autonomous driving, cryptocurrency, etc. Bugs in software frequently happen and can be harmful, causing anything from bad user experiences to loss of lives [32, 33]. To prevent software bugs and ensure software correctness, developers practice software *testing* and *verification* to check if the software matches its *specifications*, i.e., the expected behaviors of the software.

Software testing, which is the most commonly-used technique in industry for checking software correctness, exercises the software on a set of given inputs and compares the results against expected outputs. Each exercise is called a *test case*. In popular object-oriented languages like Java, a test case is usually written as a method with several statements for preparing inputs, invoking the method under test, and checking the outputs; thus, it is also referred to as a *test method*. For example, Figure 1.1 shows a test method `testToNextMarkerPartialVar1` written in Java using the JUnit [97, 98] testing framework, together with its method under test `readToNextMarkerNewBuffer`. Once *executed* with the JUnit testing framework, there are two possible outcomes for each test method: the test method passes, which means the actual execution outputs match the expectations specified in this test method; or the test method fails, due to the actual execution outputs not matching expected outputs, which means the software might contain a bug.

Software verification, on the other hand, tries to prove that the software matches its specifications for *all* cases instead of hand-picked input-output pairs. We focus on software verification using proof assistants like Coq [25], which has been adopted in various domains such as compilers [44]. The specifications are written as *lemmas*, which describe the invariants that are expected to hold at all times. Each lemma is followed by a *proof script*, which describes the steps to prove the lemma.

```
class SegmentReader {
    public ByteBuffer readToNextMarkerNewBuffer() throws IOException {
        if (done)
            return null;
        List<ByteBuffer> buffers = new ArrayList<ByteBuffer>();
        readToNextMarkerBuffers(buffers);
        return NIOUtils.combineBuffers(buffers);
    }
}

class SegmentReaderTest {
    @Test
    public void testToNextMarkerPartialVar1() throws IOException {
        byte[] bytes = new byte[] { 0, 0, 1, 42, 43, 44, 45, 46, 0, 0, 1, 43 };
        ReadableByteChannel ch = Channels.newChannel(new ByteArrayInputStream(bytes));
        SegmentReader reader = new SegmentReader(ch, 1);
        reader.setBufferIncrement(1);
        ByteBuffer buf1 = reader.readToNextMarkerNewBuffer();
        ByteBuffer buf2 = reader.readToNextMarkerNewBuffer();
        ByteBuffer buf3 = reader.readToNextMarkerNewBuffer();
        Assert.assertEquals(ByteBuffer.wrap(bytes, 0, 8), buf1);
        Assert.assertEquals(ByteBuffer.wrap(bytes, 8, 4), buf2);
        Assert.assertNull(buf3);
    }
}
```

Figure 1.1: Example test method written in Java using the JUnit testing framework. Code from `jcodec/jcodec` in files `SegmentReader.java` and `SegmentReaderTest.java`. The test method `testToNextMarkerPartialVar1` tests the method under test `readToNextMarkerNewBuffer` by invoking it on a byte array.

```
Variables A B : Type.
Hypothesis A_eq_dec : forall x y : A, {x = y} + {x <> y}.

Definition update st h (v : B) :=
  fun nm => if A_eq_dec nm h then v else st nm.

Lemma update_nop :
  forall (sigma : A -> B) x y,
    update A_eq_dec sigma x (sigma x) y = sigma y.
Proof using.
  unfold update.
  intros. break_if; congruence.
Qed.

Lemma update_diff :
  forall (sigma : A -> B) x v y,
    x <> y ->
    update A_eq_dec sigma x v y = sigma y.
Proof using.
  unfold update.
  intros.
  break_if; congruence.
Qed.
```

Figure 1.2: Example lemma and proof script written in Coq. Code from `uwplse/StructTact` in file `Update.v`. The function `update` returns a new version of a given function `st`, and this returned function maps `h` to `v` (specified by the lemma `update_diff`) but otherwise behaves as `st` (specified by the lemma `update_nop`).

For example, Figure 1.2 shows two lemmas `update_diff` and `update_nop` and their proof scripts written in Coq, together with the function to be verified, `update`. Once *executed* with the Coq proof assistant (i.e., proof checked), there are two possible outcomes for each lemma: the proof succeeds, which means the software matches its expected behaviors specified in the lemmas; or the proof fails, which may indicate an error during verification (e.g., incorrect proof scripts) or a bug in the software.

Software testing and verification are two complementary techniques to ensure software correctness. Testing detects the *presence* of bugs [37] (within the test inputs written by developers), and verification checks the *absence* of bugs (with regards to the lemmas written by developers). However, writing high-quality tests and proofs[1] remains largely a manual and tedious process. A prior study found that close to half of developers' time is spent on testing [49]. As software development time budget is limited, developers frequently prioritize writing code under test and defer writing tests [216]. Verification projects using proof assistants are growing in scale and usually incur more cost than testing [177]; for example, CompCert [44, 112], the verified C compiler written in Coq, has more than 120K lines of code and took many person-years to develop. The manual effort of writing tests and proofs can be a major burden on checking the correctness of changed or added code in a timely manner, leaving room for hidden software bugs.

One approach to reduce the manual effort is to use machine learning (ML) models to generate (some parts of) tests and proofs, which is made possible by recently proposed deep neural networks (such as recurrent neural networks [85, 212] and transformers [200]) that can effectively learn from the existing tests and proofs in software corpora. Namely, given the input of a partially written test or proof (e.g., the first few statements in a test), an ML model can make predictions on the missing part (e.g., the next statement in the test). Developers can then accept/revise the

[1]We use proofs to refer to all code artifacts that need to be written during verification, including lemmas and proof scripts.

predictions, which is easier than manually writing tests and proofs without the help from ML models.

ML models have been applied to many tasks in the generation and summarization of (general-purpose) code [11, 125, 204, 208]. However, only few researchers have studied applying ML models on generating tests and proofs, and suggested that performance of ML models on testing- and verification-related tasks is low [21, 55, 90, 198, 207, 215]. To accomplish these tasks, ML models need to reason about execution, e.g., to predict the program state after executing the method under test with some inputs. Prior work has shown that ML models make mistakes in reasoning about execution and do not generalize well to complicated code [15, 149].

Tests and proofs are themselves *executable*, which can be integrated with the ML models to enhance the reasoning about execution. We define execution as the process of a computer reading and acting on software code. In the context of testing (using the common xUnit testing frameworks), the execution starts by triggering the testing framework, which then invokes each test method and collects their results. In the context of verification (using proof assistants), the execution starts by invoking the proof assistant, which iteratively evaluates and checks each specification, lemma, and proof script. Execution is a rich source of code semantics because it involves type checking, computing the value of variables, resolving the invocations to external methods/functions, etc.

We believe that integrating execution with ML models is the key to build more performant ML models for software testing and verification. Execution can be integrated in two aspects: extracting code semantics as context and checking the quality of predictions. Code semantics can be extracted by static analysis, e.g., simulating a part of execution (e.g., type checking) on source code or compiled code, and/or dynamic analysis, e.g., alternating the execution to record the type and value of some variables. The extracted code semantics is deterministic and accurate. On the contrary, the code semantics exhibited in ML models (e.g., embeddings and middle layers

of neural networks) is "learned" based on statistical analysis of the inputs and outputs and can be nondeterministic and inaccurate. By providing accurate code semantics from execution as inputs, we can ground ML models' predictions on facts and reduce the errors caused by inaccurate code semantics learned from noisy data (e.g., from code written in an obsolete version of a programming language). Similarly, when generating the prediction as a sequence of tokens, pure ML models do not necessarily generate executable code (i.e., a passing test or proof), or even compilable code. Execution can serve as a checker to determine whether a generated prediction can lead to a passing test or proof, which can be used to improve the quality of predictions, for example, by reranking multiple predictions.

This dissertation presents two main bodies of research on ML models integrated with execution for test completion and lemma naming, respectively, with the goal to improve developers' productivity in testing and verification.

The thesis statement of this dissertation is as follows:

> *Integrating execution with ML models is vital for developing performant ML models to reduce the manual effort required in testing and verification, by making predictions on code completion and coding conventions.*

This dissertation makes the following key contributions:

⋆ We present TeCo for completing next statements when writing tests, a task we formalized as *test completion*. TeCo is the first ML model for test completion that uses execution-guided code semantics as inputs and reranks the predictions by test execution. TeCo exploits code semantics extracted from test execution results (e.g., local variable types) and execution context (e.g., last invoked method). TeCo also reranks the candidate next statements predicted by the ML model by execution to prioritize functionally correct outputs. We implemented TeCo for Java, and evaluated TeCo on a corpus of 130,934 test methods from 1,270 open-source Java projects. Compared to existing code completion models that use only syntax-level

data (including large language models trained on massive code dataset), TeCo improves the accuracy of test completion by 29%.

⋆ We present Roosterize for suggesting lemma names when developers write proofs using the Coq proof assistant. Roosterize is the first ML model for automatically learning and suggesting lemma names, using the runtime representations of the lemmas to generate more accurate predictions. Existing ML models for method naming, a similar task in other programming languages, extract and summarize information extracted from code tokens, which is not suitable for Coq where the lemma names should exhibit semantic meanings that are not explicit in code tokens. Roosterize leverages the runtime representations of the lemma from various phases of executing the proof assistant, including syntax trees from the parser and elaborated terms from the kernel. We evaluated Roosterize on a corpus of 297K lines of Coq code, and found that Roosterize improves the accuracy of lemma naming by 39% compared to the best baseline model.

The code and corpora of both TeCo[2] and Roosterize[3] are open sourced to facilitate future research. We also implemented a Visual Studio Code plugin for Roosterize[4] to make it easier for developers to use our technique.

[2] `https://github.com/EngineeringSoftware/teco`
[3] `https://github.com/EngineeringSoftware/roosterize`
[4] `https://github.com/EngineeringSoftware/roosterize-vscode`

Chapter 2: Learning Deep Semantics for Test Completion

Writing tests is a time-consuming yet essential task during software development. We propose to leverage recent advances in deep learning for text and code generation to assist developers in writing tests. We formalize the novel task of *test completion* to automatically complete the next statement in a test method based on the context of prior statements and the code under test. We develop TeCo—a deep learning model using code semantics for test completion. The key insight underlying TeCo is that predicting the next statement in a test method requires reasoning about code execution, which is hard to do with only syntax-level data that existing code completion models use. TeCo extracts and uses six kinds of code semantics data, including the execution result of prior statements and the execution context of the test method. To provide a testbed for this new task, as well as to evaluate TeCo, we collect a corpus of 130,934 test methods from 1,270 open-source Java projects. Our results show that TeCo achieves an exact-match accuracy of 18%, which is 29% higher than the best baseline using syntax-level data only. When measuring functional correctness of generated next statements, TeCo can generate runnable code in 29% of the cases compared to 19% obtained by the best baseline. Moreover, TeCo is significantly better than prior work on test oracle generation.[1]

2.1 Overview

Software testing is the most common approach in industry to check the correctness of software. However, manually writing tests is tiresome and time-consuming.

One option is to automatically generate tests. Researchers have proposed a

[1]Parts of this chapter are published at ICSE 2023 [146]. Compared to the version published at the conference, this chapter adds Codex as a baseline and a qualitative study of results.

number of techniques in this domain, including fuzz testing [156, 218, 219], property-based testing [6, 31, 38, 41, 68, 86, 108], search-based testing [61, 77], combinatorial testing [43], etc. Despite being effective in detecting software bugs, these techniques generate tests with stylistic issues, as test code generated through these techniques rarely resemble manually-written tests [50, 181, 220] and can be hard to maintain. As a result, these automated techniques end up being used only as supplements to manually-written tests.

Another option is to use machine learning, namely training a model on existing manually-written tests and applying it when writing new tests, which is a plausible methodology supported by the naturalness of software [83, 172]. Advances in deep learning such as recurrent neural networks [85, 182] and large-scale pre-trained transformer models [113, 170, 171, 200] have led to promising new research in a variety of software engineering tasks, such as code completion [40, 114, 168, 173, 195, 196] and code summarization [4, 5, 94, 109, 206]. Code generated with modern models are intelligible to humans, yet we cannot fully rely on them to generate large chunks of meaningful code, or expect them to understand the broader project context.

Our goal is to design machine learning approaches to aid developer productivity when *writing tests*. We present a novel task—*test completion*—to help developers write tests faster. Specifically, once a developer starts writing a test method, the developer can leverage test completion to automatically obtain the next statement in the test code (at any point the developer desires).

Despite being closely related to code completion [114, 168, 173, 195, 196], test completion is distinct in that test code has several unique characteristics. First, the method under test provides extra context that can be leveraged when completing a test method. Second, test code follows a different programming style that focuses on exercising the method under test. Specifically, a test method usually consists of a sequence of statements in the following order: prepare test inputs, execute method under test, and check execution results using assert statements (i.e., test oracles).

We present the first deep learning solution—TeCo—that takes into account these unique characteristics of tests. TeCo *uses code semantics as inputs for novel ML models and performs reranking via test execution.*

Code semantics refers to the information related to test/code execution not available in the syntax-level data (i.e., source code). TeCo extracts code semantics (e.g., types of local variables) using software engineering tools and feeds them directly to the model. Once top-k predictions are produced, TeCo further ensures the output quality by *executing* the generated statements, and prioritize the runnable and compilable statements over the others.

We design the code semantics used by TeCo based on our experience with software analysis in order to best capture the unique characteristics of the test completion task. In total, we consider six different kinds of code semantics that can be grouped to two categories: (1) execution result, including the types of the local variables and whether fields are initialized; (2) execution context, including the setup and teardown methods, the last called method in the test method, and statements in non-test code with similar previous statements.

We implemented TeCo to support test methods written in Java. We evaluate TeCo on a newly collected corpus consisting of 130,934 test methods with 645,633 statements from 1,270 projects. We release this corpus to the community as a testbed for the test completion task.

We performed extensive evaluations of TeCo on this corpus—covering lexical similarity and functional correctness—to show the importance of combining code semantics with deep learning. We report results comparing the generated statements against the gold manually-written statements using a suite of automatic metrics: exact-match accuracy, top-10 accuracy, BLEU [160], CodeBLEU [176], edit similarity [196], and ROUGE [120]. TeCo significantly outperforms baselines that use only syntax-level data on all metrics. We also measure functional correctness by trying to compile and run the generated statements. TeCo can produce a runnable next

statement 29% of the time, while the figure for the best baseline model is only 19%. Moreover, we also evaluated TeCo on the task of test oracle generation [55, 207], which is a sub-task of test completion. TeCo achieves an exact-match accuracy of 16%, which significantly outperforms the prior state-of-the-art's exact-match accuracy of 12%.

The main contributions of this chapter include the following:

- **Task**. We propose a novel task, test completion, with the goal to help developers write test methods faster.
- **Idea**. We propose using code semantics and code execution when designing ML models targeting code-related tasks.
- **Model**. We developed TeCo, the first transformer model trained on large code semantics data for test completion. Furthermore, TeCo performs reranking by execution. The use of code semantics is vital for correctly modeling the execution process in the test methods.
- **Corpus**. We created a large corpus of 130,934 test methods from 1,270 open-source projects. We believe this corpus will also be useful to many other tasks related to software testing.
- **Evaluation**. Our extensive evaluation shows that TeCo significantly outperforms strong baselines on all automatic metrics, both on test completion and its sub-task: test oracle generation. We also evaluate the functional correctness of generated code by compiling and running the generated statements.

TeCo and our test completion corpus are publicly available on GitHub:
`https://github.com/EngineeringSoftware/teco`.

method under test

```
public GMOperation addImage(final File file) {
  if (file == null) {
    throw new IllegalArgumentException("file must be defined"); }
  getCmdArgs().add(file.getPath());
  return this;
}
```

test method signature

```
@Test
public void addImage_ThrowsException_WhenFileIsNull() throws Exception
```

prior statements

```
exception.except(IllegalArgumentException.class);
```

next statement

```
sut.addImage((File) null);
```

Figure 2.1: Example of test completion: given the code under test (represented by the method under test), test method signature, and prior statements, the goal is to generate the next statement. Code from `sharneng/gm4java` in class `GMOperationTest`.

2.2 Task

In this section, we more formally describe the test completion task and illustrate the task using an example.

Given an incomplete test method, our goal is to automatically generate the next statement in that test method. We assume that the following inputs are provided to a test completion system: (1) the code under test, which includes both the test method's associated method under test as well as other non-test-method code in the project, (2) the test method signature, (3) prior statements in the incomplete test method (which can be zero or more statements).

We illustrate our task in Figure 2.1. The example shows (in the yellow boxes) the method under test, the test method signature, and the prior statements (only one statement in this example), as well as (in the last green box) the next statement that should be generated by a test completion system.

We seek to generate statements in the body of the test method, thus the test method signature (including the annotation and the name of test method) are only

used as inputs and they are not the prediction target of the test completion task. We also do not consider the context of other already available test methods from the same project when completing a test method, to prevent any model from cheating by copying code from other similar test methods. Our defined test completion task is applicable to the situation when a developer already knows what to test (thus knows the method under test and the test method signature), and wants to complete the next statement at any point when writing the test method, regardless of whether the project has existing tests or not. We also focus on modeling the body of test methods as a sequence of statements, because test methods with control flows (e.g., if statements, loops, and try blocks) are rare; we found less than 10% test methods have control flows in our experiments. Most testing frameworks recommend sequential test method body and provide annotations to replace control flows, for example, `@ParameterizedTest` for replacing loops in JUnit 5 [98].

2.3 Extraction of Code Semantics

In this section, we describe the six kinds of code semantics extracted and used by TeCo.

For each kind of code semantics, we design and implement a static analysis algorithm to extract it. Static analysis is the analysis of code without executing it, guided by the grammar and semantics of programming languages. The advantage of using static analysis is that it does not require configuring the runtime environment which can be cumbersome for some projects, and can be applied on partial code (for example, without accessing the dependency libraries of a project, which is needed when executing the code). It is also much faster than executing the code directly, which enables us to collect code semantics on a large corpus of code. However, static analysis can sometimes be inaccurate; for example, when some values are unknown without executing code (e.g., user inputs), static analysis has to over-estimate the analysis results (e.g., assuming both branches of an if statement may be executed).

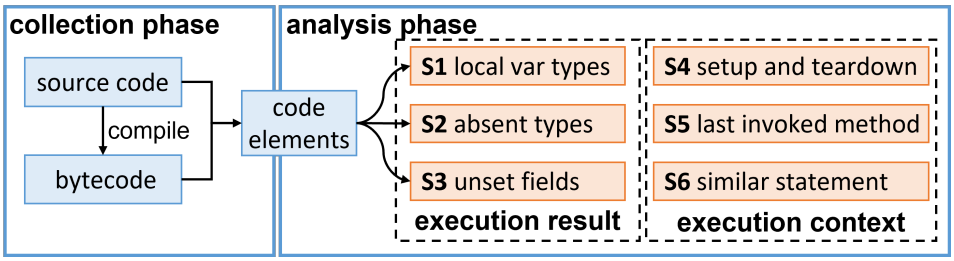


Figure 2.2: TeCo's workflow of extracting code semantics.

This may not be a problem for TeCo as the deep learning model can learn to ignore the inaccurate parts.

Figure 2.2 illustrates the general workflow of TeCo's static analysis which consists of two phases: given a project, in the collection phase TeCo collects a shared set of all code elements (classes, methods, fields), and in the analysis phase TeCo extracts each kind of code semantics from the code elements set using a specific algorithm. The code semantics can be organized into two categories based on their content: *execution result* and *execution context*. Execution result includes: (S1) local var types, (S2) absent types, and (S3) unset fields; execution context includes: (S4) setup and teardown, (S5) last invoked method, and (S6) similar statement. In the following paragraphs we describe the collection phase and the analysis phase for each kind of code semantics in more detail.

The collection phase. The goal of this phase is to collect a set of code elements that will be shared by all test methods and all six kinds of code semantics. TeCo collects three kinds of code elements: classes, methods, and fields; each method and field should have one class as its parent. For each element, TeCo collects its metadata, including name, type, access modifiers, annotations (if any), etc. For each class, TeCo records if it is a non-test class, test class, or a class in a library that is in the list of dependencies. TeCo additionally collects the source code and bytecode for all their methods of non-test and test classes. We do not utilize the source code or bytecode in dependencies because they are not needed for the analysis.

(S1) local var types data refers to the types of the local variables in the test method. The types are extracted by partially interpreting the bytecode of the test method without considering the values of variables. Note that this is more accurate than reading the local variable table which contains the declared types of local variables; for example, after interpreting the statement `AbstractWComponent comp = new SimpleComponent()`, the type of `comp` is `SimpleComponent`, which is more accurate than its declared type `AbstractWComponent`. This data provides information on what types of test inputs are available to be used in the next statement.

(S2) absent types are the types of the variables that are needed for invoking the method under test but have not been prepared in the test method. Types needed for invoking the method under test include its parameter types, plus the class under test (the declaring class of the method under test) if the method under test is not a static method. We consider a type as prepared if a local variable with this type has been initialized in prior statements, or a test class's field with this type has been initialized in prior statements or setup methods. This data focuses on what types of test inputs are missing, and thus may likely need to be prepared in the next statement.

(S3) unset fields data refers to the fields of the test class and the class under test that have not been initialized. We deem a field as initialized if there is any statement for setting the value of the field in the test method, the setup methods, or methods transitively invoked from the test method or the setup methods (up to 4 jumps, as initializations of the fields of interest further in call graph are rare). The fields in this data are likely to be initialized in the subsequent statements.

(S4) setup and teardown data refers to the source code of the setup and teardown methods in the test class. When a test framework executes tests, setup methods are executed before the test method to set up the environment (e.g., connection to a database), and teardown methods are executed after the test method to clean up the environment. By providing this context, the test completion system can know what environment is available to use in the test method, and also can avoid duplicating the

```
1  public class GMOperation extends org.im4java.core.GMOperation {
2    public GMOperation addImage(final File file) {...}
3  ...}                                          (S2) absent types
4
5  public class GMOperationTest {
6    GMOperation sut;
7    @Before
8    public void setup() {... sut = new GMOperation(); ...}
9                                          (S4) setup and teardown
10   @Test
11   public void addImage_ThrowsException_WhenFileIsNull() throws Exception {
12     exception.except(IllegalArgumentException.class);
13     sut.addImage((File) null);
14   }
15 ...}
```

Figure 2.3: Two kinds of code semantics (S2 and S4, highlighted in the top two blue boxes) extracted for the example in Figure 2.1, which help TeCo generate the correct next statement (highlighted in the bottom green box).

statements already in the setup and teardown methods.

(S5) last invoked method data is the source code of the last invoked method in the prior statements, which could be empty if no method has been invoked yet. This data provides more context on what has been executed in prior statements.

(S6) similar statement data is a statement in the non-test code of the project that has the most similar prior statements context to the prior statements in the incomplete test method. TeCo uses the BM25 algorithm [180] to search for similar prior statements. Only 2 prior statements are considered during the search, as increasing the window size leads to much longer search time without improving the quality of the returned similar statement. We expect this data to be similar to the next statement to be predicted.

Implementation. In the collection phase, TeCo uses JavaParser [96] to collect source code and ASM [36] to collect bytecode. The collection phase takes 136s per project on average. All analysis algorithms are implemented in Python, with the help of ASM for partially interpreting bytecode (for S1) and scikit-learn [163] for the BM25 algorithm. The analysis phase takes 0.018s–0.247s per test method on average, depending on the kind of code semantics data.

Example. Figure 2.3 shows two kinds of code semantics (S2 and S4) extracted for the example in Figure 2.1 that help the generating the correct next statement. The (S2) absent types data is `File`, because invoking the method under test `addImage` requires `GMOperation` (as `addImage` is not a static method) and `File`, but `GMOperation` is already available as a field `sut` in the test class (on line 6). The (S4) setup and teardown data is the method `setup` in the test class `GMOperationTest` which initializes the `sut` field (on line 8). Note that the other kinds of code semantics are either empty or not useful for this example, but are useful for some other examples.

2.4 TeCo's Deep Learning Model

This section describes the deep learning approach that TeCo uses to solve the test completion task. Figure 2.4 illustrates the overall model architecture: an encoder-decoder transformer model whose input includes both code semantics and syntax-level data and output is the prediction of the next statement.

2.4.1 Encoder-Decoder Transformer Model

Our model is based on the encoder-decoder architecture that considers both input and output as sequences, which has been applied to many sequence generation tasks including code summarization [4, 94, 109] and code generation [121, 209]. In the context of TeCo, the input is the syntax-level data (method under test, test signature, and prior statements) plus the code semantics extracted from the test to be completed (S1-S6), and the output is the next statement.

More formally, TeCo is given the test to be completed T and the code under test C as inputs, where C includes the method under test x_{mut}, and T consists of two parts: the test signature x_{sign} and prior statements x_{prior}. The goal is to generate the next statement y. TeCo extracts code semantics as described in Section 2.3:

$$x_{S1}, x_{S2}, x_{S3}, x_{S4}, x_{S5}, x_{S6} = \texttt{analysis}(T, C)$$

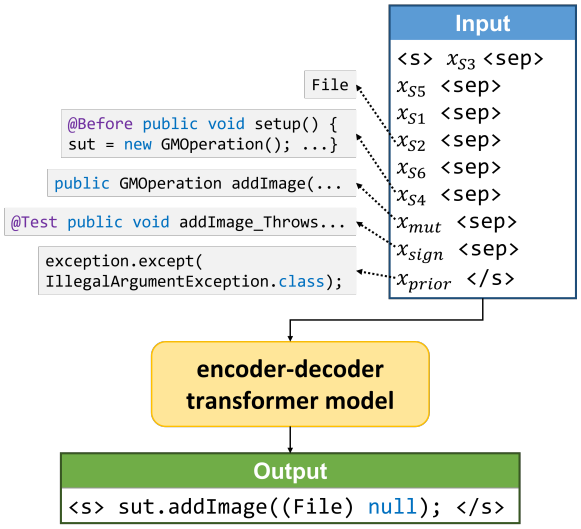


Figure 2.4: TeCo's model architecture.

Each input piece x and output y is a sequence of subtokens, which is obtained by subtokenizing the code or extracted data's string format using the BPE (byte-pair encoding) algorithm [170, 187]. The goal of subtokenization is to break down long unique identifier names into small common subtokens so that the model can learn better, for example, the identifier `GMOperation` is broken down into three subtokens `G`, `M`, and `Operation` by TeCo's BPE algorithm. TeCo combines the input pieces into a single input sequence by concatenating them with a delimiter ⟨`sep`⟩ (the ordering of the sequences is configurable):

$$x = x_{S3}\ \langle\texttt{sep}\rangle\ x_{S5}\ \langle\texttt{sep}\rangle\ x_{S1}\ \langle\texttt{sep}\rangle\ x_{S2}\ \langle\texttt{sep}\rangle\ x_{S6}\ \langle\texttt{sep}\rangle$$
$$x_{S4}\ \langle\texttt{sep}\rangle\ x_{mut}\ \langle\texttt{sep}\rangle\ x_{sign}\ \langle\texttt{sep}\rangle\ x_{prior}$$

The maximum number of subtokens that TeCo can accept, due to the limitation of the underlying model, is 512. If the input sequence is longer than that, TeCo keeps the last 512 subtokens. In our experiments, the number of subtokens in the input sequence ranges from 23 to 2,426 (average: 243.88) and exceeds the limitation of 512 subtokens in 6% of the cases. As a result, more important information should be placed at the end of the input sequence to avoid being truncated. The ordering

of code semantics is decided based on our domain knowledge of which kind of data would contain more important information, and we always put the syntax-level data at the end (as they deliver the basic information for the task). Exploring all possible orderings may discover better models but is too computationally expensive. We plan to investigate the impact of the orderings by designing experiments with more affordable costs in the future.

Then, TeCo uses a transformer model [200] to learn the conditional probability distribution $P(y|x)$. The model computes this by first using an encoder to encode the input sequence into a deep representation, and then using a decoder which reads the deep representation and generates the output sequence one subtoken at a time:

$$h = \texttt{encoder}(x)$$
$$P(y[i]|y[:i], x) = \texttt{decoder}(y[:i], h), \text{for each } i$$

2.4.2 Fine-tuning

Recent work shows that pre-training a large-scale transformer model on a large corpus of code and text and then fine-tuning the model on downstream tasks lead to better performance than training a model from scratch [5, 40, 60, 206]. However, pre-training is only performed on syntax-level data in order to be generalizable to many downstream tasks with different semantics. Prior work shows that large-scale pre-trained models may not perform well on simple tasks of executing the code [149]. This indicates that they learned little about code semantics.

We believe that fine-tuning on code semantics is vital for pre-trained models to perform well on execution-related tasks such as test completion. During pre-training, the model mainly learns the syntax and grammar of programming languages. If only syntax-level data is used during fine-tuning, the model would have to infer the semantics of execution, which can be very inaccurate because the execution can be complicated and the context provided by the syntax-level data is limited. By contrast, code semantics are extracted using reliable static analysis algorithms in TeCo so that

the model can directly use such data instead of inferring. Thus, during fine-tuning of TeCo, the model learns how to understand and process code semantics in addition to the syntax-level data.

For the pre-trained model, we use CodeT5 [206], which is a large-scale encoder-decoder transformer model pre-trained on a bimodal corpus of code in multiple programming languages (including Java) and text. We fine-tune the model on a corpus for test completion $\mathcal{TC}$ by minimizing the cross-entropy loss:

$$\texttt{loss} = \sum_{x,y \in \mathcal{TC}} -\log P(y|x) = \sum_{\substack{x,y \in \mathcal{TC} \\ i \in [0,|y|)}} -\log P(y[i]|y[:i], x)$$

2.4.3 Inference

At inference time, TeCo uses the beam search algorithm with a beam size of 10. Specifically, starting from a special begin-of-sequence subtoken $y[0] = \langle \texttt{s} \rangle$, TeCo iteratively runs `decoder` to generate the most likely next subtokens which is appended to the output sequence; only the top 10 sequences with the highest total probability are kept at each step. Each output sequence is completed upon generating a special end-of-sequence subtoken $\langle / \texttt{s} \rangle$. The beam search terminates after generating 10 completed output sequences. As repeating the same subtoken rarely happens in practice, we apply a repetition prevention mechanism that penalizes the probability of generating the same subtoken as the previous subtoken [101].

2.4.4 Reranking by Execution

The model can generate plausible outputs by maximizing the generation probability that it learnt during fine-tuning. However, there is no guarantee for the generated statements—subtoken sequences—to be compilable and runnable code. Arguably, generating compilable and runnable code is more important than generating plausible but non-executable code in the test completion task, because it is crucial for developers observe the runtime behavior to check and revise the predictions.

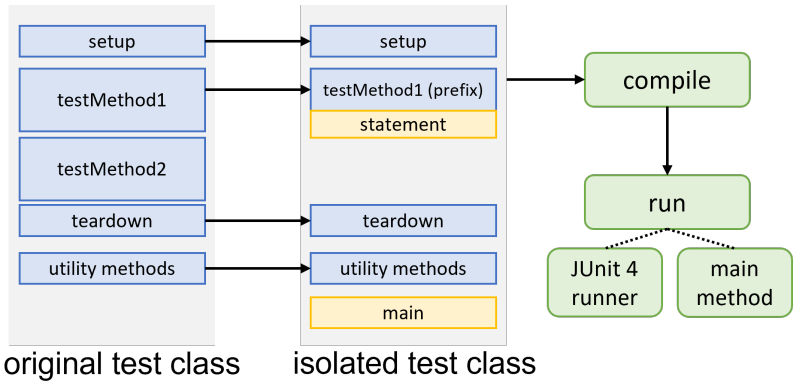


Figure 2.5: Procedure of detecting whether a statement is compilable and runnable.

We propose to use *reranking by test execution* to improve the quality of the generated statements. Specifically, after collecting the top-10 predictions from beam search ranked by their probabilities, denoted as $\mathcal{Y}$, TeCo checks whether each of them is compilable and runnable. Then, TeCo reranks the outputs into $\hat{\mathcal{Y}}$ where one output y_i is ranked higher than another y_j if: (1) y_i is runnable and y_j is not; or (2) both are not runnable, but y_i is compilable and y_j is not; or (3) both have the same runnable and compilable status, and $P(y_i) > P(y_j)$. In this way, generated statements that are compilable and runnable are prioritized over the others.

TeCo detects whether a generated statement is compilable and runnable by putting it in a synthesized test class with the required context, isolated from being affected by other test methods in the same project. The procedure is illustrated in Figure 2.5, specifically:

1. Create an isolated test class with a test method using the signature and prior statements, followed by the generated statement.
2. Extract the other non-test methods from the original test class (including setup, teardown, and utility methods) into the isolated test class.
3. Generate an ad-hoc main method in the isolated test class which calls setup methods, the test method, and teardown methods in order.

4. Compile the isolated test class with all the dependencies specified in the project's build configuration as well as all non-test classes in the project.
5. If the compilation succeeds (at which point the statement is considered to be compilable), execute the compiled test class; the statement is considered to be runnable only if there is no unexpected exception or assertion failure during the execution (catching expected exception, e.g., via `exception.except(...)`, is considered as runnable). The execution of the isolated test class includes two steps:
   (a) If the project is using JUnit 4 [97] testing framework, we try to use its command line runner to run the isolated test class; the JUnit 4 runner is more robust because it properly utilizes all JUnit features that our ad-hoc main method does not handle, e.g., `@RunWith`.
   (b) If the project is not using JUnit 4 or running with JUnit 4 runner failed (i.e., JUnit 4 runner considers the statement as not runnable, which is sometimes a false alarm), we run the ad-hoc main method.

2.5 Corpus

As test completion is a new task, we construct a large corpus that can serve as a testbed for our work and future research. We collected data from the same subject projects used by CodeSearchNet [91], which is a large corpus of code and comments that is frequently used in ML + code research [60, 125, 206]. Out of the 4,767 Java projects in CodeSearchNet, we used the 1,535 projects that: (1) use the Maven build system (for the simplicity of data collection; TeCo is not limited to any build system); (2) compile successfully, and (3) have a license that permits the use of its data. We collected the corpus in Spring 2022. To ensure corpus quality, we try to use the latest stable revision of each project by finding its latest git-tag; but if it does not have any git-tag on or after Jan 1st, 2020, we use its latest revision.

To extract test methods from these projects, we first collected the set of code elements from each project using the same toolchain for the collection phase

of TeCo's static analysis (Section 2.3). We identified the test methods written in JUnit 4 [97] and JUnit 5 [98] testing frameworks, which are the main frameworks used for writing tests in Java. Specifically, we searched for methods with a test annotation (@`org.junit.Test` or @`org.junit.jupiter.api.Test`) and without an ignored-test annotation (@`org.junit.Ignore` or @`org.junit.jupiter.api.Disabled`). This initial search resulted in 221,666 test methods in all projects.

Then, we further filtered the test methods to ensure corpus quality. We filtered test methods that are badly named (e.g., `test0`; 2,490 cases) and that do not follow the method signature of common JUnit non-parameterized test methods (e.g., parameter list is not empty, return type is not void; 1,908 cases). Then, we tried to locate the method under test for each test method, using the following enhanced procedure originally proposed by Watson et al. [207]:

1. If there is only one invocation to a method, select it as the method under test;
2. If a class under test can be found by removing "Test" from the test class's name:
   (a) If there is only one invocation to a method declared in class under test, select it as the method under test;
   (b) Select the last method declared in class under test invoked before the first assertion statement, if any;
3. Select the last invoked called before the first assertion statement, if any;
4. Select the last invoked called, if any.

We removed 36,818 test methods for which we could not locate the method under test after this procedure.

We used the line number table to find the bytecode instructions corresponding to each statement, and we removed 633 cases where we could not do this because of multiple statements on the same line. We also removed 5 corner cases where the source code of the test method was not properly collected by JavaParser. After that, we set size constraints on the data: the test method should have at most 20 statements (filtered 8,222 cases); the method under test should have at most 200 tokens (filtered

9,787 cases); the method under test and the test method together should have at most 400 tokens (filtered 1,288 cases); each statement in the test method should have at most 100 tokens (filtered 1,726 cases).

We also removed several cases that introduce extra overhead during analysis: test methods with if statements, loops, and try blocks, because they entail non-sequential control flow which is not suitable to be modeled by predicting the next statement given prior statements (22,435 cases); and test methods using lambda expressions [154], because they complicate many static analysis algorithms (5,420 cases). We plan to lift these limitations in future work.

Lastly, we masked the string literals in data by replacing them with a common token "STR", similar to prior work on code completion [196]. Although string literals are frequently used in test methods, for example as logging messages, test inputs, or expected outputs, they pose challenges for a pure-deep-learning solution to generate because they have a different style than other parts of the code and can sometimes be very long. Thus, we focus on predicting the next statement with masked string literals, and leave predicting the content of the string literals as future work.

After filtering, we have a corpus with 1,270 projects (removed 265 projects because no data was left after filtering), 130,934 test methods, and 645,633 statements.

We follow the same project-level training/validation/evaluation split as CodeSearchNet. Because CodeT5, the pre-trained model that TeCo uses, also followed the same project-level split, our experiments will not have data leakage issues of evaluating on the data that the model was pre-trained on. Table 2.1 (top part) shows the statistics of our corpus, where the first row (All) is for the entire corpus, and the next three rows (Training, Validation, Evaluation) are for each set after the split. Out of the 130,934 test methods, 101,965 (77.88%) are runnable following our procedure described in Section 2.4.4. Note that with the masking of string literals, some test methods that would originally pass may be considered as "not runnable" in our current corpus (e.g., when the test method compares a variable with a string literal).

Table 2.1: Statistics of our test completion corpus. #proj = number of projects; #test = number of test method; #stmt = number of statements; len(test) = average number of tokens in test method; len(MUT) = average number of tokens in method under test.

	#proj	**#test**	**#stmt**	**len(test)**	**len(MUT)**
All	1,270	130,934	645,633	79.57	40.88
Training	1,163	120,521	584,924	79.58	40.61
Validation	43	5,413	30,515	73.24	45.09
Evaluation	64	5,000	30,194	86.26	42.85
Evaluation, runnable subset	55	4,223	25,074	83.17	42.87
Evaluation, oracle subset	61	4,212	4,212	92.59	40.90
Evaluation, oracle-runnable subset	51	3,540	3,540	89.19	40.54

2.6 Experiments Setup

We evaluate TeCo by answering the following research questions:

RQ1: What is the performance of TeCo on the test completion task and how does it compare to baselines?

RQ2: On the runnable subset of evaluation set, how frequently can TeCo predict a compilable and runnable next statement?

RQ3: What is the performance of TeCo on test oracle generation, which is a sub-task of test completion, and how does it compare to prior work?

RQ4: How does reranking by execution help with more accurately predicting the next statement?

RQ5: How does each kind of code semantics help with more accurately predicting the next statement, and how complementary are different kinds of code semantics?

To answer these questions, we set up an experiment to evaluate TeCo and baseline models on our test completion corpus. We train each model on the training and validation sets (validation set is used for tuning hyper-parameters of the model

and early stopping[2]), apply the model to predict each statement of each test method in the evaluation set (or subsets of the evaluation set), and measure the quality of the prediction via a number of evaluation metrics, both intrinsically and extrinsically.

All models are trained and evaluated on a cluster of machines each equiped with 2x Intel(R) Xeon(R) E5-2620 v4 @ 2.10GHz CPUs, 4x NVidia 1080-TI GPUs, and 128GB RAM. We ran each experiment three times with different random seeds and report average values. When comparing models, we conducted statistical significance tests using bootstrap tests [24] with a 95% confidence level.

We next describe the TeCo models (Section 2.6.1) and baseline models (Section 2.6.2) used in the experiments, the subsets of the evaluation set for computing compilable and runnable metrics and evaluating on the test oracle generation task (Section 2.6.3), and the evaluation metrics (Section 2.6.4).

2.6.1 TeCo Models

We run a TeCo model that uses all six kinds of code semantics and with reranking by test execution. To study RQ4, we run a TeCo-noRr model that uses the same code semantics but does not use reranking. To study RQ5, we run six TeCo models with only one kind of code semantics at a time, which we call TeCo-KIND (e.g., TeCo-S1 only uses S1).

2.6.2 Baseline Models

We compare our TeCo models to the following baseline models that only use syntax-level data.

[2]Early stopping is a common strategy to mitigate overfitting in training an ML model by monitoring the model's performance on both the training set and the validation set and halting the training if the model stops improving on the validation set even if it improves on the training set. If early stopping is not used, the model is fully trained to maximize its performance on the training set, but may have bad performance on a separate set (e.g., evaluation set). In our experiments, we set an early stopping threshold of 3, which means the training is halted if the model does not obtain smaller loss on the validation set for 3 consecutive checkpoints.

CodeT5 [206] is a pre-trained encoder-decoder transformer model for code-related tasks, and is built on top of Google's popular T5 framework [171]. CodeT5 was pre-trained on eight commonly used programming languages (including Java) using both mask language modeling and identifier name recovering tasks. We fine-tune TeCo models based on CodeT5. As such, we compare to a baseline CodeT5 model that is fine-tuned on our test completion corpus using syntax-level data. For completeness, we also compare to a CodeT5-noFt that is only pre-trained and not fine-tuned.

CodeGPT [125] is a pre-trained decoder-only transformer model built on GPT-2 [170]. Svyatkovskiy et al. [196] trained and evaluated a very similar model (which is not publicly available) for code completion. We compare to a CodeGPT model that is pre-trained on natural language and Java code (i.e., the `java-adapted` version), and then fine-tuned on our test completion corpus using syntax-level data. As CodeGPT tends to generate longer code than a statement (without generating the ⟨/s⟩ subtoken to stop the generation), we slightly modify its decoding algorithm to terminate upon generating the first ';' subtoken for the test completion task.

Codex [40] is a pre-trained decoder-only transformer model built on GPT-3 [34]. At the time of experiment, the only way of using Codex was through OpenAI's API[3], and fine-tuning Codex was not possible. We compare to the largest pre-trained Codex with 12B parameters (`code-davinci-002`); the pre-training dataset of Codex is unknown, but usually perceived as all publicly available code from GitHub [40, 129], which is much larger than the pre-training dataset of CodeT5 and CodeGPT. We use Codex to perform test completion in the zero-shot learning setup [104], i.e., providing Codex with a prompt that contains the method under test, test method signature, and prior statements, and letting it generate the next statement. Because Codex is pre-trained to perform code completion, the prompt needs to be carefully designed as a code fragment to be completed. Figure 2.6 illustrates the prompt format we used.

[3] `https://platform.openai.com/docs/guides/code`, which we accessed in March 2023; unfortunately, this API for using Codex has been deprecated at the time of writing.

```
public GMOperation addImage(final File file) {
  if (file == null) {
    throw new IllegalArgumentException("file must be defined");
  }
  getCmdArgs().add(file.getPath());
  return this;
}

// Here is a test for the above method. Please complete the next statement of the test
@Test public void addImage_ThrowsException_WhenFileIsNull() throws Exception {
exception.except(IllegalArgumentException.class);
// Please compete the next statement:
```

Figure 2.6: The prompt format we designed for using Codex to perform test completion, including the method under test, test method signature, prior statements, and some comments to guide the model.

We configure Codex to generate until seeing the first ';', similar to the way we used CodeGPT. Because the inference time of Codex is quite high, we configure Codex to only generate the top-1 next statement using the greedy decoding algorithm. Running Codex on our evaluation set (with 30,194 statements) took 18 hours.

Test oracle generation is the task of generating the assertion statement given the code under test (including the method under test), test method signature, and prior statements before the assertion statement. When studying this task, we additionally compare to the following two deep learning baseline models for test oracle generation developed in prior work, both of which only use syntax-level data. Following prior work that proposed the baseline models, we only consider generating the first assertion statement in each test method.

ATLAS [207] is an RNN encoder-decoder model for test oracle generation. We used the "raw model" version of it, i.e., that does not abstract out the identifiers in code.

TOGA [55] is a transformer encoder-only model for classifying the suitability of an assertion statement for an incomplete test method without assertions. It can be used for test oracle generation by first generating a set of assertion statements and then using the model to rank them and select the best one. The ranking model is initialized from CodeBERT [60], which is also pre-trained on the CodeSearchNet corpus [91].

For all baseline models, we use the default hyper-parameters and training configurations recommended by the authors. We fine-tune CodeT5 and CodeGPT on the entire training and validation set of our corpus. We train/fine-tune ATLAS and TOGA on a subset of our training and validation set that only predicts the first assertion statement in each test method, which contains 92,567 statements and 3,050 statements, respectively.

2.6.3 Subsets of the Evaluation Set

To study the ability of models in predicting a compilable and runnable next statement, we evaluate models on the runnable subset. That is, the subset of the evaluation set where the gold (i.e., developer-written) statement is runnable. We follow the same procedure to check if the gold statement is runnable as described in Section 2.4.4. Not all gold statements can be successfully executed because of the difficulties in setting up the proper runtime environment, such as missing resources (that may need to be downloaded or generated via other commands), requiring other runtime environments than Java, etc. Our runnable subset contains 25,074 statements (83.04% of all statements in the evaluation set) from 4,223 test methods.

To study the test oracle generation task, we evaluate models on the oracle subset: the subset of the evaluation set where the statement to generate is the first assertion statement in the test method, which contains 4,212 statements. To compute compilable and runnable metrics on the test oracle generation task, we evaluate models on the oracle-runnable subset: the subset of the oracle subset where the gold statement is runnable, which contains 3,540 statements.

Table 2.1 (bottom part) shows the statistics of the runnable subset, oracle subset, and oracle-runnable subset subsets of the evaluation set, including number of projects and test methods (after removing the projects and test methods that do not have any statement that belongs to the subset), number of statements, and average length of test methods and methods under test.

2.6.4 Evaluation Metrics

(1) Lexical-level metrics: We use the following metrics to measure how close the predicted statements are to the gold statements; these metrics have been frequently used in prior work on code generation and comment generation [89, 94, 109, 118]:

Exact-match accuracy (XM) is the percentage of predicted statements matches exactly with the gold. This metric is the most strict one; each point of improvement directly entails a larger portion of code that is both syntactically and semantically correct, yet it does not take into account paraphrases or give any partial credit.

Top-10 accuracy (Acc@10) is the percentage of any top-10 predicted statements matches exactly with the gold. This metric evaluates the use case where the developer can see and select from the top-10 predictions of the model.

BLEU [160] calculates the number of n-grams (consecutive n subtokens) in the prediction that also appear in the gold; specifically, we compute the $1\sim4$-grams overlap between the subtokens in the prediction and the subtokens in the gold, averaged between $1\sim4$-grams with smoothing method proposed by Lin and Och [119].

CodeBLEU [176] is an improved version of BLEU adapted for code. It is a combination of the traditional BLEU, the BLEU if only considering keywords, syntactical AST match, and semantic data-flow match.

Edit similarity (EditSim) = 1 - Levenshtein edit distance, where the Levenshtein edit distance measures the amount of single-character edits (including insertion, substitution, or deletion) that need to be made to transform the prediction to the gold, normalized by the maximum number of characters in the prediction and the gold. This metric was proposed and used in prior work on code completion [196].

ROUGE [120] measures the overlap between the prediction subtokens and the gold subtokens based on the Longest Common Subsequence statistics, using F1 score.

(2) Functional correctness: The aforementioned metrics only capture the lexical similarity between the prediction against the gold, but the gold statement may not be the only correct solution for competing the next statement. Namely, the prediction can be functionally correct despite being different from the gold statement. To measure the functional correctness, we additionally use the following metrics:

%Compile is the percentage of the predicted statements that are compilable when appended to the incomplete test.

%Run is the percentage of the predicted statements that are compilable and runnable when appended to the incomplete test, without incurring assertion failures or runtime errors.

Note that %Compile and %Run are over-estimations of the functional correctness: %Compile detects only compilation errors; %Run is more strict and checks for runtime errors including unexpected exceptions and assertion failures. Neither metric considers whether the underlying logic of the code is meaningful to developers, which may need to be evaluated with a costly user study. Prior work has used a similar methodology to evaluate the functional correctness of text-to-code transduction by running generated code with test cases [40], which was performed on a small dataset because of the difficulty in collecting manually labeled data. Thanks to the executable nature of tests, we are able to design the two automatic functional correctness metrics for a large corpus.

2.7 Results

This section presents the results of TeCo and baseline models on the test completion task, and answers the research questions from Section 2.6.

Table 2.2: Test completion performance of TeCo and baseline models. The best number for each metric is bolded. In each table, numbers marked with the same greek letter prefix are not statistically significantly different.

(a) On the evaluation set.

Model	XM	Acc@10	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	13.57	24.11	38.33	33.88	60.81	62.40
CodeT5-noFt	0.00	0.00	0.00	3.36	1.68	0.02
CodeGPT	12.20	22.67	36.30	31.84	59.09	61.10
Codex	12.69	N/A	34.53	29.91	58.08	56.04
TeCo	**17.61**	**27.20**	**42.01**	**37.61**	**63.49**	**65.23**

(b) On the runnable subset.

Model	%Compile	%Run	XM	Acc@10	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	54.84	17.62	14.38	25.39	39.26	34.55	61.36	63.15
CodeT5-noFt	0.00	0.00	0.00	0.00	0.00	3.35	1.65	0.03
CodeGPT	53.77	15.13	12.95	24.03	37.19	32.46	59.75	61.90
Codex	38.80	19.12	12.88	N/A	34.89	30.12	58.44	56.58
TeCo	**76.22**	**28.63**	**18.96**	**28.40**	**43.15**	**38.45**	**64.12**	**66.09**

(c) On the oracle subset.

Model	XM	Acc@10	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	8.45	24.04	39.03	31.03	66.50	66.63
CodeT5-noFt	$^{\alpha}$0.00	0.00	0.00	1.18	1.86	0.01
CodeGPT	10.56	27.19	40.91	33.33	67.63	67.94
Codex	12.30	N/A	35.09	30.24	59.59	57.67
ATLAS	$^{\alpha}$0.21	0.66	21.55	13.39	54.06	50.70
TOGA	9.01	9.01	25.46	24.73	29.60	28.06
TeCo	**16.44**	**27.41**	**43.09**	**35.88**	**68.05**	**68.71**

(d) On the oracle-runnable subset.

Model	%Compile	%Run	XM	Acc@10	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	44.45	16.87	8.79	24.37	40.26	32.23	67.08	67.42
CodeT5-noFt	0.00	0.00	$^{\alpha}$0.00	0.00	0.00	1.26	1.87	0.01
CodeGPT	47.39	16.13	10.41	**28.17**	41.56	33.79	67.78	68.43
Codex	39.46	20.73	12.15	N/A	35.05	30.12	59.79	57.86
ATLAS	3.62	1.45	$^{\alpha}$0.23	0.68	21.81	13.56	54.19	50.87
TOGA	25.61	9.37	9.10	9.10	26.51	25.74	31.00	29.38
TeCo	**67.93**	**30.29**	**17.37**	27.39	**44.27**	**36.98**	**68.43**	**69.35**

2.7.1 RQ1: Performance of TeCo vs. Baseline Models

Table 2.2a shows the results of TeCo and baseline models on solving the test completion task. Our model, TeCo, significantly outperforms all baseline models on all metrics. TeCo achieves 17.61% exact-match accuracy, which is 29% higher than the best baseline model, CodeT5's 13.57%. This indicates that using code semantics and reranking by execution can greatly improve deep learning model's performance on test completion.

Both CodeT5-noFt and Codex are not fine-tuned. CodeT5-noFt is not capable of solving test completion task at all, as it was optimized to solve different tasks during pre-training and does not have the domain knowledge of the input-output format of the test completion task. On the contrary, Codex achieves performance on par with the fine-tuned baseline CodeT5, because it is a larger model and has been pre-trained on much more data.

CodeGPT has shown to be effective on the task of code completion [125, 196], where the primary goal is to continue generating code similar to the context code. However, it performs slightly worse than the encoder-decoder baseline CodeT5 on test completion, because the task requires generating statement in the test method which has different style than the method under test in the provided context.

To summarize, the following designs can improve the model's performance on test completion: integration with execution (comparing TeCo vs. CodeT5), scaling up the model size and pre-training data (comparing Codex vs. CodeT5-noFt), fine-tuning on the test completion dataset (comparing CodeT5 vs. CodeT5-noFt), and using encoder-decoder architecture instead of decoder-only architecture (comparing CodeT5 vs. CodeGPT). We believe these improvements are mostly orthogonal, which means TeCo's performance can be further improved by replacing its underlying model from CodeT5 to a larger pre-trained model.

2.7.2 RQ2: Functional Correctness

Table 2.2b shows the results of TeCo and baseline models on the runnable subset, with %Compile and %Run metrics that measure the functional correctness of the generated statements. Our model, TeCo, can generate runnable statements 28.63% of the time, and compilable statements 76.22% of the time, which are much higher than the best baseline model's 19.12% (Codex) and 54.84% (CodeT5). On this runnable subset, TeCo also outperforms all baseline models on other metrics measuring lexical similarity.

2.7.3 RQ3: Performance on Test Oracle Generation

Tables 2.2c and 2.2d show the results of the test oracle generation sub-task, on the oracle subset and the oracle-runnable subset, respectively. TeCo significantly improves the exact-match accuracy on this task by a large margin (by 33%), from 12.30% for the best baseline (Codex), to ours 16.44%.

TOGA, the best model designed specifically for test oracle generation in prior work, achieves 9.01%, which is on par with CodeT5. However, it is worse than CodeT5 on other metrics that consider partial matches. This is because TOGA is a classification model that ranks a set of assertion statement candidates generated using heuristics, and when the gold statement is not in the set, the model fails to correctly rank a sub-optimal candidate.

2.7.4 RQ4: Improvements from Reranking by Execution

Table 2.3 shows the results of TeCo-noRr (the top-10 accuracy for TeCo-noRr is always the same as TeCo, because the reranking is performed on top-10 predictions, thus we did not include this metric in the table).

Comparing TeCo with TeCo-noRr on the evaluation set (Table 2.3a), reranking by execution alone contributes to 2 points in exact-match accuracy. However, the improvements over other similarity metrics, which take into account partial matches,

Table 2.3: Results for TeCo without and with reranking by execution. The best number for each metric is bolded. The differences between models for each metric are statistical significant.

(a) On the evaluation set.

Model	XM	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	13.57	38.33	33.88	60.81	62.40
TeCo-noRr	15.25	40.84	36.34	62.92	64.71
TeCo	**17.61**	**42.01**	**37.61**	**63.49**	**65.23**

(b) On the runnable subset.

Model	%Compile	%Run	XM	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	54.84	17.62	14.38	39.26	34.55	61.36	63.15
TeCo-noRr	60.80	19.49	15.99	41.64	36.82	63.38	65.42
TeCo	**76.22**	**28.63**	**18.96**	**43.15**	**38.45**	**64.12**	**66.09**

(c) On the oracle subset.

Model	XM	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	8.45	39.03	31.03	66.50	66.63
TeCo-noRr	9.92	40.81	32.90	67.32	67.92
TeCo	**16.44**	**43.09**	**35.88**	**68.05**	**68.71**

(d) On the oracle-runnable subset.

Model	%Compile	%Run	XM	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	44.45	16.87	8.79	40.26	32.23	67.08	67.42
TeCo-noRr	48.13	18.45	9.62	41.55	33.44	67.57	68.41
TeCo	**67.93**	**30.29**	**17.37**	**44.27**	**36.98**	**68.43**	**69.35**

are smaller. This indicates that reranking by execution is effective in prioritizing the exact correct generated statement than other non-runnable candidates most of the times, but in a few cases it may prioritize runnable candidates that are less similar to the gold statement than the original top-1. TeCo-noRr still significantly outperforms CodeT5 on all metrics. On the runnable subset (Table 2.3b), TeCo improves both %Compile and %Run over TeCo-noRr by large margins, which shows that reranking by execution is an effective strategy for improving the quality of generated statements.

Table 2.4: Results for TeCo models with only one kind of code semantics on the evaluation set. The best number for each metric is bolded.

(a) On the evaluation set.

Model	XM	Acc@10	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	13.57	24.11	38.33	33.88	60.81	62.40
TeCo-S1	13.88	24.93	39.12	34.66	61.58	63.51
TeCo-S2	14.06	25.11	**39.56**	**35.17**	**62.20**	**63.92**
TeCo-S3	14.04	24.40	38.81	34.24	61.21	62.87
TeCo-S4	**14.44**	**25.55**	39.39	35.00	61.63	63.40
TeCo-S5	14.05	24.78	38.74	34.34	61.26	63.00
TeCo-S6	14.13	24.74	38.70	34.36	60.86	62.52

(b) On the oracle subset.

Model	XM	Acc@10	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	8.45	24.04	39.03	31.03	66.50	66.63
TeCo-S1	8.86	24.37	38.03	29.93	65.59	65.95
TeCo-S2	8.23	23.69	38.13	30.27	65.52	65.98
TeCo-S3	8.72	23.57	**39.90**	31.96	**67.20**	**67.44**
TeCo-S4	8.14	23.84	38.54	30.77	65.59	65.92
TeCo-S5	8.43	24.10	38.67	30.85	66.17	66.23
TeCo-S6	**9.81**	**25.47**	39.88	**32.20**	66.89	66.97

Reranking by execution ended up being very important for improving performance on the task of test oracle generation, as shown on the oracle subset (Table 2.3c) and the oracle-runnable subset (Table 2.3d). For example, TeCo outperforms TeCo-noRr by 6–8 points in exact-match accuracy and 12 points in %Run. This is because logical errors in assertion statements can be easily found by execution (e.g., generating the wrong expected value will cause an assertion to fail).

2.7.5 RQ5: Comparisons of Code Semantics

Tables 2.4a and 2.4b show the results of the TeCo models with only one kind of code semantics, comparing with the strongest baseline model CodeT5, on the full evaluation set and the oracle subset, respectively. We did not perform statistical significance tests for the results here as the performances of the models are too similar.

Each model outperforms CodeT5 on at least one metric, meaning that each code semantics provides some information useful for test completion. In Table 2.4a, TeCo-S2 (absent types) is the best model in terms of BLEU, CodeBLEU, EditSim and ROUGE metrics, and TeCo-S4 (setup and teardown) is the best model in terms of exact-match accuracy and top-10 accuracy, which indicates that these two kinds of code semantics are relatively more important than others. Interestingly, in Table 2.4b, the models that achieved the best performance among single-data models changed: TeCo-S3 (unset fields) is the best model in terms of BLEU, EditSim, and ROUGE, and TeCo-S6 (similar statement) is the best model in terms of exact-match accuracy, top-10 accuracy, and CodeBLEU. Thus, different kinds of code semantics provide complementary information for test completion.

2.7.6 Qualitative Analysis

We perform a small qualitative analysis of TeCo's completed statements compared to baseline models using the examples shown in tables 2.5, 2.6 and 2.7. For each example, we show the method under test, test method signature, prior statements, the gold statement, and the statement generated by {TeCo, CodeT5, CodeGPT, and Codex}.

In the example shown in Table 2.5 (which is the same example used in figures 2.1 and 2.3), TeCo is able to generate the correct next statement thanks to the code semantics (S2) absent types and (S4) setup and teardown, as we pointed out in Section 2.3. On the contrary, CodeT5 generates a wrong statement that repeats some part of the setup method and does not use `File` correctly; and CodeGPT does not correctly use the `sut` field prepared in the setup method. Codex generates a code comment instead of a statement, as it was not fine-tuned to perform the test completion task.

In the example shown in Table 2.6, TeCo generates the correct next statement, primarily because the statement is included in the code semantics (S6) similar

Table 2.5: Example of test completion results by TeCo and baseline models, for `addImage_ThrowsException_WhenFileIsNull` in `sharneng/gm4java`, `GMOperationTest.java`.

Method under test

```
public GMOperation addImage(final File file) {
  if (file == null) {
    throw new IllegalArgumentException("STR");
  }
  getCmdArgs().add(file.getPath());
  return this;
}
```

Test method signature

```
@Test public void addImage_ThrowsException_WhenFileIsNull() throws Exception
```

Prior statements

```
exception.expect(IllegalArgumentException.class);
```

Gold

```
sut.addImage((File) null);
```

TeCo CodeBLEU = 100

```
sut.addImage((File) null);
```

CodeT5 CodeBLEU = 13

```
new GMOperation().addImage(null);
```

CodeGPT CodeBLEU = 9

```
command.addImage(new File("STR"));
```

Codex CodeBLEU = 1

```
// rpcClient.shutdown() ;
```

statement (i.e., the statement that appears the most of the times after the prior statement `ContainerHandler containerHandle = getContainerHandler()` in the project is `PodTemplateHandler podTemplateHandler = newPodTemplateHandler(containerHandler)`). The baseline models generate incorrect statements without knowing this information.

In the example shown in Table 2.7, the statement to be completed is an assertion statement that should check whether `count` equals to 1. CodeT5 and CodeGPT use the `asserThat` API, which seems plausible (and indeed compiles) but is not runnable. Codex again generates a code comment due to lack of fine-tuning. TeCo is able to generate the correct assertion statement using the `assertEquals` API.

Table 2.6: Example of test completion results by TeCo and baseline models, for `statefulSetHandlerWithoutControllerTest` in `fabric8io/fabric8-maven-plugin`, `StatefulSetHandlerTest.java`.

Method under test

```
public StatefulSet getStatefulSet(ResourceConfig config, List<ImageConfiguration> images) {
  return new StatefulSetBuilder().withMetadata(createStatefulSetMetaData(config))
    .withSpec(createStatefulSetSpec(config, images)).build();
}
```

Test method signature

```
@Test(expected=IllegalArgumentException.class) public void statefulSetHandlerWithoutControllerTest()
```

Prior statements

```
ContainerHandler containerHandler = getContainerHandler();
```

Gold

```
PodTemplateHandler podTemplateHandler = new PodTemplateHandler(containerHandler);
```

TeCo CodeBLEU = 100

```
PodTemplateHandler podTemplateHandler = new PodTemplateHandler(containerHandler);
```

CodeT5 CodeBLEU = 19

```
StatefulSet statefulSet = new StatefulSetBuilder()
  .withMetadata(createStatefulSetMetaData(config, images))
  .withSpec(createStatefulSetSpec(config, images)).build();
```

CodeGPT CodeBLEU = 2

```
containerHandler.getStatefulSet(new DefaultResourceConfigBuilder().build(),
  Collections.<ImageConfiguration>emptyList());
```

Codex CodeBLEU = 19

```
Map<String, String> actualA = mergeMaps(mapA, mapB);
```

2.8 Limitations

We discuss several limitations of our work and the future work inspired by those limitations.

Usability. We envision our models being integrated into an IDE. At any point, a user would be able to see top-k results from our models and potentially decide to use one of the predictions. This is similar to email completion that has recently been integrated into several popular web-based email clients, e.g., GMail.

Structured representation. Currently we do not considering using any structured

Table 2.7: Example of test completion results by TeCo and baseline models, for `shouldIndexVeryLongDescriptionWithSingleField` in `orientechnologies/orientdb`, `LuceneSortTest.java`.

Method under test

```
default Stream<OResult> stream() {
  return StreamSupport.stream(this, false).onClose(() -> this.close());
}
```

Test method signature

```
@Test public void shouldIndexVeryLongDescriptionWithSingleField() throws Exception
```

Prior statements

```
db.save(new ODocument("STR").field("STR", DESCRIPTION));
db.command(new OCommandSQL("STR")).execute();
long count = db.query("STR").stream().count();
```

Gold

```
Assert.assertEquals(1, count);
```

TeCo CodeBLEU = 100

```
Assert.assertEquals(1, count);
```

CodeT5 CodeBLEU = 6

```
assertThat(count).isEqualTo(1);
```

CodeGPT CodeBLEU = 6

```
assertThat(count).isEqualTo(1);
```

Codex CodeBLEU = 1

```
// doc.save() ;
```

representation of code, e.g., abstract syntax trees (ASTs). Such a representation could enhance performance of our models and enable a quick check of validity of generated code. We leave this for future work.

Testing frameworks. We focus on tests written in the JUnit style, which is the most popular testing framework among Java projects. Future work can support advanced testing features in JUnit (e.g., parameterized tests) and other testing frameworks (e.g., TestNG).

Order of assertion statements. Multiple assertion statements in a test method are usually unordered, i.e., can be written and executed in any order. Our work currently

do not consider this, which means test completion requires the model to generate assertion statements in the same order as they appear in the developer-written test methods, and test oracle generation requires the model to generate only the first assertion statement even if there are many (following prior work that defined this task [55, 207]). Future work should consider alternative orders of assertion statements both during training (which should improve our model's performance) and evaluation.

2.9 Summary

We introduced an idea of designing ML models for code-related tasks with code semantics inputs and reranking based on test execution outcomes. Based on this idea, we developed a concrete model, named TeCo, targeting a novel task: test completion. We evaluated TeCo on a new corpus, containing 130,934 methods and 101,965 executable methods. Our results show that TeCo significantly outperforms the state-of-the-art on code completion and test oracle generation tasks, across a number of evaluation metrics. We believe that TeCo is only a starting point in the exciting area of ML for code with code semantics and execution data.

Chapter 3: Deep Generation of Coq Lemma Names Using Elaborated Terms

Coding conventions for naming, spacing, and other essentially stylistic properties are necessary for developers to effectively understand, review, and modify source code in large software projects. Consistent conventions in verification projects based on proof assistants, such as Coq, increase in importance as projects grow in size and scope. While conventions can be documented and enforced manually at high cost, emerging approaches automatically learn and suggest idiomatic names in Java-like languages by applying statistical language models on large code corpora. However, due to its powerful language extension facilities and fusion of type checking and computation, Coq is a challenging target for automated learning techniques. We present novel generation models for learning and suggesting lemma names for Coq projects. Our models, based on multi-input neural networks, are the first to leverage syntactic and semantic information from Coq's lexer (tokens in lemma statements), parser (syntax trees), and kernel (elaborated terms) for naming; the key insight is that learning from elaborated terms can substantially boost model performance. We implemented our models in a toolchain, dubbed Roosterize, and applied it on a large corpus of code derived from the Mathematical Components family of projects, known for its stringent coding conventions. Our results show that Roosterize substantially outperforms baselines for suggesting lemma names, highlighting the importance of using multi-input models and elaborated terms.[1]

[1]Parts of this chapter are published at IJCAR 2020 [143] and ICSE DEMO 2021 [144]. Compared to the version published at the conference, this chapter expands the evaluation to a larger corpus and adds an ablation study on the tree chopping algorithms.

3.1 Overview

Programming language source code with deficient coding conventions, such as misleading function and variable names and irregular spacing, is difficult for developers to effectively understand, review, and modify [16, 132, 192]. Code with haphazard adherence to conventions may also be more bug-prone [30]. The problem is exacerbated in large projects with many developers, where different source code files and components may have inconsistent and clashing conventions.

Many open source software projects manually document coding conventions that contributors are expected to follow, and maintainers willingly accept fixes of violations to such conventions [8]. Enforcement of conventions can be performed by static analysis tools [72, 151]. However, such tools require developers to write precise checks for conventions, which are tedious to define and often *incomplete*. To address this problem, researchers have proposed techniques for automatically learning coding conventions for Java-like languages from code corpora by applying statistical language models [11]. These models are applicable because code in these languages has high naturalness [83], i.e., statistical regularities and repetitiveness. Learned conventions can then be used to, e.g., suggest names in code.

Proof assistants, such as Coq [25], are increasingly used to formalize results in advanced mathematics [70, 71] and develop large trustworthy software systems, e.g., compilers, operating systems, file systems, and distributed systems [39, 112, 213]. Such projects typically involve contributions of many participants over several years, and require considerable effort to maintain over time. Coding conventions are essential for evolution of large verification projects, and are thus highly emphasized in the Coq libraries HoTT [87] and Iris [92], in Lean's Mathlib [17], and in particular in the influential Mathematical Components (MathComp) *family of Coq projects* [42]. Extensive changes to adhere to conventions, e.g., on naming, are regularly requested by MathComp maintainers for proposed external contributions [130], and its conventions have been adopted, to varying degrees, by a growing number of independent

Coq projects [2, 23, 57, 189].

We believe these properties make Coq code related to MathComp an attractive target for automated learning and suggesting of coding conventions, in particular, for suggesting *lemma names* [14]. However, serious challenges are posed by, on the one hand, Coq's powerful language extension facilities and fusion of type checking and computation [22], and on the other hand, the idiosyncratic conventions used by Coq practitioners compared to software engineers. Hence, although suggesting lemma names is similar in spirit to suggesting method names in Java-like languages [214], the former task is more challenging in that lemma names are typically much shorter than method names and tend to include heavily abbreviated terminology from logic and advanced mathematics; a single character can carry significant information about a lemma's meaning. For example, the MathComp lemma names `card_support_normedTI` ("cardinality of support groups of a normed trivial intersection group") and `extprod_mulgA` ("associativity of multiplication operations in external product groups") concisely convey information on lemma statement structure and meaning through both abbreviations and suffixes, as when the suffix `A` indicates an associative property.

In this paper, we present novel generation models for learning and suggesting lemma names for Coq verification projects that address these challenges. Specifically, based on our knowledge of Coq and its implementation, we developed multi-input encoder-decoder neural networks for generating names that use information directly from Coq's internal data structures related to lexing, parsing, and type checking. In the context of naming, our models are the first to leverage the *lemma statement* as well as the corresponding *syntax tree* and *elaborated term* (i.e., execution data which we call *kernel tree*) processed by Coq's kernel [51].

We implemented our models in a toolchain, dubbed Roosterize, which we used to learn from a high-quality Coq corpus derived from the MathComp family. We then measured the performance of Roosterize using automatic metrics, finding

that it significantly outperforms baselines. Using our best model, we also suggested lemma names for the PCM library [138, 189], which were manually reviewed by the project maintainer with encouraging results.

To allow ROOSTERIZE to use information directly from Coq's lexer, parser, and kernel, we extended the SerAPI library [63] to serialize Coq tokens, syntax trees, and kernel trees into a machine-readable format. This allowed us to achieve robustness against user-defined notations and other extensions to Coq syntax. Thanks to our integration with SerAPI and its use of metaprogramming, we expect our toolchain to only require modest maintenance as Coq evolves.

We make the following key contributions in this work:

- **Models**: We propose novel generation models based on multi-input neural networks to learn and suggest lemma names for Coq verification projects. A key property of our models is that they combine data from several Coq phases, including lexing, parsing, and term elaboration.
- **Corpus**: Advised by MathComp developers, we constructed a corpus of high-quality Coq code for learning coding conventions, totaling over 297k LOC taken from 21 core projects. We believe that our corpus can enable development of many novel techniques for Coq based on statistical language models.
- **Toolchain**: We implemented a toolchain, dubbed ROOSTERIZE, which suggests lemma names for a given Coq project. We envision ROOSTERIZE being useful during the review process of proposed contributions to a Coq project.
- **Evaluation**: We performed several experiments with ROOSTERIZE to evaluate our models using our corpus. Our results show that ROOSTERIZE performs significantly better than several strong baselines, as measured by standard automatic metrics [160]. The results also reveal that our novel multi-input models, as well as the incorporation of kernel trees, are important for prediction quality. Finally, we performed a manual quality analysis by suggesting lemma names for a medium sized

Coq project [138], evaluated by its maintainer, who found many of the predictions useful for improving naming consistency.

We provide artifacts related to our toolchain and corpus at:
`https://github.com/EngineeringSoftware/roosterize`.

3.2 Background

This section gives brief background related to Coq and the Mathematical Components (MathComp) family of projects, as well as the SerAPI library.

Coq and Gallina. Coq is a proof assistant based on dependent types, implemented in the OCaml language [25, 45]. For our purposes, we view Coq as a programming language and type-checking toolchain. Specifically, Coq *files* are sequences of *sentences*, with each sentence ending with a period. Sentences are essentially either (a) commands for printing and other output, (b) definitions of functions, lemmas, and datatypes in the Gallina language [47], or (c) expressions in the Ltac tactic language [53]. We will refer to definitions of lemmas as in (b) as *lemma sentences*. Coq internally represents a lemma sentence both as a sequence of tokens (lexing phase) and as a syntax tree (parsing phase).

In the typical workflow for a Coq-based verification project, users write datatypes and functions and then interactively prove lemmas about them by executing different tactic expressions that may, e.g., discharge or split the current proof goal. Both statements to be proved and proofs are represented internally as *terms* produced during an *elaboration* phase [51]; we refer to elaborated terms as *kernel trees*. Hence, as tactics are successfully executed, they gradually build a kernel tree. The `Qed` command sends the kernel tree for a tentative proof to Coq's kernel for final certification. We call a collection of Ltac tactic sentences that build a kernel tree a *proof script*.

Figure 3.1 shows a Coq lemma and its proof script, taken verbatim from a development on the theory of regular languages [57]. Line 1 contains a lemma sentence

```
1 Lemma mg_eq_proof L1 L2 (N1 : mgClassifier L1) : L1 =i L2 -> nerode L2 N1.
2 Proof. move => H0 u v. split => [/nerodeP H1 w|H1].
3   - by rewrite -!H0.
4   - apply/nerodeP => w. by rewrite !H0.
5 Qed.
```

Figure 3.1: Coq lemma on the theory of regular languages, including proof script.

with the lemma name `mg_eq_proof`, followed by a *lemma statement* (on the same line) involving the arbitrary languages `L1` and `L2`, i.e., typed variables that are implicitly universally quantified. When Coq processes line 5, the kernel certifies that the kernel tree generated by the proof script (lines 2 to 4) has the type (is a proof) of the kernel tree for the lemma statement on line 1.

MathComp and lemma naming. The MathComp family of Coq projects, including in particular the MathComp library of general mathematical definitions and results [128], grew out of Gonthier's proof of the four-color theorem [70], with substantial developments in the context of the landmark proof of the odd order theorem in abstract algebra [71]. The MathComp library is now used in many projects outside of the MathComp family, such as in the project containing the lemma in Figure 3.1 [58]. MathComp has documented naming conventions for two kinds of entities: (1) variables and (2) functions and lemmas [42]. Variable names tend to be short and simple, while function and lemma names can be long and consist of several *name components*, typically separated by an underscore, but sometimes using CamelCase. Examples of definition and lemma names in Figure 3.1 include `mg_eq_proof`, `mgClassifier`, `nerode`, and `nerodeP`. Note that lemma names sometimes have *suffixes* to indicate their meaning, such as `P` in `nerodeP` which says that the lemma is a *characteristic property*. Coq functions tend to be named based on corresponding function definition bodies rather than just types (of the parameters and return value), analogously to methods in Java [122]. In contrast, MathComp lemma names tend to be based solely on the lemma statement. Hence, a more suitable name for the lemma in Figure 3.1 is `mg_eq_nerode`.

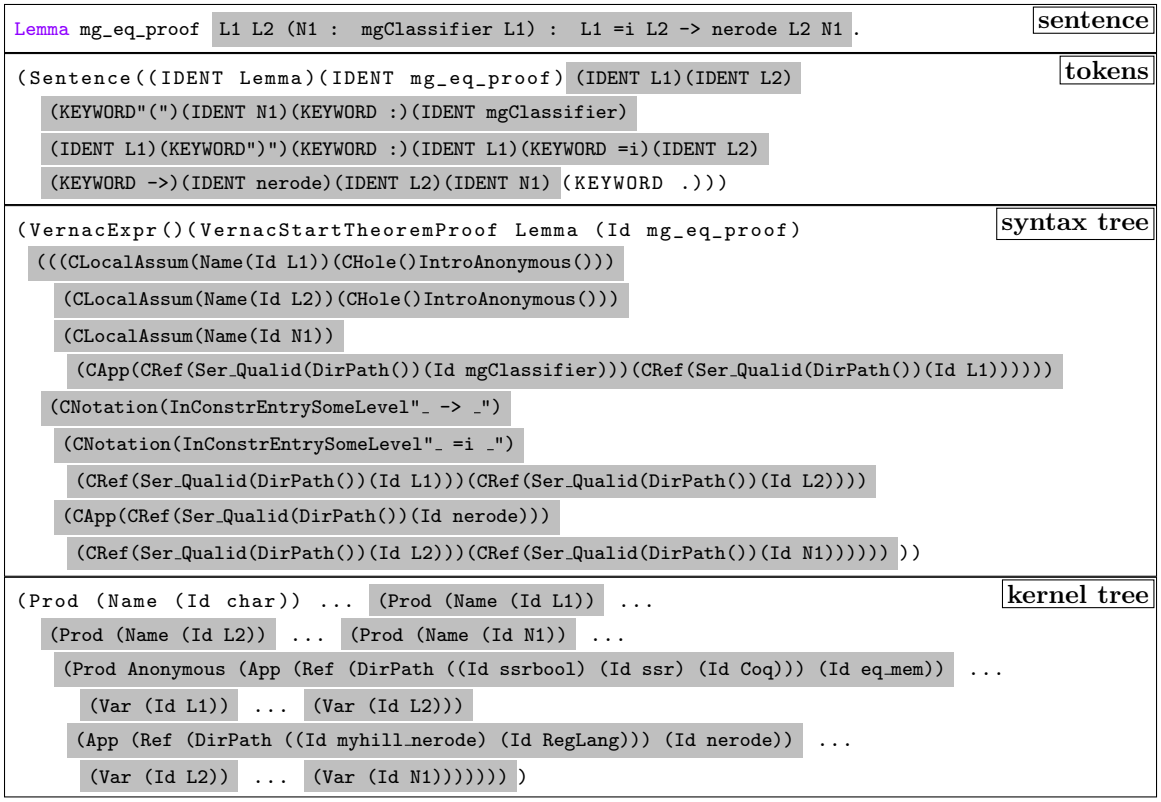


Figure 3.2: Coq lemma sentence at the top, with sexps for, from just below to bottom: tokens, syntax tree, and kernel tree; the lemma statement in each is highlighted.

SerAPI and Coq serialization. SerAPI is an OCaml library and toolchain for machine interaction with Coq [63], which provides serialization and deserialization of Coq internal data structures to and from S-expressions (sexps) [131]. SerAPI is implemented using OCaml's PPX metaprogramming facilities [150], which enable modifying OCaml program syntax trees at compilation time. Figure 3.2 shows the lemma sentence on line 1 in Figure 3.1, and below it, the corresponding (simplified) sexps for its tokens, syntax tree, and kernel tree, with the lemma statement highlighted in each representation. Note that the syntax tree omits the types of some quantified variables, e.g., for the types of `L1` and `L2`, as indicated by the `CHole` constructor. Note also that during elaboration of the syntax tree into the kernel tree by Coq, an implicit variable `char` is added (all-quantified via `Prod`), and the extensional equality operator `=i` is

translated to its globally unique *kernel name*, `Coq.ssr.ssrbool.eq_mem`. Hence, a kernel tree can be much larger and contain more information than the corresponding syntax tree.

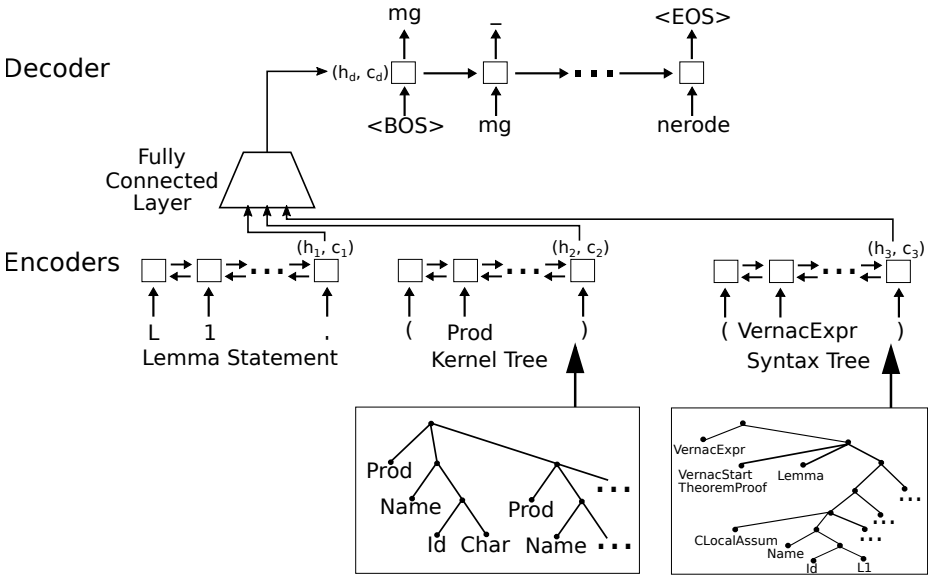
3.3 Models

In this section, we describe our multi-input generation models for suggesting Coq lemma names. Our models consider lemma name generation with an *encoder-decoder* mindset, i.e., we use neural architectures specifically designed for transduction tasks [193]. This family of architectures is commonly used for sequence generation, e.g., in machine translation [20] and code summarization [109], where it has been found to be much more effective than traditional probabilistic and retrieval-based approaches.

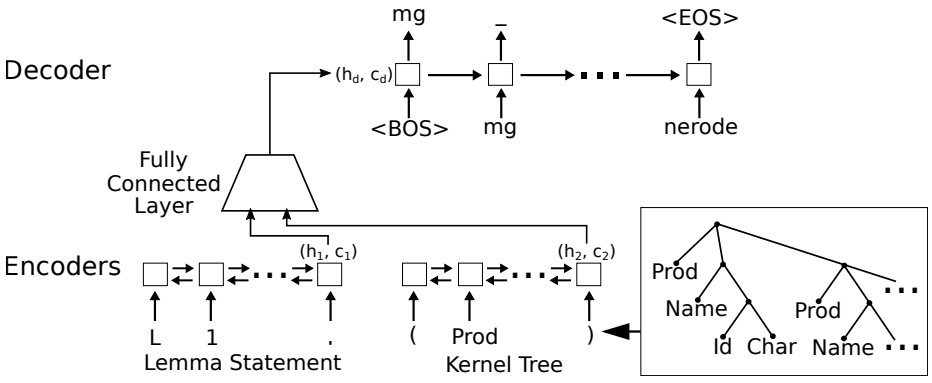
3.3.1 Core Architecture

Our encoders are Recurrent Neural Networks (RNNs) that learn a deep semantic representation of a given lemma statement from its tokens, syntax tree, and kernel tree. The decoder—another RNN—generates the descriptive lemma name as a sequence. The model is trained end-to-end, maximizing the probability of the generated lemma name given the input. In contrast to prior work in language-code tasks that uses a single encoder [64], we design multi-input models that leverage both syntactic and semantic information from Coq's lexer, parser, and kernel. A high-level visualization of our architecture is shown in Figure 3.3. The number of inputs to use can be configured, for example, Figure 3.3a shows an architecture that uses all three inputs, and Figure 3.3b shows an architecture that uses two inputs (lemma statement and kernel tree).

Encoding. Our multi-input encoders combine different kinds of syntactic and semantic information in the encoding phase. We use a different encoder for each input, which are: lemma statement, syntax tree, and kernel tree.



(a) With three inputs: lemma statement, kernel tree, and syntax tree.



(b) With two inputs: lemma statement and kernel tree.

Figure 3.3: Core architecture of ROOSTERIZE's multi-input encoder-decoder models.

Coq data structure instances can be large, with syntax trees having an average depth of 28.03 and kernel trees 46.51 in our corpus (we provide detailed statistics in Section 3.6). Therefore, we flatten the trees into sequences, which can be trained more efficiently than tree encoders without performance loss [88]. We flatten the trees with pre-order traversal, and we use "(" and ")" as the boundaries of the children of a node. In later parts of this paper, we use syntax and kernel trees to refer to their flattened versions. In Section 3.3.2, we introduce *tree chopping* to reduce the length of the resulting sequences.

To encode lemma statements and flattened tree sequences, we use bi-directional Long-Short Term Memory (LSTM) [85] networks. LSTM is a type of RNN good at capturing long-range dependencies in a sequence, and is widely used in encoders [88]. A bi-directional LSTM learns stronger representations (than a uni-directional LSTM) by encoding a sequence from both left to right and right to left [224].

Decoding. We use an LSTM (left to right direction only) as our decoder. To obtain the initial hidden and cell states (h_d, c_d) of the decoder, we learn a unified representation of these separate encoders by concatenating their final hidden and cell states (h_i, c_i), and then applying a fully connected layer on the concatenated states: $h_d = W_h \cdot \texttt{concat}([h_i]) + b_h$ and $c_d = W_c \cdot \texttt{concat}([c_i]) + b_c$, where W_h, W_c, b_h, and b_c are learnable parameters.

During training, we maximize the log likelihood of the reference lemma name given all input sequences. Standard beam search is used to reduce the search space for the optimal sequence of tokens. With regular decoding, at each time step the decoder generates a new token relying on the preceding *generated* token, which can be error-prone and leads to slow convergence and instability. We mitigate this problem by performing decoding with teacher forcing [212] such that the decoder relies on the preceding *reference* token. At test time, the decoder still uses the proceeding generated token as input.

Attention. With RNN encoders, the input sequence is compressed into the RNN's

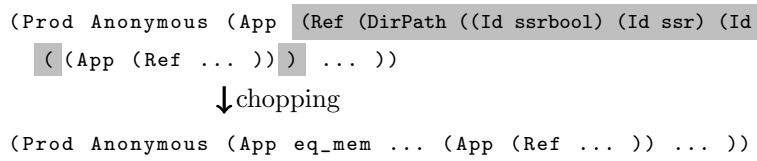


Figure 3.4: Kernel tree sexp before and after chopping; chopped parts are highlighted.

final hidden states, which results in a loss of information, especially for longer sequences. The attention mechanism [127] grants the decoder access to the encoder hidden and cell states for all previous tokens. At each decoder time step, an attention vector is calculated as a distribution over all encoded tokens, indicating which token the decoder should "pay attention to". To make the attention mechanism work with multiple encoders, we concatenate the hidden states of the n encoders $[h_1, ..., h_n]$ and apply an attention layer on the result [199].

Initialization. Since there are no pre-trained token embeddings for Coq, we initialize each unique token in the vocabulary with a random vector sampled from the uniform distribution $U(-0.1, 0.1)$. These embeddings are trained together with the model. The hidden layer parameters of the encoders and decoders are also initialized with random vectors sampled from the same uniform distribution.

3.3.2 Tree Chopping

While syntax and kernel trees for lemma statements can be large, not all parts of the trees are relevant for naming. For instance, each constant reference is expanded to its fully qualified form in the kernel tree, but the added prefixes are usually related to directory paths and likely do not contain relevant information for generating the name. Irrelevant information in long sequences can be detrimental to the model, since the model would have to reason about and encode all tokens in the sequence.

To this end, we implemented *chopping* heuristics for both syntax trees and kernel trees to remove irrelevant parts. The heuristics essentially: (1) replace the

fully qualified name sub-trees with only the last component of the name; (2) remove the location information from sub-trees; (3) extract the singletons, i.e., non-leaf nodes that have only one child. Figure 3.4 illustrates the chopping of a kernel tree, with the upper box showing the tree before chopping with the parts to be removed highlighted, and the lower box showing the tree after chopping; in this example, we chopped a fully qualified name and extracted a singleton. These heuristics greatly reduce the size of the tree: for kernel trees, they reduce the average depth from 39.20 to 11.39.

Our models use chopped trees as the inputs to the encoders. As we discuss in more detail in Section 3.7, the chopped trees help the models to focus better on the relevant parts of the inputs. While the attention mechanism in principle could learn what the relevant parts of the trees are, our evaluation shows that it can easily be overwhelmed by large amounts of irrelevant information.

3.3.3 Copy Mechanism

We found it common for lemma name tokens to only occur in a single Coq file, whence they are unlikely to appear in the vocabulary learned from the training set, but can still appear in the respective lemma statement, syntax tree, or kernel tree. For example, `mg` occurs in both the lemma name and lemma statement in Figure 3.1, but not outside the file the lemma is in. To account for this, we adopt the copy mechanism [186] which improves the generalizability of our model by allowing the decoder to *copy* from inputs rather than always choosing one word from the fixed vocabulary from the training set. To handle multiple encoders, similar to what we did with the attention layer, we concatenate the hidden states of each encoder and apply a copy layer on the concatenated hidden states.

3.3.4 Subtokenization

We subtokenize all inputs (lemma statements, syntax and kernel trees) and outputs (lemma names) in a pre-processing step. Previous work on learning from

software projects has shown that subtokenization helps to reduce the sparsity of the vocabulary and improves the performance of the model [18]. However, unlike Java-like languages where the method names (almost) always follow the CamelCase convention, lemma names in Coq use a mix of snake_case, CamelCase, prefixes, and suffixes, thus making subtokenization more complex. For example, `extprod_mulgA` should be subtokenized to `extprod`, _, `mul`, `g`, and `A`.

To perform subtokenization, we implemented a set of heuristics based on the conventions outlined by MathComp developers [42]. After subtokenization, the vocabulary size of lemma names in our corpus was reduced from 8,861 to 2,328. When applying the subtokenizer on the lemma statements and syntax and kernel trees, we subtokenize the identifiers and not the keywords or operators.

3.3.5 Repetition Prevention

We observed that decoders often generated repeated tokens, e.g., `mem_mem_mem`. This issue is common in encoder-decoder models and also exists in natural language summarization [194]; it is largely because the attention mechanism (while helping the model for the most part) does not store information on how much information the model has "covered" in the encoded sequence. We further observed that it is very unlikely to have repeated subtokens in lemma names used by proof engineers (only 1.37% of cases in our corpus). Hence, we simply forbid the decoder from repeating a subtoken (modulo "_") during beam search.

3.4 Implementation

In this section, we briefly describe our toolchain which implements the models in Section 3.3 and processes and learns from Coq files; we dub this toolchain ROOSTERIZE. The components of the toolchain can be divided into two categories: (1) components that interact with Coq or directly process information extracted from Coq, and (2) components concerned with machine learning and name generation.

The first category includes several OCaml-based tools integrated with SerAPI [63] (and thus Coq itself), and Python-based tools for processing of data obtained via SerAPI from Coq. All OCaml tools have either already been included in, or accepted for inclusion into, SerAPI itself. The tools are as follows:

`sercomp`. We integrated the existing program `sercomp` distributed with SerAPI into ROOSTERIZE to serialize Coq files to lists of sexps for syntax trees.

`sertok`. We developed an OCaml program dubbed `sertok` on top of SerAPI. The program takes a Coq file as input and produces sexps of all tokens found by Coq's lexer in the file, organized at the sentence level.

`sername`. We developed an OCaml program dubbed `sername` on top of SerAPI. The program takes a list of fully qualified (kernel) lemma names and produces sexps for the kernel trees of the corresponding lemma statements.

`postproc` & `subtokenizer`. We created two small independent tools in Python to post-process Coq sexps and perform subtokenization, respectively.

For the second category, we implemented our machine learning models in Python using two widely-used deep learning libraries: PyTorch [161] and OpenNMT [103]. More specifically, we extended the sequence-to-sequence models in OpenNMT to use multi-input encoders, and extended attention and copy layers to use multiple inputs.

3.5 Usage

This section describes how to install and use ROOSTERIZE. We provide both a command-line interface that is suitable for CI integration and batch mode and a Visual Studio Code extension that is suitable for interactive mode.

3.5.1 Installation

ROOSTERIZE currently supports macOS and Linux-based operating systems. The first installation step is to download the ROOSTERIZE repository:

```
$ git clone https://github.com/EngineeringSoftware/roosterize
$ cd roosterize && git checkout v1.1.0+8.10.2
```

Required software and libraries. ROOSTERIZE depends on two sets of software and libraries: (1) OCaml, Coq, and SerAPI; (2) PyTorch and other Python libraries.

To install OCaml (4.07.1), Coq (8.10.2) and SerAPI (0.7.1), we recommend using the OCaml-based package-management system OPAM [152] version 2.0.7 or later:

```
$ opam switch create roosterize 4.07.1
$ opam switch roosterize && eval $(opam env)
$ opam update
$ opam pin add coq 8.10.2
$ opam pin add coq-serapi 8.10.0+0.7.1
```

To install PyTorch and other Python libraries, we recommend using the package-management system Conda [12]. The installation script may be different depending on the operating system and whether to use GPU or not. For example, on Linux, to use CPU only:

```
$ conda env create --name roosterize --file conda-envs/cpu.yml
$ conda activate roosterize
```

After installing these required software and libraries, users can use ROOSTERIZE via its command-line interface.

3.5.2 Command-Line Interface

After installation, users can launch ROOSTERIZE via `./bin/roosterize` (in short, `roosterize`). We focus on the main usage of ROOSTERIZE—suggesting lemma names for Coq projects. For other usages, e.g., training models, users can refer to the help included in ROOSTERIZE:

```
(roosterize) pynie@pynie-ThinkPad-T470:~/roosterize-demo/fcsl-pcm$ ~/projects/roosterize/bin/roosterize \
suggest_naming --file=$PWD/finmap/finmap.v
== Analyzed 110 lemma names, 8 (7.3%) conform to the learned naming conventions.
==========
== 21 can be improved and here are Roosterize's suggestions:
Line 851: fcatsK => eq_fcat (likelihood: 0.45)
Line 822: fcatC => eq_fcat (likelihood: 0.44)
Line 862: fcatKs => eq_fcat (likelihood: 0.43)
Line 1178: zip_supp' => eq_zip (likelihood: 0.31)
Line 1118: map_key_zip' => eq_zip (likelihood: 0.31)
Line 1258: zunit_fcat => zip_fcat (likelihood: 0.30)
Line 769: disjC => eq_disj (likelihood: 0.30)
Line 962: mapf_disj => eq_map (likelihood: 0.29)
Line 526: fcats0 => fcat_nil (likelihood: 0.28)
Line 1273: zunit_disj => disj_zip (likelihood: 0.27)
Line 1186: zip_supp => eq_zip (likelihood: 0.27)
Line 937: mapf_ins => map_ins (likelihood: 0.26)
Line 525: fcat0s => fcat_nil (likelihood: 0.25)
Line 443: seqof_ins => path_ordP (likelihood: 0.24)
```

Figure 3.5: Screenshot of using ROOSTERIZE from command line. The predictions are ordered by likelihood (i.e., a 0–1 score indicating how confidence the model is), so that more important predictions are shown first.

```
$ roosterize --help
```

Users should first obtain a model, e.g., by downloading a pre-trained model. The following command downloads the model we pre-trained:

```
$ roosterize download_global_model
```

Applying ROOSTERIZE to a Coq project requires (1) a `_CoqProject` file in the project root directory in the format used by the `coq_makefile` tool [46], and (2) that the project source code has been compiled. If a user specified the compilation command in the `.roosterizerc` configuration file at the root directory of the project, ROOSTERIZE will automatically compile the project before suggesting lemma names. ROOSTERIZE can suggest lemma names for one Coq file at a time. For example, running the following commands downloads the Coq project the PCM library at Git revision eef4503, prepares a `.roosterizerc` configuration file, and suggests lemma names for the `finmap.v` file in the project:

```
$ git clone https://github.com/imdea-software/fcsl-pcm
$ git checkout eef4503
$ echo "compile_cmd: make -j8" > ./.roosterizerc
$ roosterize suggest_naming --file=$PWD/finmap/finmap.v
```

In the last command, ROOSTERIZE uses SerAPI to parse `finmap.v` and extracts all lemmas, then uses the lemma names prediction model to generate top k (default $k = 5$) likely lemma names for each lemma, and then compares the generated lemma names with the original lemma names, and finally prints a report to suggest potential lemma names changes. Figure 3.5 shows part of the report generated for `finmap.v`[2].

3.5.3 Visual Studio Code Extension

The ROOSTERIZE Visual Studio Code extension can be installed easily from Visual Studio Code marketplace: launch "VS Code Quick Open" (Ctrl+P), paste the following command:

```
ext install EngineeringSoftware.roosterize-vscode
```

Then, users should configure the path to ROOSTERIZE executable (./`bin`/-`roosterize`) using the following steps: open "Settings" (Ctrl+,), search for the entry "ROOSTERIZE: Bin Path", and fill in the path to ROOSTERIZE executable file.

Users should first open the Coq files they want to analyze. The steps to obtain the lemma names predictions for them are: (1) open "Command Palettes" (Ctrl+Shift+P), and (2) choose "Roosterize: Suggest Naming (for all .v files)". After ROOSTERIZE produces the predictions, the lemma names that do not conform to the conventions are underlined, and users can hover the mouse pointer over that underlined name to view ROOSTERIZE's prediction in a tooltip. Users can also view all predictions in the "Problems" tab. Figure 3.6 shows a screenshot of this step.

3.6 Corpus

We constructed a corpus of 21 large Coq projects from the MathComp family, totaling 297k lines of code (LOC). We selected these projects based on the recom-

[2]Full report available at: `https://github.com/EngineeringSoftware/roosterize/blob/v1.1.0%2B8.10.2-beta/docs/example-suggestion.txt`

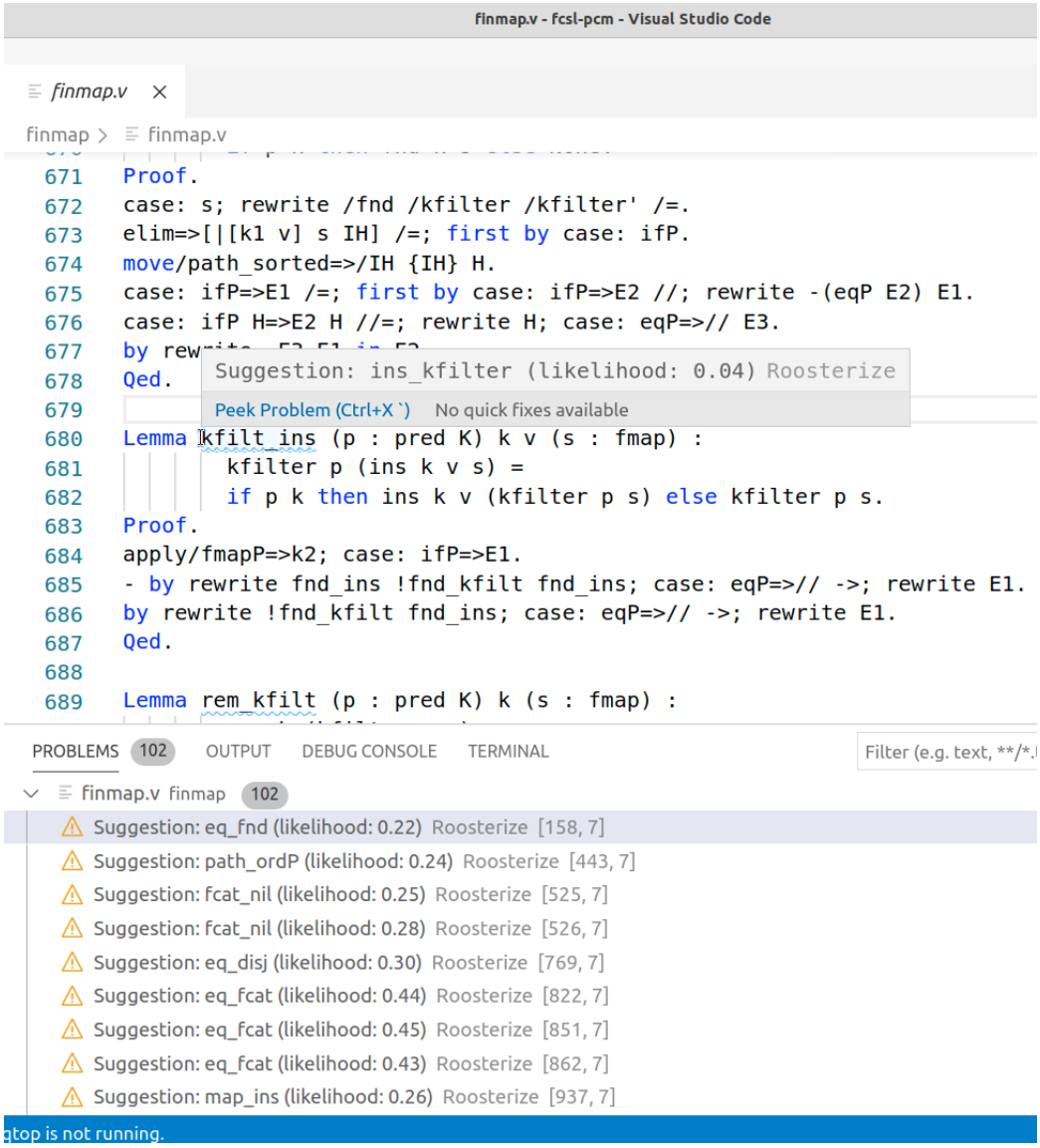


Figure 3.6: Screenshot of using ROOSTERIZE from Visual Studio Code.

mendation of MathComp developers, who emphasized their high quality and stringent adherence to coding conventions. Our corpus is *self-contained*: there are inter-project dependencies within the corpus, but no project depends on a project outside the corpus (except Coq's standard library). All projects build with Coq version 8.10.2. Note that we need to be able to build projects to be able to extract tokens, syntax trees, and kernel trees.

3.6.1 Constituent Projects

Table 3.1 lists the projects in the corpus, along with basic information about each project. The table includes columns for the project identifier, revision SHA, number of files (#Files), number of lemmas (#Lemmas), number of tokens (#Toks), LOC for specifications (Spec.) and proof scripts (Proof), and average LOC per file for specifications and proof scripts. The math-comp SHA corresponds to version 1.9.0 of the library. The LOC numbers are computed with Coq's bundled `coqwc` tool. Our corpus consists of two parts: the main part consists of 20 projects and is used for training and evaluating ROOSTERIZE; the left-out (LO) part is one project, infotheo, which is used to study the generalizability of ROOSTERIZE on an unseen project.

We constructed and organized the corpus based on recommendations from MathComp developers. The 4 core MathComp projects used in the original corpus are included as the *tier 1* set. We selected 9 projects for the *tier 2* set, such that each included project (1) has a main contributor who is also a significant contributor to one of the tier 1 projects, and (2) follows to a significant degree the coding conventions specified for MathComp. (infotheo would be in this set had we not added it to the left-out part.) Finally, we selected 8 projects which follow MathComp coding conventions but do not fulfill the tier 2 criteria, for inclusion in the *tier 3* set.

We briefly describe each project in our corpus:

analysis. A library for general real analysis.

bigenough. A small library for $\epsilon - N$ reasoning.

Table 3.1: Projects from the MathComp family used in our corpus.

	Project		SHA	#Files	#Lemmas	#Toks	LOC		LOC/file	
							Spec.	Proof	Spec.	Proof
Tier 1	finmap		27642a8	4	940	78,449	4,260	2,191	1,065.00	547.75
	fourcolor		0851d49	60	1,157	560,682	9,175	27,963	152.92	466.05
	math-comp		748d716	89	8,802	1,076,096	38,243	46,470	429.70	522.13
	odd-order		ca602a4	34	367	519,855	11,882	24,243	349.47	713.03
Tier 2	analysis		9e5fe1d	17	969	152,542	5,553	6,186	326.65	363.88
	bigenough		5f79a32	1	4	731	70	8	70.00	8.00
	elliptic-curves		631af89	18	625	110,480	3,298	6,298	183.22	349.89
	grobner		dfa54f9	1	81	15,656	312	1,018	312.00	1,018.00
	multinomials		691d795	5	831	83,438	3,699	3,664	739.80	732.80
	real-closed		495a1fa	10	561	108,925	4,348	4,577	434.80	457.70
	robot		b341ad1	13	864	130,024	3,881	7,249	298.54	557.62
	two-square		1c09aca	2	200	20,326	413	1,308	206.50	654.00
Tier 3	bits		3cd298a	10	411	40,420	1,578	2,463	157.80	246.30
	comp-dec-pdl		c1f813b	16	494	61,731	2,305	2,114	144.06	132.12
	disel		e8aa80c	20	256	51,473	2,575	1,898	128.75	94.90
	fcsl-pcm		eef4503	12	690	70,273	2,937	2,852	244.75	237.67
	games		3d3bd31	12	231	43,438	1,450	3,503	120.83	291.92
	monae		9d0e461	18	349	73,578	3,422	3,233	190.11	179.61
	reglang		da333e0	12	230	41,327	1,299	1,734	108.25	144.50
	toychain		97bd697	14	67	61,997	1,747	3,528	124.79	252.00
	Main	**Avg.**	N/A	18.40	906.45	165,072.05	5,122.35	7,625.00	278.39	414.40
		Σ	N/A	368	18,129	3,301,441	102,447	152,500	102,447	152,500
LO	infotheo		6c17242	81	1,891	463,593	12,517	29,778	154.53	367.63
	All	**Avg.**	N/A	21.38	953.33	179,287.33	5,474.48	8,679.90	256.04	405.96
		Σ	N/A	449	20,020	3,765,034	114,964	182,278	114,964	182,278

bits. A library for reasoning about bit-level operations [29].

comp-dec-pdl. Formal proofs of completeness and decidability of converse propositional dynamic logic [56].

disel. A framework for distributed separation logic, useful for verifying implementations of distributed systems [190].

elliptic-curves. A formalization of the algebraic theory of elliptic curves [23].

fcsl-pcm. A library formalizing partial commutative monoids, which are useful for reasoning about pointer-based programs [189].

finmap. A library with definitions and results about finite maps and sets with finitely many members.

fourcolor. An updated version of the formal proof of the four-color theorem in graph theory [70], which states that in all planar graphs, four colors suffice for coloring all vertices such that no two adjacent vertices have the same color.

games. Definitions and formal proofs of theorems in algorithmic game theory [19].

grobner. A formalization of Gröbner bases.

math-comp. The MathComp library itself [128].

monae. A library for monadic equational reasoning [3].

multinomials. A library formalizing monoidal rings and multinomials, and related results.

odd-order. The formal proof of the odd order (Feit-Thompson) theorem in abstract algebra [71], which states that all groups of odd order are solvable.

real-closed. Theorems on real closed fields in algebra.

reglang. A formalization of the theory of regular languages [57].

robot. A formalization of the mathematics of rigid body transformations to enable proofs about robot manipulators [1].

Table 3.2: Statistics of the lemmas extracted from our corpus, divided into training, validation, and evaluation sets.

		#Lemmas	Name		Stmt	
			#Char	#SubToks	#Char	#SubToks
All Tiers	before filtering	23,615	10.57	4.32	59.21	25.93
	after filtering	18,129	9.89	4.13	47.48	21.16
	training	15,011	9.99	4.12	47.93	21.20
	validation	1,556	9.20	4.08	41.44	18.65
	evaluation	1,562	9.68	4.26	49.19	23.21
Tier 1	training	8,861	10.14	4.22	44.16	19.59
	validation	1,085	9.20	4.20	38.28	17.30
	evaluation	1,320	9.76	4.34	48.49	23.20
Tier 2	training	3,692	10.03	4.02	50.52	22.37
	validation	403	9.27	3.86	46.26	20.92
	evaluation	40	8.75	3.77	49.25	22.32
Tier 3	training	2,458	9.37	3.90	57.65	25.28
	validation	68	8.74	3.44	63.34	26.82
	evaluation	202	9.35	3.82	53.70	23.43

toychain. Formalization and verification of a blockchain network protocol [167].

two-square. A proof of Fermat's theorem on the sum of two squares, including a definition of Gaussian integers.

infotheo. Formalizations of notions and results from information theory and probability theory [2].

3.6.2 Corpus Statistics

We extracted all lemmas from the corpus, and initially we obtained 23,615 lemmas in total. However, we found several outlier lemmas where the lemma statement, syntax tree and kernel tree were very large. To ensure stable training, and similar to prior work on generating method names for Java [122], we excluded the lemmas with the deepest 25% kernel trees. This left us with 18,129 lemmas. Column 4 of Table 3.1 shows the number of lemmas after filtering.

We randomly split corpus files into training, validation, and evaluation sets

which contain 80%, 10%, 10% of the files, respectively. Table 3.2 shows statistics on the lemmas in each set (of all tiers, tier 1, tier 2, and tier 3, respectively), which includes columns for the number of files, the number of lemmas, the average number of characters and subtokens in lemma names, and the average number of characters and subtokens in lemma statements.

Figure 3.7 shows the boxplots to illustrate the changes of the depth, number of nodes and number of subtokens (after flattening) of the kernel trees (first row) and syntax trees (second row), before filtering, after filtering (before chopping), and after chopping, respectively. Figure 3.7a shows the boxplots on the entire corpus, and Figure 3.7b shows the boxplots on the tier 1 corpus. Our chopping process reduced tree depth by 71.1% for kernel trees and 70.7% for syntax trees, and reduced the number of nodes by 91.5% for kernel trees and 91.0% for syntax trees; after flattening, the resulting average sequence length is, for kernel trees 157 comparing to the original 2,003, and for syntax trees 149 comparing to the original 1,677.

3.7 Experiments Setup

We evaluate ROOSTERIZE by answering the following research questions, using a combination of automatic evaluation and manual quality assessment:

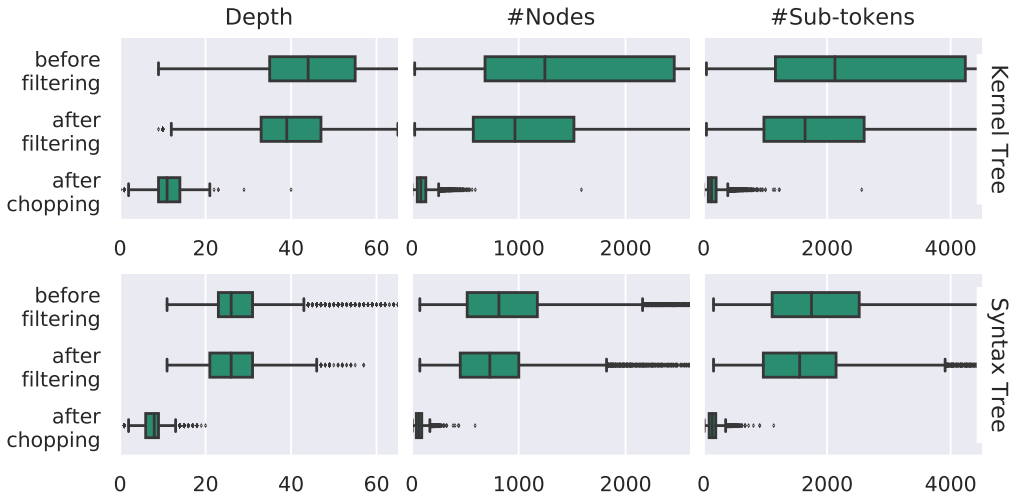
RQ1: What is the performance of ROOSTERIZE on the lemma naming task and how does it compare to baselines, when training and evaluating on the tier 1 corpus?

RQ2: How do different chopping algorithms affect the performance of ROOSTERIZE?

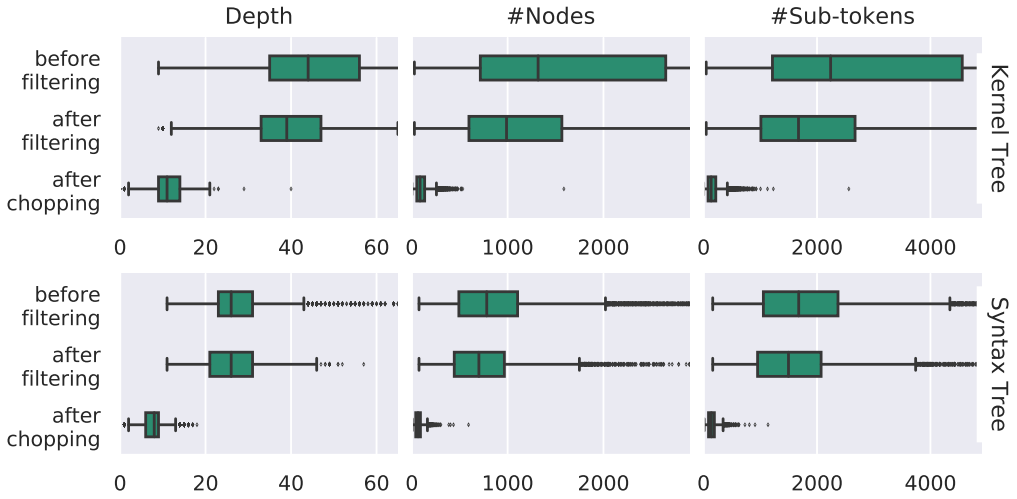
RQ3: What is the performance of ROOSTERIZE, when training and evaluating across different tiers of our corpus?

RQ4: What is the performance of ROOSTERIZE on a project unseen during training, and how does it change with additional training on the project?

RQ5: How useful are the names generated by ROOSTERIZE, as manually assessed by the maintainer of a project?



(a) On the entire corpus.



(b) On the tier 1 corpus.

Figure 3.7: Boxplots of statistics of syntax and kernel trees before filtering, after filtering (before chopping), and after chopping.

3.7.1 Models and Baselines

We study the combinations of: (1) using individual input (lemma statement and trees) in a single encoder, or multi-input encoders with different mixture of these inputs; and (2) using the attention and copy mechanisms. Our inputs include: lemma statement (*Stmt*), syntax tree (*STree*), chopped syntax tree (*CSTree*), kernel tree (*KTree*), and chopped kernel tree (*CKTree*). For multiple inputs, the models are named by concatenating inputs with "+"; a "+" is also used to denote the presence of attention (*attn*) or copy (*copy*). For example, Stmt+CKTree+attn+copy refers to a model that uses two encoders—one for lemma statement and one for chopped kernel tree—and uses attention and copy mechanisms.

We consider the vanilla encoder-decoder models with only one input (lemma statement, syntax tree, or kernel tree) as baseline models. We also compare with a retrieval-based baseline model implemented using Lucene [13]: a k-nearest neighbors classifier using the tf-idf[3] of the tokens in lemma statement as features.

Hyper-parameters are tuned on the validation set within the following options: embedding dimensions from {200, 500, 1000}, number of hidden units in each LSTM from {200, 500, 1000}, number of stacked LSTM layers from {1, 2, 3}. We set the dropout rate[4] between LSTM layers to 0.5. We set the output dimension of the fully connected layer for combining encoders to the same number as the number of hidden units in each LSTM. We checked the validation loss every 200 training steps (as defined in OpenNMT [103], which is similar to one training epoch on our dataset),

[3]Tf-idf is a numerical metric reflecting the importance of a token to a document in a corpus, calculated as the product of term frequency (proportional to the frequency of the token in the document) and inverse document frequency (inversely proportional to the number of documents containing the token). In our retrieval-based baseline model, we used Lucene's implementation of tf-idf [13].

[4]Dropout [84] is a regularization technique for reducing overfitting, by randomly resetting a fraction of neural connections between two layers during training (and during training only). In our experiments, a dropout rate of 0.5 between the LSTM layers means that 50% of the bits of the hidden and cell states are set to 0 when they are passed from a previous layer to its next layer in the LSTM during training.

and set an early stopping threshold of 3. We used the Adam [102] optimizer with a learning rate[5] of 0.001. We used a beam size of 5 in beam search. All the experiments were run on a cluster of machines each equiped with 2x Intel(R) Xeon(R) E5-2620 v4 @ 2.10GHz CPUs, 4x NVidia 1080-TI GPUs, and 128GB RAM.

3.7.2 Metrics

We use four automatic metrics which evaluate generated lemma names against the reference lemma names (as written by developers) in the evaluation set. Each metric captures a different level of granularity of the generation quality.

BLEU [160] is a standard metric used in transduction tasks including language $\leftrightarrow$ code transduction. It calculates the number of n-grams in a generated sequence that also appear in the reference sequence, where one "n-gram" is n consecutive items in a sequence (in our case, one "n-gram" is n consecutive characters in the sequence of characters of the lemma name). We use it to compute the $1\sim4$-grams overlap between the characters in generated name and characters in the reference name, averaged between $1\sim4$-grams with smoothing method proposed by Lin and Och [119].

Fragment accuracy (FragAcc) computes the accuracy of generated names on the fragment level, which is defined by splitting the name by underscores ("_"). For example, when the reference name is `det_map_mx` and the generated name is `map_determinant_mx`, the fragment accuracy is 66.7%. Unlike BLEU, fragment accuracy ignores the ordering of the fragments.

Exact-match accuracy (XM) computes how often the true name fully matches the generated name.

[5]The learning rate controls the speed of adjusting models' learnable parameters based on the loss at each iteration of the training. An excessively large learning rate makes training faster, but may result in "overshooting": adjusting so much that it results in jumping over the minima. A too low learning rate means training is unnecessarily slow to complete, and may result in the training getting stuck in a local minimum.

Table 3.3: The combinations of training, validation, and evaluation sets used in our evaluation.

Training & Validation	Evaluation	Results Table
Tier 1	Tier 1	Table 3.4
All Tiers	All Tiers	Table 3.6
All Tiers	Tier 1	Table 3.7
All Tiers	Tier 2	Table 3.8
All Tiers	Tier 3	Table 3.9
Tier 1	All Tiers	Table 3.10
Tier 1	Tier 2	Table 3.11
Tier 1	Tier 3	Table 3.12
Tier 2	Tier 2	Table 3.13
Tier 3	Tier 3	Table 3.14

Top-5 accuracy (Acc@5) computes how often the true name is one of the top-5 generated names.

3.7.3 Training, Validation, and Evaluation Sets

Different tiers of our corpus have different levels of alignment to MathComp's coding conventions, as we explained in Section 3.6. We primarily compare Roosterize against baselines and draw conclusions from the results by training the models on tier 1's training & validation set and evaluating on tier 1's evaluation set, because the tier 1 corpus is the least noisy (i.e., most developer-written lemma names are correct). We also experiment on other combinations of training, validation, and evaluation set to study: (1) whether models trained on the all tiers corpus, which is larger but more noisy, can achieve better performance; (2) whether models trained on the tier 2 / tier 3 corpus, which is less coherent than the tier 1 corpus, can perform well on the tier 2 / tier 3 corpus itself. Table 3.3 lists the combinations we used; the first column shows the corpus that training and validation sets are from, the second column shows the corpus that evaluation set is from, and the third column indicates the results table for each combination.

3.8 Results

This section presents the results of ROOSTERIZE and baseline models, and answers the research questions from Section 3.7.

3.8.1 RQ1: Roosterize vs. Baselines on the Tier 1 Corpus

Table 3.4 shows the performance of the models. Similar models are grouped together. The first column shows the names of the model groups and the second column shows the names of the models. For each model, we show values for the four automatic metrics, BLEU, fragment accuracy (FragAcc), exact-match accuracy (XM), and top-5 accuracy (Acc@5). We repeated each experiment 3 times, with different random initialization each time, and computed the averages of each automated metric. We performed statistical significance tests—under significance level $p < 0.05$ using the bootstrap method [24]—to compare each pair of models. We use bold text to highlight the best value for each automatic metric, and gray background for baseline models. We make several observations:

Finding #1: The best overall performance (BLEU = 47.2) is obtained using the multi-input model with lemma statement and chopped kernel tree as inputs, which also includes copy and attention mechanisms (Stmt+CKTree+attn+copy). The improvements over all other models are statistically significant and all automatic metrics are consistent in identifying the best model. This shows the importance of using Coq's execution data and focusing only on certain parts of the kernel trees.

Finding #2: The copy mechanism brings statistically significant improvements to all models. This can be clearly observed by comparing groups 1 and 3 in the table, as well as groups 2 and 4. For example, BLEU for Stmt+attn and Stmt+attn+copy are 26.9 and 38.9, respectively. We believe that the copy mechanism plays an important role because many subtokens are specific to the file context and do not appear in the fixed vocabulary learned on the files in training set.

Finding #3: Using chopped trees greatly improves performance of models and the

Table 3.4: Results of ROOSTERIZE models and baselines trained on the tier 1 corpus and evaluated on the tier 1 corpus.

Group	Model	BLEU	FragAcc	XM	Acc@5
Multi-input +attn +copy	Stmt+CKTree+CSTree+attn+copy	45.4	22.2%	7.5%	16.5%
	Stmt+CKTree+attn+copy	**47.2**	**24.9%**	**9.6%**	**18.0%**
	Stmt+CSTree+attn+copy	37.7	18.1%	6.1%	10.6%
	CKTree+CSTree+attn+copy	45.4	22.9%	7.6%	15.3%
Single-input +attn +copy	CKTree+attn+copy	42.9	19.8%	5.0%	11.7%
	CSTree+attn+copy	39.8	18.3%	6.8%	12.2%
	KTree+attn+copy	37.0	14.2%	2.2%	8.4%
	STree+attn+copy	31.0	10.8%	2.8%	6.1%
	Stmt+attn+copy	38.9	19.4%	6.9%	11.6%
Multi-input +attn	Stmt+CKTree+CSTree+attn	24.5	8.6%	0.4%	0.9%
	Stmt+CKTree+attn	25.6	8.5%	0.9%	1.7%
	Stmt+CSTree+attn	23.8	8.2%	0.8%	1.6%
	CKTree+CSTree+attn	28.4	10.9%	1.8%	3.4%
Single-input +attn	CKTree+attn	19.5	4.9%	0.6%	1.3%
	CSTree+attn	28.9	12.1%	1.5%	2.9%
	KTree+attn	14.1	1.6%	0.0%	0.0%
	STree+attn	8.8	1.0%	0.0%	0.0%
	Stmt+attn	26.9	11.1%	1.1%	2.5%
Multi-input	Stmt+CKTree+CSTree	17.7	3.5%	0.1%	0.2%
	Stmt+CKTree	19.5	4.5%	0.1%	0.3%
	Stmt+CSTree	12.6	0.6%	0.0%	0.0%
	CKTree+CSTree	16.7	2.4%	0.0%	0.1%
Single-input	CKTree	15.5	1.6%	0.0%	0.0%
	CSTree	14.5	0.8%	0.1%	0.1%
	KTree	12.0	0.6%	0.0%	0.0%
	STree	5.7	0.4%	0.0%	0.0%
	Stmt	20.0	4.7%	0.1%	0.3%
-	Retrieval-based	28.3	10.0%	0.2%	0.3%

improvements brought by upgrading KTree to CKTree or STree to CSTree are statistically significant. For example, this can be clearly seen in the second group: BLEU for KTree+attn+copy and CKTree+attn+copy are 37.0 and 42.9, respectively. We believe that the size of the original trees, and a lot of irrelevant data in those trees, hurt the performance. The fact that CKTree and CSTree both perform much better than using KTree or STree across all groups indicate that the chopped trees could be viewed as a form of supervised attention with flat values that helps later attention layers to focus better.

Finding #4: Although chopped syntax tree with attention outperforms (statistically significant) chopped kernel tree with attention (BLEU 28.9 vs. 19.5), chopped kernel tree with attention and copy by far outperforms (statistically significant) chopped syntax tree with attention and copy (BLEU 42.9 vs. 39.8). The copy mechanism helps kernel trees much more than the syntax trees, because the mathematical notations and symbols in the syntax trees get expanded to their names in the kernel trees, and some of them are needed as a part of the lemma names.

Finding #5: Lemma statement and syntax tree do not work well together, primarily because the two representations contain mostly the same information. In which case, a model taking both as inputs may not work as well as using only one of the inputs, because more parameters need to be trained.

Finding #6: The retrieval-based baseline, which is the strongest among baselines, outperforms several encoder-decoder models without attention and copy or with only attention, but is worse than (statistically significant) all models with both attention and copy mechanisms enabled.

3.8.2 RQ2: Ablation Study on Tree Chopping

In order to corroborate the effectiveness of ROOSTERIZE's tree chopping heuristics, we designed an ablation study that applies three different sets of chopping heuristics and compares them with the one in ROOSTERIZE (Section 3.3.2). The three sets

of chopping heuristics are:

- **Keep-category chopping**. This set of heuristics is almost the same as ROOSTERIZE chopping, except that it keeps the category of a referenced name in kernel trees (e.g., whether it is a constant or inductive type), since that semantic information could be relevant for naming.

- **Rule-based chopping**. Removes all nodes after depth 10 for syntax and kernel trees. This is similar to the proof kernel tree processing heuristics used in ML4PG [81].

- **Random chopping**. Randomly removes a subset of nodes from syntax and kernel trees so that the resulting trees have the same average number of nodes compared to ROOSTERIZE's chopped trees, i.e., the heuristic removes 90.9% nodes from syntax trees and 91.4% nodes from kernel trees.

We performed the ablation study by training models on the tier 1 corpus and evaluating on the tier 1 corpus.

Table 3.5 shows the results of the ablation study. The models using the same chopping heuristics are grouped together. We make several observations:

- Among keep-category chopping models, Stmt+CKTree+CSTree+attn+copy and Stmt+CKTree+attn+copy perform the best, and they have performance similar to Stmt+CKTree+attn+copy using ROOSTERIZE chopping (ROOSTERIZE's best model). The measured differences between these three models are not statistically significant, under significance level $p < 0.05$ using the bootstrap method [24]. This indicates that although the category of a referenced name may contain some relevant semantic information, the most relevant information is already preserved by ROOSTERIZE chopping heuristics.

Table 3.5: Results of the ablation study on tree chopping.

Group	Model	BLEU	FragAcc	XM	Acc@5
ROOSTERIZE Chopping	Stmt+CKTree+CSTree+attn+copy	45.4	22.2%	7.5%	16.5%
	Stmt+CKTree+attn+copy	**47.2**	24.9%	**9.6%**	18.0%
	Stmt+CSTree+attn+copy	37.7	18.1%	6.1%	10.6%
	CKTree+CSTree+attn+copy	45.4	22.9%	7.6%	15.3%
Keep-category Chopping	Stmt+CKTree+CSTree+attn+copy	46.8	**25.3%**	9.5%	**19.0%**
	Stmt+CKTree+attn+copy	**47.2**	25.2%	9.4%	18.0%
	Stmt+CSTree+attn+copy	37.1	17.9%	6.2%	10.5%
	CKTree+CSTree+attn+copy	46.4	22.6%	7.5%	15.0%
Rule-based Chopping	Stmt+CKTree+CSTree+attn+copy	37.0	17.7%	5.9%	10.5%
	Stmt+CKTree+attn+copy	38.8	19.7%	6.7%	11.0%
	Stmt+CSTree+attn+copy	36.9	16.0%	6.3%	10.5%
	CKTree+CSTree+attn+copy	13.2	0.4%	0.0%	0.0%
Random Chopping	Stmt+CKTree+CSTree+attn+copy	37.7	19.2%	6.7%	10.9%
	Stmt+CKTree+attn+copy	38.5	19.3%	7.3%	11.3%
	Stmt+CSTree+attn+copy	38.0	18.6%	7.1%	11.4%
	CKTree+CSTree+attn+copy	17.1	2.7%	0.1%	0.1%

- The models using rule-based chopping and random chopping have poor performance. This indicates that the performance gain achieved by ROOSTERIZE through chopping is not only due to the size reduction of the input trees, but also due to the relevant information retained by our chopping heuristics.

3.8.3 RQ3: Roosterize vs. Baselines on Different Training, Validation, and Evaluation Sets

We present the results of ROOSTERIZE vs. Baselines by: (1) training on the all tiers corpus and evaluating on the all tiers (Table 3.6), tier 1 (Table 3.7), tier 2 (Table 3.8), and tier 3 (Table 3.9) corpus; (2) training on the tier 1 corpus and evaluating on the all tiers (Table 3.10), tier 2 (Table 3.11), and tier 3 (Table 3.12) corpus; (3) training and evaluating on the tier 2 (Table 3.13) and tier 3 (Table 3.14) corpus.

All our observations in Section 3.8.1 on training and testing on our original corpus (here, tier 1) hold when training and testing on all tiers. Additionally, we make

Table 3.6: Results of ROOSTERIZE models and baselines trained on the all tiers corpus and evaluated on the all tiers corpus.

Group	Model	BLEU	FragAcc	XM	Acc@5
Multi-input +attn +copy	Stmt+CKTree+CSTree+attn+copy	43.2	23.2%	7.1%	15.5%
	Stmt+CKTree+attn+copy	**47.2**	**26.1%**	**10.3%**	**19.0%**
	Stmt+CSTree+attn+copy	34.9	18.0%	4.9%	10.7%
	CKTree+CSTree+attn+copy	44.2	22.2%	7.4%	14.8%
Single-input +attn +copy	CKTree+attn+copy	44.1	20.9%	5.8%	13.1%
	CSTree+attn+copy	39.0	19.1%	7.9%	13.3%
	KTree+attn+copy	35.4	14.6%	1.2%	6.4%
	STree+attn+copy	31.3	14.2%	3.4%	7.1%
	Stmt+attn+copy	39.7	20.8%	7.5%	13.6%
Multi-input +attn	Stmt+CKTree+CSTree+attn	23.1	7.9%	1.1%	2.0%
	Stmt+CKTree+attn	27.3	10.9%	1.6%	3.0%
	Stmt+CSTree+attn	23.6	9.5%	1.7%	3.0%
	CKTree+CSTree+attn	26.6	10.4%	2.5%	4.5%
Single-input +attn	CKTree+attn	22.8	7.0%	1.0%	1.7%
	CSTree+attn	31.0	13.1%	2.5%	4.8%
	KTree+attn	13.5	2.0%	0.1%	0.4%
	STree+attn	11.5	1.8%	0.0%	0.1%
	Stmt+attn	27.5	11.0%	1.1%	2.0%
Multi-input	Stmt+CKTree+CSTree	18.2	4.3%	0.2%	0.4%
	Stmt+CKTree	20.3	5.5%	0.4%	0.8%
	Stmt+CSTree	11.2	0.1%	0.0%	0.0%
	CKTree+CSTree	12.1	0.8%	0.0%	0.0%
Single-input	CKTree	14.1	1.3%	0.0%	0.0%
	CSTree	14.4	1.1%	0.1%	0.1%
	KTree	11.1	0.0%	0.0%	0.0%
	STree	10.8	0.0%	0.0%	0.0%
	Stmt	20.3	5.4%	0.3%	0.5%
-	Retrieval-based	28.2	10.9%	0.6%	0.8%

Table 3.7: Results of ROOSTERIZE models and baselines trained on the all tiers corpus and evaluated on the tier 1 corpus.

Group	Model	BLEU	FragAcc	XM	Acc@5
Multi-input +attn +copy	Stmt+CKTree+CSTree+attn+copy	45.0	23.5%	7.6%	16.9%
	Stmt+CKTree+attn+copy	**49.3**	**26.7%**	**11.1%**	**20.6%**
	Stmt+CSTree+attn+copy	35.0	17.3%	4.8%	10.4%
	CKTree+CSTree+attn+copy	45.9	22.3%	8.1%	16.3%
Single-input +attn +copy	CKTree+attn+copy	45.8	21.3%	6.1%	14.1%
	CSTree+attn+copy	39.4	18.8%	7.9%	13.4%
	KTree+attn+copy	36.4	14.6%	1.0%	6.3%
	STree+attn+copy	31.2	13.8%	3.6%	7.5%
	Stmt+attn+copy	40.2	20.4%	7.7%	14.0%
Multi-input +attn	Stmt+CKTree+CSTree+attn	23.5	7.8%	1.1%	2.0%
	Stmt+CKTree+attn	28.1	10.9%	1.6%	3.0%
	Stmt+CSTree+attn	23.9	8.9%	1.5%	2.8%
	CKTree+CSTree+attn	27.8	10.6%	2.5%	4.7%
Single-input +attn	CKTree+attn	23.4	6.8%	1.0%	1.8%
	CSTree+attn	32.1	13.3%	2.7%	5.0%
	KTree+attn	13.6	1.9%	0.2%	0.5%
	STree+attn	11.4	1.9%	0.0%	0.1%
	Stmt+attn	27.9	10.7%	1.0%	1.9%
Multi-input	Stmt+CKTree+CSTree	18.4	4.2%	0.2%	0.4%
	Stmt+CKTree	20.6	5.4%	0.4%	0.7%
	Stmt+CSTree	11.3	0.1%	0.0%	0.0%
	CKTree+CSTree	12.2	0.8%	0.0%	0.0%
Single-input	CKTree	14.3	1.3%	0.0%	0.0%
	CSTree	14.5	1.2%	0.1%	0.2%
	KTree	11.1	0.0%	0.0%	0.0%
	STree	10.8	0.0%	0.0%	0.0%
	Stmt	20.6	5.2%	0.4%	0.5%
-	Retrieval-based	29.0	10.5%	0.3%	0.3%

Table 3.8: Results of ROOSTERIZE models and baselines trained on the all tiers corpus and evaluated on the tier 2 corpus.

Group	Model	BLEU	FragAcc	XM	Acc@5
Multi-input +attn +copy	Stmt+CKTree+CSTree+attn+copy	31.7	20.7%	5.8%	15.0%
	Stmt+CKTree+attn+copy	35.5	25.0%	**14.2%**	**21.7%**
	Stmt+CSTree+attn+copy	35.6	18.9%	7.5%	13.3%
	CKTree+CSTree+attn+copy	35.4	**29.2%**	9.2%	13.3%
Single-input +attn +copy	CKTree+attn+copy	32.4	21.2%	6.7%	12.5%
	CSTree+attn+copy	**38.7**	23.9%	10.8%	18.3%
	KTree+attn+copy	29.2	19.9%	6.7%	10.0%
	STree+attn+copy	27.5	11.2%	1.7%	4.2%
	Stmt+attn+copy	33.2	17.8%	7.5%	16.7%
Multi-input +attn	Stmt+CKTree+CSTree+attn	23.2	10.6%	2.5%	6.7%
	Stmt+CKTree+attn	26.0	14.2%	4.2%	10.0%
	Stmt+CSTree+attn	25.9	17.6%	6.7%	10.0%
	CKTree+CSTree+attn	22.4	15.4%	7.5%	8.3%
Single-input +attn	CKTree+attn	23.2	10.8%	3.3%	5.0%
	CSTree+attn	30.2	18.8%	6.7%	11.7%
	KTree+attn	13.7	2.9%	0.0%	0.0%
	STree+attn	9.6	1.7%	0.0%	0.0%
	Stmt+attn	27.0	15.1%	4.2%	10.0%
Multi-input	Stmt+CKTree+CSTree	20.4	7.9%	1.7%	2.5%
	Stmt+CKTree	18.8	7.6%	0.8%	2.5%
	Stmt+CSTree	11.7	0.4%	0.0%	0.0%
	CKTree+CSTree	11.9	0.4%	0.0%	0.0%
Single-input	CKTree	15.0	2.5%	0.0%	0.0%
	CSTree	14.8	1.8%	0.0%	0.0%
	KTree	12.1	1.4%	0.0%	0.0%
	STree	12.3	0.0%	0.0%	0.0%
	Stmt	23.6	13.7%	0.8%	0.8%
-	Retrieval-based	31.0	27.5%	2.5%	7.5%

Table 3.9: Results of ROOSTERIZE models and baselines trained on the all tiers corpus and evaluated on the tier 3 corpus.

Group	Model	BLEU	FragAcc	XM	Acc@5
Multi-input +attn +copy	Stmt+CKTree+CSTree+attn+copy	33.6	21.4%	3.8%	6.8%
	Stmt+CKTree+attn+copy	36.0	22.6%	4.3%	8.6%
	Stmt+CSTree+attn+copy	34.4	22.1%	5.0%	**11.9%**
	CKTree+CSTree+attn+copy	34.5	20.3%	2.8%	5.6%
Single-input +attn +copy	CKTree+attn+copy	34.5	17.8%	2.8%	6.3%
	CSTree+attn+copy	36.3	20.5%	**6.9%**	11.6%
	KTree+attn+copy	29.6	12.1%	1.5%	5.9%
	STree+attn+copy	32.5	17.6%	2.5%	5.1%
	Stmt+attn+copy	**37.4**	**24.2%**	6.1%	10.2%
Multi-input +attn	Stmt+CKTree+CSTree+attn	20.3	7.7%	0.8%	1.3%
	Stmt+CKTree+attn	22.1	10.2%	0.8%	1.5%
	Stmt+CSTree+attn	20.7	11.2%	1.7%	2.5%
	CKTree+CSTree+attn	19.2	7.7%	1.3%	2.0%
Single-input +attn	CKTree+attn	18.9	7.0%	0.5%	0.7%
	CSTree+attn	24.2	10.4%	0.8%	2.1%
	KTree+attn	12.7	2.2%	0.0%	0.2%
	STree+attn	12.2	1.5%	0.0%	0.0%
	Stmt+attn	24.4	12.5%	0.8%	1.2%
Multi-input	Stmt+CKTree+CSTree	16.2	4.3%	0.2%	0.3%
	Stmt+CKTree	18.5	5.7%	0.3%	0.8%
	Stmt+CSTree	10.9	0.1%	0.0%	0.0%
	CKTree+CSTree	12.0	0.7%	0.0%	0.0%
Single-input	CKTree	12.5	0.6%	0.0%	0.0%
	CSTree	13.5	0.1%	0.0%	0.0%
	KTree	11.0	0.0%	0.0%	0.0%
	STree	10.6	0.0%	0.0%	0.0%
	Stmt	17.6	5.0%	0.0%	0.5%
-	Retrieval-based	23.7	11.6%	2.5%	3.0%

Table 3.10: Results of ROOSTERIZE models and baselines trained on the tier 1 corpus and evaluated on the all tiers corpus.

Group	Model	BLEU	FragAcc	XM	Acc@5
Multi-input +attn +copy	Stmt+CKTree+CSTree+attn+copy	43.3	21.4%	6.8%	14.9%
	Stmt+CKTree+attn+copy	**44.5**	**23.9%**	**8.5%**	**16.2%**
	Stmt+CSTree+attn+copy	36.6	17.5%	5.6%	10.1%
	CKTree+CSTree+attn+copy	43.3	22.3%	6.8%	13.8%
Single-input +attn +copy	CKTree+attn+copy	40.7	19.2%	4.5%	10.6%
	CSTree+attn+copy	38.8	18.3%	6.4%	11.6%
	KTree+attn+copy	35.1	13.1%	2.0%	7.3%
	STree+attn+copy	30.9	10.9%	2.6%	5.9%
	Stmt+attn+copy	38.0	19.1%	6.4%	10.9%
Multi-input +attn	Stmt+CKTree+CSTree+attn	23.8	8.5%	0.4%	0.9%
	Stmt+CKTree+attn	24.6	8.4%	0.9%	1.7%
	Stmt+CSTree+attn	23.2	8.0%	0.7%	1.5%
	CKTree+CSTree+attn	27.1	10.5%	1.7%	3.2%
Single-input +attn	CKTree+attn	18.9	4.7%	0.5%	1.2%
	CSTree+attn	27.9	11.9%	1.6%	2.8%
	KTree+attn	13.7	1.5%	0.0%	0.0%
	STree+attn	8.8	1.0%	0.0%	0.0%
	Stmt+attn	26.0	10.8%	1.2%	2.4%
Multi-input	Stmt+CKTree+CSTree	17.4	3.4%	0.1%	0.2%
	Stmt+CKTree	19.1	4.4%	0.1%	0.2%
	Stmt+CSTree	12.6	0.6%	0.0%	0.0%
	CKTree+CSTree	16.2	2.2%	0.0%	0.0%
Single-input	CKTree	15.2	1.6%	0.0%	0.0%
	CSTree	14.4	0.8%	0.1%	0.1%
	KTree	12.2	0.6%	0.0%	0.0%
	STree	5.7	0.3%	0.0%	0.0%
	Stmt	19.4	4.6%	0.1%	0.3%
-	Retrieval-based	26.9	9.6%	0.3%	0.4%

Table 3.11: Results of ROOSTERIZE models and baselines trained on the tier 1 corpus and evaluated on the tier 2 corpus.

Group	Model	BLEU	FragAcc	XM	Acc@5
Multi-input +attn +copy	Stmt+CKTree+CSTree+attn+copy	34.2	22.8%	**9.2%**	11.7%
	Stmt+CKTree+attn+copy	35.0	**26.7%**	8.3%	14.2%
	Stmt+CSTree+attn+copy	36.7	14.2%	6.7%	14.2%
	CKTree+CSTree+attn+copy	33.9	25.8%	**9.2%**	**16.7%**
Single-input +attn +copy	CKTree+attn+copy	34.8	25.0%	8.3%	14.2%
	CSTree+attn+copy	35.6	20.1%	7.5%	15.8%
	KTree+attn+copy	25.3	11.2%	3.3%	7.5%
	STree+attn+copy	33.3	14.2%	2.5%	6.7%
	Stmt+attn+copy	**42.4**	20.8%	6.7%	15.8%
Multi-input +attn	Stmt+CKTree+CSTree+attn	24.8	12.8%	1.7%	5.8%
	Stmt+CKTree+attn	25.5	13.2%	5.0%	6.7%
	Stmt+CSTree+attn	22.8	10.4%	0.0%	5.0%
	CKTree+CSTree+attn	25.6	15.6%	5.8%	7.5%
Single-input +attn	CKTree+attn	19.4	6.7%	0.8%	3.3%
	CSTree+attn	28.0	17.5%	6.7%	7.5%
	KTree+attn	13.1	1.4%	0.0%	0.8%
	STree+attn	10.9	2.1%	0.0%	0.0%
	Stmt+attn	28.5	17.9%	5.0%	6.7%
Multi-input	Stmt+CKTree+CSTree	20.4	10.4%	0.8%	2.5%
	Stmt+CKTree	17.9	6.7%	0.0%	0.0%
	Stmt+CSTree	14.3	0.4%	0.0%	0.0%
	CKTree+CSTree	15.6	4.6%	0.0%	0.0%
Single-input	CKTree	15.3	3.6%	0.0%	0.0%
	CSTree	14.9	2.2%	0.0%	0.0%
	KTree	12.2	0.4%	0.0%	0.0%
	STree	6.6	0.0%	0.0%	0.0%
	Stmt	20.9	9.9%	2.5%	4.2%
-	Retrieval-based	24.8	14.6%	2.5%	7.5%

Table 3.12: Results of ROOSTERIZE models and baselines trained on the tier 1 corpus and evaluated on the tier 3 corpus.

Group	Model	BLEU	FragAcc	XM	Acc@5
Multi-input +attn +copy	Stmt+CKTree+CSTree+attn+copy	31.2	15.9%	2.1%	5.4%
	Stmt+CKTree+attn+copy	29.1	16.9%	1.3%	4.5%
	Stmt+CSTree+attn+copy	29.8	13.8%	2.1%	5.8%
	CKTree+CSTree+attn+copy	31.3	**17.9%**	1.7%	3.5%
Single-input +attn +copy	CKTree+attn+copy	27.6	13.8%	1.0%	2.6%
	CSTree+attn+copy	**33.1**	17.8%	**3.8%**	**6.8%**
	KTree+attn+copy	24.3	6.4%	0.0%	1.2%
	STree+attn+copy	30.4	11.1%	1.8%	4.3%
	Stmt+attn+copy	30.9	16.7%	3.1%	5.8%
Multi-input +attn	Stmt+CKTree+CSTree+attn	19.5	7.6%	0.3%	0.3%
	Stmt+CKTree+attn	18.3	6.8%	0.2%	0.3%
	Stmt+CSTree+attn	19.4	6.3%	0.3%	0.7%
	CKTree+CSTree+attn	18.9	7.4%	0.3%	0.8%
Single-input +attn	CKTree+attn	15.0	2.8%	0.0%	0.2%
	CSTree+attn	21.3	9.4%	1.2%	1.2%
	KTree+attn	10.7	0.4%	0.0%	0.0%
	STree+attn	7.7	0.5%	0.0%	0.0%
	Stmt+attn	19.5	7.5%	0.5%	0.8%
Multi-input	Stmt+CKTree+CSTree	14.9	1.6%	0.0%	0.0%
	Stmt+CKTree	16.5	3.8%	0.0%	0.0%
	Stmt+CSTree	12.4	0.5%	0.0%	0.0%
	CKTree+CSTree	13.4	1.0%	0.0%	0.0%
Single-input	CKTree	13.1	1.2%	0.0%	0.0%
	CSTree	13.5	0.7%	0.0%	0.0%
	KTree	12.9	1.0%	0.0%	0.0%
	STree	5.8	0.0%	0.0%	0.0%
	Stmt	15.7	3.2%	0.0%	0.0%
-	Retrieval-based	18.3	5.7%	0.0%	0.0%

Table 3.13: Results of ROOSTERIZE models and baselines trained on the tier 2 corpus and evaluated on the tier 2 corpus.

Group	Model	BLEU	FragAcc	XM	Acc@5
Multi-input +attn +copy	Stmt+CKTree+CSTree+attn+copy	31.8	17.6%	**5.8%**	9.2%
	Stmt+CKTree+attn+copy	29.5	17.5%	4.2%	8.3%
	Stmt+CSTree+attn+copy	28.8	9.6%	0.0%	2.5%
	CKTree+CSTree+attn+copy	30.5	15.8%	5.0%	**10.8%**
Single-input +attn +copy	CKTree+attn+copy	**33.6**	18.2%	4.2%	5.8%
	CSTree+attn+copy	29.3	9.9%	0.8%	5.0%
	KTree+attn+copy	31.2	17.2%	4.2%	7.5%
	STree+attn+copy	25.0	2.8%	0.0%	0.0%
	Stmt+attn+copy	29.7	15.4%	2.5%	5.8%
Multi-input +attn	Stmt+CKTree+CSTree+attn	22.3	10.0%	1.7%	1.7%
	Stmt+CKTree+attn	21.4	8.1%	0.8%	2.5%
	Stmt+CSTree+attn	24.3	14.2%	1.7%	7.5%
	CKTree+CSTree+attn	18.0	6.9%	0.8%	3.3%
Single-input +attn	CKTree+attn	15.7	1.1%	0.0%	0.0%
	CSTree+attn	17.5	3.5%	0.0%	0.0%
	KTree+attn	11.2	0.0%	0.0%	0.0%
	STree+attn	11.3	0.0%	0.0%	0.0%
	Stmt+attn	20.6	7.8%	0.8%	6.7%
Multi-input	Stmt+CKTree+CSTree	15.5	3.5%	0.0%	0.0%
	Stmt+CKTree	15.3	2.6%	0.0%	0.0%
	Stmt+CSTree	11.4	0.3%	0.0%	0.0%
	CKTree+CSTree	11.1	0.0%	0.0%	0.0%
Single-input	CKTree	9.7	1.4%	0.0%	0.0%
	CSTree	12.8	0.0%	0.0%	0.0%
	KTree	8.7	0.0%	0.0%	0.0%
	STree	6.8	0.0%	0.0%	0.0%
	Stmt	14.6	1.9%	0.0%	0.0%
-	Retrieval-based	27.7	**25.2%**	0.0%	0.0%

Table 3.14: Results of ROOSTERIZE models and baselines trained on the tier 3 corpus and evaluated on the tier 3 corpus.

Group	Model	BLEU	FragAcc	XM	Acc@5
Multi-input +attn +copy	Stmt+CKTree+CSTree+attn+copy	33.5	19.5%	2.3%	5.9%
	Stmt+CKTree+attn+copy	31.2	16.9%	0.8%	6.3%
	Stmt+CSTree+attn+copy	32.0	17.3%	3.3%	**7.6%**
	CKTree+CSTree+attn+copy	31.8	15.5%	1.3%	5.0%
Single-input +attn +copy	CKTree+attn+copy	29.9	15.7%	1.3%	6.1%
	CSTree+attn+copy	32.7	15.3%	**4.0%**	6.3%
	KTree+attn+copy	30.7	12.4%	1.3%	5.0%
	STree+attn+copy	28.4	12.6%	2.0%	6.4%
	Stmt+attn+copy	**34.3**	**19.8%**	3.3%	5.3%
Multi-input +attn	Stmt+CKTree+CSTree+attn	14.5	1.4%	0.0%	0.0%
	Stmt+CKTree+attn	17.9	5.3%	0.0%	0.0%
	Stmt+CSTree+attn	19.0	6.6%	0.2%	0.2%
	CKTree+CSTree+attn	12.4	2.2%	0.0%	0.0%
Single-input +attn	CKTree+attn	14.8	2.3%	0.0%	0.0%
	CSTree+attn	17.0	2.8%	0.0%	0.0%
	KTree+attn	13.1	0.2%	0.0%	0.0%
	STree+attn	4.8	0.3%	0.0%	0.0%
	Stmt+attn	17.6	5.8%	0.0%	0.0%
Multi-input	Stmt+CKTree+CSTree	12.6	0.7%	0.0%	0.0%
	Stmt+CKTree	14.4	1.4%	0.0%	0.0%
	Stmt+CSTree	9.1	0.2%	0.0%	0.0%
	CKTree+CSTree	11.1	0.0%	0.0%	0.0%
Single-input	CKTree	11.7	0.1%	0.0%	0.0%
	CSTree	10.1	0.0%	0.0%	0.0%
	KTree	14.3	0.2%	0.0%	0.0%
	STree	14.4	0.2%	0.0%	0.0%
	Stmt	14.4	1.4%	0.0%	0.0%
-	Retrieval-based	22.1	9.6%	2.5%	3.0%

the following observations based on the results of models using different combinations of training, validation, and evaluation sets:

- Training on all tiers helps ROOSTERIZE obtain better performance, although the corpus includes some noise from tier 2 and tier 3 projects. This observation is based on comparing the results of training on different sets and testing on the same set. For example, when testing on all tiers, the best BLEU score among models trained on all tiers (47.2, see Table 3.6) is higher than the best score for models trained on tier 1 (44.5, see Table 3.10). As another example, when testing on tier 2, the best BLEU score among models trained on all tiers is 38.7 (see Table 3.8), which is higher than the best score among models trained on tier 2, namely, 33.6 (see Table 3.13).
- Tier 2 and tier 3 projects are indeed less conforming to MathComp naming conventions than tier 1 projects, confirming the judgment of domain experts. With the same models trained on all tiers, ROOSTERIZE's best BLEU score on the tier 1 evaluation set (49.3) is greater than the best BLEU score on the tier 2 evaluation set (38.7), and the latter is greater than the best BLEU score on the tier 3 evaluation set (37.4). The same relationships hold for the models trained on tier 1.

3.8.4 RQ4: Generalization Case Study

The project in our left-out corpus, infotheo, consists of 81 Coq files, and contains 1,891 lemmas. We randomly split the files into training, validation, and evaluation sets which contain 40%, 10%, 50% of the files, respectively. After splitting, there were 580 lemmas in the training set, 144 lemmas in the validation set, and 1,167 lemmas in the evaluation set.

Table 3.15 shows the results of applying ROOSTERIZE with the best model, trained on the all tiers corpus, on infotheo without and with additional training. The first column shows the number of lemmas from the infotheo training set used for additional training. The rest of the columns show the four automatic metrics. We can

Table 3.15: Results of the generalization study with ROOSTERIZE models trained on all tiers, potentially further trained on the left-out corpus, and evaluated on the left-out corpus.

#Lemmas	BLEU	FragAcc	XM	Acc@5
0	33.9	21.3%	4.4%	8.9%
105	32.6	21.5%	3.3%	5.3%
223	34.1	22.7%	3.8%	6.9%
505	35.7	24.3%	5.0%	8.7%
580	**37.4**	**26.5%**	**7.4%**	**12.5%**

observe that applying ROOSTERIZE without additional training achieves moderate performance (BLEU = 33.9). With some additional training, performance can be markedly improved (up to a BLEU score of 37.4 when training on all 580 lemmas).

3.8.5 RQ5: Manual Quality Analysis

While generated lemma names may not always match the manually written ones in the training set, they can still be semantically valid and conform to prevailing conventions. However, such name properties are not reflected in our automatic evaluation metrics, since these metrics only consider exactly matched tokens as correct. To obtain a more complete evaluation, we therefore performed a manual quality analysis of generated lemma names from ROOSTERIZE by applying the model trained on the tier 1 corpus to a Coq project in the tier 3 corpus, the PCM library [138]. This project depends on MathComp, and follows, to a degree, many of the MathComp coding conventions. The PCM library consists of 12 Coq files, and contains 690 lemmas.

We ran ROOSTERIZE with the best model (Stmt+CKTree+attn+copy) on the PCM library to get the top-1 predictions for all lemma names. Overall, the ROOSTERIZE predictions achieved a BLEU score of 36.3 and a fragment accuracy of 17%, and 36 predictions (5%) exactly match the existing lemma names. Next, we asked the maintainer of the PCM library to evaluate the remaining 654 lemma names (those that do not match exactly) and send us feedback.

Table 3.16: Representative examples of ROOSTERIZE's predictions and developer comments in our qualitative study.

Lemma statement: `p s : supp (kfilter p s) = filter p (supp s)` **Hand-written**: `supp_kfilt` **Roosterize**: `supp_kfilter` **Comment**: ✓ Using only `kfilt` has cognitive overhead.
Lemma statement: `g e k v f : path ord k (supp f) ->` `foldfmap g e (ins k v f) = g (k, v) (foldfmap g e f)` **Hand-written**: `foldf_ins` **Roosterize**: `foldfmap_ins` **Comment**: ✓ The whole function name is used in the suggested name.
Lemma statement: `m : @mapk A A V id m = m` **Hand-written**: `map_id` **Roosterize**: `mapk_id` **Comment**: ✓ The suggested name is less generic than the existing one.
Lemma statement: `: transitive (@ord T)` **Hand-written**: `trans` **Roosterize**: `ord_trans` **Comment**: ✓ Useful to add the `ord` prefix to the name.
Lemma statement: `s : sorted (@ord T) s -> sorted (@oleq T) s` **Hand-written**: `sorted_oleq` **Roosterize**: `ord_sorted` **Comment**: ✗ The conclusion content should have greater priority.
Lemma statement: `x y : total_spec x y (ord x y) (x == y) (ord y x)` **Hand-written**: `totalP` **Roosterize**: `ordP` **Comment**: ✗ Maybe this lemma should be named `ord_totalP`?
Lemma statement: `p1 p2 s : kfilter (predI p1 p2) s =` `kfilter p1 (kfilter p2 s)` **Hand-written**: `kfilter_predI` **Roosterize**: `eq_kfilter` **Comment**: ✗ The suggested name is too generic.

The maintainer spent one day on the task and provided comments on 150 suggested names. We analyzed these comments to identify patterns and trends. He found that 20% of the suggested names he inspected were of good quality, out of which more than half were of high quality. Considering that the analysis was of top-1 predictions excluding exact matches, we find these figures encouraging. For low-quality names, a clear trend was that they were often "too generic". Similar observations have been made about the results from encoder-decoder models in dialog generation [115, 188]. In contrast, useful predictions were typically able to expand or elaborate on name components that are intuitively too concise, e.g., replacing `kfilt` with `kfilter`. Ta-

ble 3.16 lists examples that are representative of these trends; checkmarks indicate useful predictions, while crosses indicate unsuitability. We also include comments from the maintainer. As illustrated by the comments, even predictions considered unsuitable may contain useful parts.

3.9 Discussion

Our toolchain builds on Coq 8.10.2, and thus we only used projects that support this version. However, we do not expect any fundamental obstacles in supporting future Coq releases. Thanks to the use of OCaml metaprogramming via PPX, which allowed eliding explicit references to the internal structure of Coq datatypes, SerAPI itself and our extensions to it are expected to require only modest effort to maintain as Coq evolves.

Our models and toolchain may not be applicable to Coq projects unrelated to the MathComp family of projects, i.e., projects which do not follow any MathComp conventions. To the best of our knowledge, MathComp's coding conventions are the most recognizable and well-documented in the Coq community; suggesting coding conventions based on learning from projects unrelated to MathComp are likely to give more ambiguous results that are difficult to validate manually. Our case study also included generating predictions for a project outside the MathComp family, the PCM library, with encouraging results.

Our models are in principle applicable to proof assistants with similar foundations, such as Lean [52]. However, the version at the time of our work, Lean 3, does not provide serialization of internal data structures as SerAPI does for Coq, which prevents direct application of our toolchain. Application of our models to proof assistants with different foundations and proof-checking toolchains, such as Isabelle/HOL, is even less straightforward, although the Archive of Formal Proofs (AFP) contains many projects with high-quality lemma names [59].

3.10 Summary

We presented novel techniques, based on neural networks, for learning and suggesting lemma names in Coq verification projects. We designed and implemented multi-input encoder-decoder models that use Coq's internal data structures, including (chopped) syntax trees and kernel trees (extracted during execution). Additionally, we constructed a large corpus of high quality Coq code that will enable development and evaluation of future techniques for Coq. We performed an extensive evaluation of our models using the corpus. Our results show that the multi-input models, which use internal data structures (in particular data extracted during execution), substantially outperform several baselines; the model that uses the lemma statement tokens and the chopped kernel tree with attention and copy mechanism performs the best. Based on our findings, we believe that multi-input models leveraging key parts of internal data structures extracted from execution can play a critical role in producing high-quality lemma name predictions.

Chapter 4: Related Work

This chapter presents prior work in the area of ML for SE that are most related to this dissertation. We first review recent LLMs developed for SE tasks (Section 4.1), and present related work on combining software execution with ML models (Section 4.2). Then, we present recent ML models targeting code completion (Section 4.3), method naming (Section 4.4), enforcing coding conventions (Section 4.5), which are closely related to the test completion and lemma naming tasks studied in this dissertation. Finally, we summarize related work on test generation and recommendation (Section 4.6) and proof mining and automation (Section 4.7). More complete reviews of the ML for SE research area can be found in recent surveys [11, 125, 204, 208, 217] and online resources [7, 135].

4.1 LLMs for SE

The application of ML models for SE tasks has been an active research area in the past decades, which has been growing as the ML models become more powerful: early work could help developers with ranking tasks (e.g., in code completion [35, 178, 179]) using non-deep learning models, and recent deep learning models (e.g., RNNs, CNNs, transformers) can better handle code understanding and generation tasks. Transformer [200] quickly became the most popular deep learning model architecture because of its ability to scale to a large number of parameters and large amount of training data. The larger models, called *large language models (LLMs)* by the community, present some new "emergent abilities" unseen on smaller models [210], including multitask learning [170], few-shot learning [34], zero-shot reasoning [104], and instruction following [155]. While it is arguable if such emergent abilities translate to higher intelligence learned by the models [184], LLMs do achieve better performance in many natural language processing tasks, as well as SE tasks.

CodeBERT [60] is the first LLM for code, which is based on the BERT (encoder-only) architecture [54] and pre-trained both bimodal comment-code pairs data and unimodal code data. GraphCodeBERT [74] extends CodeBERT to incorporate code structures in the model, by adding a pre-training task of predicting data flow graphs. CodeGPT [125] is based on the GPT-2 (decoder-only) architecture [170], which is more suitable for code generation tasks compared to CodeBERT (with similar model size). PLBART [5] is the first encoder-decoder LLM for code, based on the BART architecture [113], pre-trained on Java and Python code from GitHub and technical text from StackOverflow. CodeT5 [206] is also an encoder-decoder LLM for code, based on the T5 architecture [171], and introduces a new pre-training task of predicting masked code identifiers. CoditT5 [222] extends CodeT5 with another new pre-training task of recovering code mutations to support code editing.

More recently, researchers and industry companies worked on scaling up the size of LLMs from millions to billions to trillions of parameters. Codex [40] is a branch of GPT-3 [34] (which targets natural language processing tasks) and is further pre-trained on GitHub code, and has been commercialized as the GitHub Copilot product [67]. AlphaCode [117] is also pre-trained on GitHub code and mainly targets competition-level coding problems on the Codeforces platform [133]. CodeGen [147, 148] is pre-trained on both natural language and GitHub code and mainly targets multi-turn program synthesize task. InCoder [62] is a decoder-only LLM pre-trained not only for left-to-right generation, but also infilling for completing the middle part of a code snippet given both left and right context. CodeGeeX [225] is pre-trained on a diverse source of code dataset and is evaluated on a new benchmark, HumanEval-X, composed of multiple programming languages. StarCoder [116] is an LLMs for code pre-trained by BigCode (a worldwide team of researchers led by Hugging Face and ServiceNow), whose goal is to make the model open source and accessible to the community. Some larger LLMs, such as OpenAI's GPT-4 [153] and Google's Bard [73], are pre-trained on larger amount of natural language and code data and target both natural language processing and SE tasks.

4.2 ML + Software Execution

Prior work explored the use of code execution data in ML for SE. Wang et al. [205] used the similarity of natural language descriptions and execution traces to detect duplicate bug reports. Wang et al. [202] proposed to train semantic code embeddings from execution traces, which can be used to improve the performance of program repair models. Wang and Su [201] blended syntactical and semantic code embeddings and applied them in a method naming model. Pei et al. [164] developed TREX for detecting similar binary functions, which learns from the traces of binary code from forced execution (i.e., directly executing a binary function with randomly initialized input states and ignoring the control flow). Pi et al. [166] proposed PoEt that improves the reasoning capabilities of language models by pre-training on code execution data. Shi et al. [191] proposed to improve code generation models' outputs using a minimum Bayes risk decoding algorithm based on execution results.

Recent benchmarks for evaluating ML models, especially LLMs, adopt software execution as an approach to check the correctness of models' predictions, in complementary to syntax-level similarities (which does not account for functional correctness) and human evaluation (which can be costly). HumanEval [40] is a benchmark of 164 hand-written Python code generation tasks, where the correctness of each generated code is checked by executing it on a set of test cases (input-output examples); HumanEval-X [225] is the multi-programming-language extension of HumanEval. MBPP (most basic Python problems) [15] is another benchmark of around 1,000 simple Python code generation tasks with test cases. We also evaluated TeCo and baselines on our test completion dataset using test execution, by measuring the percentage of predictions that are compilable and runnable (Chapter 2).

4.3 Code Completion

Code completion helps developers write code faster by predicting the incomplete part of a partial code snippet, which is usually the next tokens or statements.

Early code completion techniques, which have already been popular features in modern IDEs like Eclipse and IntelliJ IDEA, are based on programming language's grammar and type analysis and have low prediction accuracy. These limitations motivated researchers to improve code completion by leveraging the context code and code change histories. Robbes and Lanza [178, 179] improved type-analysis-based code completion by prioritizing the predictions with recently changed code. Bruch et al. [35] developed the Best Matching Neighbors algorithm, based on the k-Nearest-Neighbors algorithm, to suggest APIs with similar structural context during code completion; Proksch et al. [168] extended this algorithm to use Pattern-Based Bayesian Networks. Han et al. [76] proposed the abbreviation completion technique which completes several tokens from abbreviated input characters, using a Hidden Markov Model. Nguyen et al. [139] developed GraPacc, a graph-based, pattern-oriented, context-sensitive code completion technique.

Motivated by the study on the naturalness of code [83], researchers explored using statistical and neural language models for code completion. Raychev et al. [173] developed Slang, which allows developers to write code with several statements omitted as holes and then completes the holes using a combination of n-gram and RNN language modes. Li et al. [114] proposed to use a pointer network to help RNN language models to predict out-of-vocabulary tokens during code completion. Svyatkovskiy et al. [195] developed Pythia, an LSTM language model for code completion focusing on the Python programming language.

LLMs are good at the code completion task because they commonly use masked language modeling (i.e., predicting the next tokens or infilling missed tokens) as the primary pre-training task. As such, many recent code completion techniques are developed on top of LLMs. Svyatkovskiy et al. [196] presented IntelliCode Compose, a code completion framework based on a GPT-2 pre-trained on code which is distilled for better inference speed. Zhou et al. [226] studied using transfer learning to improve code completion, by applying pre-trained LLMs on programming languages

with limited data. Lu et al. [126] augmented an LLM with a code retrieval model allowing copying and referring to code with similar semantics.

The test completion task proposed in Chapter 2 is closely related to code completion, but different in the extra context of method under test and unique programming style that test code has. The evaluation results in Section 2.7 show that our test completion model, TeCo, outperforms the state-of-the-art code completion model, including CodeT5 and Codex, by integrating test execution with ML models.

4.4 Method Naming

Prior work on suggesting names mostly concerns method naming, i.e., suggesting method names based on method bodies. Liu et al. [122] used a similarity matching algorithm, based on deep representations of Java method names and bodies learned with Paragraph Vector and CNNs, to detect and fix inconsistent Java method names. Allamanis et al. [9] used logbilinear neural language models supplemented by additional manual features to predict Java method and class names. Method names have also been treated as short, descriptive "summaries" of its body; in this view, prior work has augmented attention mechanisms in CNNs [10], used sequence-to-sequence models to learn from descriptions (e.g., Javadoc comments) [64], utilized the tree-structure of the code in a hierarchical attention network [214], and trained semantic code embeddings from the method's execution traces [201]. Researchers have also applied ML models for method naming to detect and fix inconsistency names in open-source projects [78, 140].

The lemma naming task proposed in Chapter 3 is similar to method naming. However, unlike abstract syntax trees in programming languages like Java, the syntax and kernel trees in Coq contain considerable semantic information that needs to be extracted for suggesting accurate names. In the work closest to our domain, Aspinall and Kaliszyk [14] used a k-Nearest-Neighbors multi-label classifier on a corpus for the HOL Light proof assistant to suggest names of lemmas, but their technique is limited

to suggests names that exist in the training data and therefore does not generalize. To our knowledge, ROOSTERIZE is the first deep learning generation model for suggesting names in a proof assistant context.

Related to method naming, researchers also studied the comment generation task: summarizing method bodies into natural language comments [4, 89, 94, 109, 118, 145, 157–159, 209, 221]. While the outputs are different in terms of length and content, we expect that combining with execution can also improve the performance of comment generation ML models.

4.5 Enforcing Coding Conventions

Hindle et al. [83] proposed the concept of the naturalness of software, i.e., repetitive patterns in code that can be modeled by statistical and ML models. The initial study [83] was performed on C and Java code, and later work extended the concept of naturalness to multiple programming languages [172] and other domains including proofs [79] and hardware descriptions [110]. Allamanis et al. [8] used the concept of naturalness and statistical language models to learn and suggest coding conventions, including names, for Java, and Raychev et al. [174] used conditional random fields to learn and suggest coding conventions for JavaScript. ROOSTERIZE, introduced in Chapter 3, is the first model for learning and suggesting lemma naming coding conventions for the Coq proof assistant. We have also developed an RNN-based model for learning and suggesting formatting coding conventions in Coq [142].

4.6 Test Generation and Recommendation

Existing automatic test generation work includes fuzz/random testing [156, 218, 219], property-based testing [6, 31, 38, 41, 68, 86, 108], search-based testing [61, 77], and combinatorial testing [43]. The typical goal in automated test generation techniques, e.g., Randoop [156] and EvoSuite [61], is to achieve high code coverage of the code under test by generating a large amount of tests, either randomly or

systematically. However, the generated tests would not be added to the manually written tests in the code repository due to their low quality and the excessive amount. Some prior work explored improving the quality of the generated tests, for example: Holmes et al. [86] proposed to use relative LOC to guide the choosing of test generation targets; Reddy et al. [175] proposed to use reinforcement learning to guide the random input generator in property-based test generation; Lemieux et al. [111] proposed to use LLMs to generate test inputs to escape coverage plateaus. So far, these automated techniques are used only in addition to manually written tests. In contrast, TeCo focuses on improving developers' productivity when writing manual tests.

Another disadvantage of the automated test generation approaches is the lack of test oracles. To remedy that, prior work explored extracting test oracles from code comments, focusing on test oracles related to exceptional behaviors, null pointer checks, and boundary conditions [28, 69, 134, 197]. Prior work also explored using deep learning models for test oracle generation without the use of comments, including ATLAS [207] and TOGA [55], which are described and used as baselines in our evaluation on the test oracle generation task in Section 2.6.2. These techniques target generating/completing test oracles, but TeCo targets completing any part of the tests, including test oracles.

Tufano et al. [198] developed a test generation technique based on a BART architecture pre-trained on English and code corpora. Schafer et al. [185] proposed to use LLMs to generate tests and refine the generated tests that do not execute successfully. While they target to generate the entire test method as a whole, TeCo targets to complete one statement at a time, which allows the developer to observe and control the process of writing a test method.

Prior work also explored improving developers' productivity in testing by test recommendation: given a method under test, suggest relevant test methods from the existing test suite using a recommendation system [95, 165, 169, 227]. These techniques rely on having a set of relevant existing tests to recommend tests from,

which is usually not the case when developers are starting a new project or adding tests to a project without tests. TeCo helps developers by providing completions while they are writing tests and does not have this limitation.

4.7 Proof Mining and Automation

Müller et al. [136] exported Coq kernel trees as XML strings to translate 49 Coq projects to the OMDoc theory graph format. Rather than translating documents to an independently specified format, Roosterize produce lightweight machine-readable representations of Coq's internal data structures. Wiedijk et al. [211] collected early basic statistics on the core libraries of several proof assistants, including Coq and Isabelle/HOL. Blanchette et al. [26] mined the AFP to gather statistics such as the average number of lines of Isabelle/HOL specifications and proof scripts. However, these corpora were not used to perform learning. Heras and Komendantskaya [80–82] and Komendantskaya et al. [105] used non-deep ML models to identify patterns in Coq tactic sequences and proof kernel trees, e.g., to find structural similarities between lemmas and simplify proof development. In contrast, Roosterize captures similarity among several different representations of lemma *statements* to generate lemma names. Hellendoorn et al. [79] studied the naturalness of proofs in the Coq and HOL Light proof assistants.

Many previous projects also used ML models to directly improve proof automation in proof assistants. Tools called *hammers* [27, 48, 162] outsource proof goals to automatic solvers, and use learning to improve the process of selecting premises that may be relevant for automatically proving a goal [93, 100, 106, 107, 203]. For example, [99] perform "lemma mining" on a large corpus for the HOL Light proof assistant with all primitive inference steps from the Flyspeck project [75], in order to augment existing human-organized lemmas as premises.

Gauthier et al. [65, 66] used learning of tactics in the context of proof goals to improve success rates for automated proof techniques in the HOL4 proof assistant.

[137] used custom encodings of proof state in Isabelle/HOL for learning in order to predict suitable proof methods to apply. [90] performed learning on Ltac proof scripts and proof states to perform algebraic rewriting in Coq. [21] presented a deep learning environment for the HOL Light proof assistant. [215] used learning of Coq proof scripts, using kernel trees extracted from SerAPI similar to ROOSTERIZE does, to achieve general proof automation for Coq. [183] presented a similar approach for proof automation for Coq using SerAPI evaluated on the CompCert project.

Chapter 5: Future Work

We now present our plans for future work that can build upon our contributions as described in chapters 2 and 3.

Deployment for developers' usages. We plan to deploy TeCo and Roosterize as open-source plugins to popular IDEs (e.g., Visual Studio Code and Emacs) to simplify the usage of our tools for developers. As of writing, we have published a Visual Studio Code plugin for Roosterize (which we plan to upgrade to use cloud-hosted LLMs rather than local ML models that require GPUs), and we are working on a Visual Studio Code plugin for TeCo. There will be multiple design choices to make about the user interface, such us showing top-1 prediction as inline completion vs. showing top-k predictions in a dropbox, whether to show ML model's confidence score, whether to include an explanation of the prediction (e.g., by pointing to the extracted code semantics), etc. We plan to make these settings configurable by users, and also collect user feedbacks to guide the default settings and our future development of ML for SE tools.

Evaluation of impact on manual effort. To date, our evaluations of TeCo and Roosterize are based on comparisons with developer-written tests and lemma names using similarity metrics or human judgments, and executions of the generated tests. To better understand how much can TeCo and Roosterize reduce manual effort in testing and verification, we plan to setup user studies where the participants are asked to write *new* tests and proofs with/without using our tools. We will design the user studies based on our prior experience of setting up user studies for our TrigIt framework [141] and comment update ML model [158]. Specifically, we will measure metrics including the acceptance rate of the predictions from our tools, the time spent on writing tests and proofs, the ratio of this time compared to the time spent on writing code under test/verification, and the quality of the written tests and proofs. By analyzing the results and comparing them with our current evaluation

results, we will know if and how the improvements in the automated metrics map to reductions in manual effort.

Usage of LLMs for testing and verification. There is no conceptual difficulty in replacing ROOSTERIZE's multi-input RNN model with LLMs, or upgrading TECO's underlying model (CodeT5) to recent larger LLMs. As more recent LLMs are reported to outperform previous models in many (general) code completion and summarization tasks, we expect using LLMs will likely improve the accuracy of our tools. LLMs pre-trained on vast amount of code are shown to make less syntax errors [15], which may require us to shift the focus of using execution, e.g., from preventing compilation errors to preventing deeper runtime errors.

Usage of execution in pre-training LLMs. Although the pre-training dataset of recent LLMs include large amount of software data, the pre-training process still treats code as non-executable natural language text. We plan to explore using execution, especially the execution of tests and proofs, to improve the pre-training of LLMs. Execution can be added as new pre-training tasks, e.g., predicting the types and values of variables after executing a code snippet, which can improve the LLMs' ability in understanding and generating code with complex control and data flows. On the other hand, execution can be used to improve the quality of the pre-training dataset, e.g., by filtering out code that does not compile, and by prioritizing trustworthy software projects that has high-quality tests and proofs.

File- and project-level predictions. TECO considers one test method at a time, and ROOSTERIZE considers one lemma at a time. However, the structure of all tests or lemmas in a file/project should also follow conventions. For example, in Java, developers typically write one test class (test suite) for each class under test, and each test class contains several test methods; different test classes and test methods should not overlap in their fault detection capabilities. In the future, we plan to design ML models that make predictions to tests and proofs at file level and project level. Such models should not only suggest adding new tests/proofs for the part of

code that is not covered by existing tests/proofs, but also suggest removing redundant tests/proofs and reorganization of badly-designed tests/proofs.

Reducing manual effort on maintenance. Both TeCo and Roosterize are designed to make predictions from scratch, i.e., assuming the test method or the name of the lemma does not exist before using the ML model. In practice, code constantly evolves, and a lot of manual effort is spent on maintaining existing tests and proofs to keep them up-to-date with the code changes. We plan to design ML models that suggest edits to tests and proofs, based on our experience to support the maintenance of code comments [158, 222]. As the integration with execution is vital for the accuracy of ML models for testing and verification, we plan to utilize evolving code execution data, e.g., diff of execution data when software evolves, and the evolution of the program state during the execution.

Aligning tests and proofs to expected software behavior. We assumed that the code under test/verification is correct when generating/summarizing tests and proofs. This may not be true in practice because developers may want to generate tests and proofs to detect bugs (i.e., that should fail on the buggy version of the code), or may want to write tests and proofs before the code under test/verification (e.g., in test-driven development). To address these needs, we plan to explore designing ML models that generate tests and proofs not conditioning on the code under test/verification, but conditioning on expected software behavior, which can be extracted from software requirements, documentations, bug reports, etc.

Generalization to new programming languages and paradigms. Over time, new programming languages (e.g., Go, Python, Rust) are becoming popular, and new testing and verification paradigms (e.g., inline testing [123, 124], which we proposed for testing individual statements) are invented. ML models can help developers to establish good testing and verification standards in these new languages and paradigms. However, the ML models trained on existing tests and proofs may not generalize well to new languages and paradigms, which typically have limited data for re-training the

models. We plan to explore using LLMs with few-shot/zero-shot learning to address this limitation. Another direction that we are exploring is to develop ML models for translating and maintaining code with tests/proofs across different programming languages [223].

Improving static and dynamic analyses with ML models. Although static and dynamic analyses can help extract code semantics, they have some problems that affect the effectiveness of using them with ML models. Static analysis needs to be carefully tuned considering the precision-recall trade-off, for example, to statically infer the call graph in Java, ignoring the reflection mechanism may miss some caller-callee relationships, but considering it leads to many false positives. On the other hand, dynamic analysis can be slow when extracting too much code semantics, for example, recording each method call and their parameter values incurs excessive overhead, thus a practical technique would only record a (small) subset of method calls. The exact steps of static and dynamic analyses need to be carefully designed to extract what is needed for the ML models without causing too much accuracy loss and time overhead. This is currently designed manually with the help of domain experts in TeCo and Roosterize, but in the future we plan to explore automatically designing and tuning the static and dynamic analyses with ML models.

Chapter 6: Conclusion

Software testing and verification are essential for keeping software systems reliable and safe to use, but can require lots of manual effort. This dissertation proposes to use ML models integrated with software execution to reduce the manual effort in writing tests and proofs.

First, we presented TeCo for test completion, i.e., completing next statements when writing tests. TeCo exploits code semantics extracted from test execution results and execution context to aid ML models in reasoning about code execution. TeCo also reranks ML model's predictions by executing the predicted statements, to prioritize functionally correct predictions. Compared to existing code completion models that use only syntax-level data (including LLMs trained on massive code dataset), TeCo improves the accuracy of test completion by 29%.

Second, we presented Roosterize to suggest lemma names when developers write proofs using proof assistants such as Coq. Consistent coding conventions are important as verification projects based on proof assistants become larger, but manually enforcing the conventions can be costly. Roosterize is the first ML model for automatically learning and suggesting lemma names. Existing ML models extract and summarize information extracted from only code tokens, which is not suitable for Coq where the lemma names should exhibit semantic meanings that are not explicit in code tokens. Roosterize leverages the runtime representations of the lemma from execution, including syntax trees from the parser and elaborated terms from the kernel. Roosterize improves the accuracy of lemma naming by 39%.

With the rapid growth of new ML techniques (especially LLMs) and available sources of software testing and verification corpora, we envision that ML models integrated with execution will soon play an important role in assisting developers in testing and verification to build trustworthy software systems.

References

[1] Reynald Affeldt and Cyril Cohen. Formal foundations of 3D geometry to model robot manipulators. In *Certified Programs and Proofs*, pages 30–42, 2017. https://doi.org/10.1145/3018610.3018629.

[2] Reynald Affeldt and Jacques Garrigue. Formalization of error-correcting codes: From Hamming to modern coding theory. In *International Conference on Interactive Theorem Proving*, pages 17–33, 2015. https://doi.org/10.1007/978-3-319-22102-1_2.

[3] Reynald Affeldt, David Nowak, and Takafumi Saikawa. A hierarchy of monadic effects for program verification using equational reasoning. In *International Conference on Mathematics of Program Construction*, pages 226–254, 2019. https://doi.org/10.1007/978-3-030-33636-3_9.

[4] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. In *Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, 2020. https://doi.org/10.18653/v1/2020.acl-main.449.

[5] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, 2021. https://doi.org/10.18653/v1/2021.naacl-main.211.

[6] Nader Al Awar, Kush Jain, Christopher J. Rossbach, and Milos Gligoric. Programming and execution models for parallel bounded exhaustive testing. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 166:1–28, 2021. https://doi.org/10.1145/3485543.

[7] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. Machine learning for big code and naturalness. https://ml4code.github.io.

[8] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *International Symposium on the Foundations of Software Engineering*, pages 281–293, 2014. https://doi.org/10.1145/2635868.2635883.

[9] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *International Symposium on the Foundations of Software Engineering*, pages 38–49, 2015. https://doi.org/10.1145/2786805.2786849.

[10] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100, 2016. http://proceedings.mlr.press/v48/allamanis16.html.

[11] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4):81:3–81:37, 2018. https://doi.org/10.1145/3212695.

[12] Anaconda, Inc. Miniconda - conda documentation. https://docs.conda.io/en/latest/miniconda.html.

[13] Apache Software Foundation. Apache Lucene. https://lucene.apache.org.

[14] David Aspinall and Cezary Kaliszyk. What's in a theorem name? In *International Conference on Interactive Theorem Proving*, pages 459–465, 2016. https://doi.org/10.1007/978-3-319-43144-4_28.

[15] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le,

and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. https://doi.org/10.48550/arXiv.2108.07732.

[16] Eran Avidan and Dror G. Feitelson. Effects of variable names on comprehension: An empirical study. In *International Conference on Program Comprehension*, pages 55–65, 2017. https://doi.org/10.1109/ICPC.2017.27.

[17] Jeremy Avigad. Mathlib naming conventions. https://github.com/leanprover-community/mathlib/blob/snapshot-2019-10/docs/contribute/naming.md.

[18] Hlib Babii, Andrea Janes, and Romain Robbes. Modeling vocabulary for big code machine learning. *arXiv preprint arXiv:1904.01873*, 2019. https://doi.org/10.48550/arXiv.1904.01873.

[19] Alexander Bagnall, Samuel Merten, and Gordon Stewart. A library for algorithmic game theory in Ssreflect/Coq. *Journal of Formalized Reasoning*, 10(1): 67–95, 2017. https://doi.org/10.6092/issn.1972-5787/7235.

[20] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*, 2015. https://arxiv.org/abs/1409.0473.

[21] Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. HOList: An environment for machine learning of higher order logic theorem proving. In *International Conference on Machine Learning*, pages 454–463, 2019. http://proceedings.mlr.press/v97/bansal19a.html.

[22] Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28:321–336, 2002. https://doi.org/10.1023/A:1015761529444.

[23] Evmorfia-Iro Bartzia and Pierre-Yves Strub. A formal library for elliptic curves in the Coq proof assistant. In *International Conference on Interactive Theorem Proving*, pages 77–92, 2014. https://doi.org/10.1007/978-3-319-08970-6_6.

[24] Taylor Berg-Kirkpatrick, David Burkett, and Dan Klein. An empirical investigation of statistical significance in NLP. In *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 995–1005, 2012. https://aclanthology.org/D12-1091.

[25] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. https://doi.org/10.1007/978-3-662-07964-5.

[26] Jasmin Christian Blanchette, Maximilian Haslbeck, Daniel Matichuk, and Tobias Nipkow. Mining the archive of formal proofs. In *International Conference on Intelligent Computer Mathematics*, pages 3–17, 2015. https://doi.org/10.1007/978-3-319-20615-8_1.

[27] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1): 101–148, 2016. https://doi.org/10.6092/issn.1972-5787/4593.

[28] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In *International Symposium on Software Testing and Analysis*, pages 242–253, 2018. https://doi.org/10.1145/3213846.3213872.

[29] Arthur Blot, Pierre-Évariste Dagand, and Julia Lawall. From sets to bits in Coq. In *International Symposium on Functional and Logic Programming*, pages 12–28, 2016. https://doi.org/10.1007/978-3-319-29604-3_2.

[30] Cathal Boogerd and Leon Moonen. Evaluating the relation between coding standard violations and faults within and across software versions. In *International Working Conference on Mining Software Repositories*, pages 41–50, 2009. https://doi.org/10.1109/MSR.2009.5069479.

[31] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis*, pages 123–133, 2002. https://doi.org/10.1145/566171.566191.

[32] Fiorenza Brady. Cambridge university study states software bugs cost economy $312 billion per year. http://www.prweb.com/releases/2013/1/prweb10298185.htm.

[33] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. Technical report, Judge Business School, University of Cambridge, 2013. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0be446c155f8efcf17ef3a103188c6ae6d27d9b4.

[34] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. 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 Ziegler, Jeffrey Wu, Clemens Winter, Chris 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. In *Conference on Neural Information Processing Systems*, pages 1877–1901, 2020. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[35] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *International Symposium on the Foundations of Software Engineering*, pages 213–222, 2009. https://doi.org/10.1145/1595696.1595728.

[36] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. *Adaptable and Extensible Component Systems*, 30(19), 2002. https://citeseerx.ist.psu.edu/doc/10.1.1.117.5769.

[37] John N. Buxton and Randell Brian. *Software Engineering Techniques: Report of a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27th-31st October 1969*. Scientific Affairs Division, NATO, 1970. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF.

[38] Ahmet Celik, Sreepathi Pai, Sarfraz Khurshid, and Milos Gligoric. Bounded exhaustive test-input generation on GPUs. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 94:1–25, 2017. https://doi.org/10.1145/3133918.

[39] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash Hoare logic for certifying the FSCQ file system. In *Symposium on Operating Systems Principles*, pages 18–37, 2015. https://doi.org/10.1145/2815400.2815402.

[40] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura 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, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol,

Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. 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. *arXiv preprint arXiv:2107.03374*, 2021. https://doi.org/10.48550/arXiv.2107.03374.

[41] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming*, pages 268–279, 2000. https://doi.org/10.1145/351240.351266.

[42] Cyril Cohen. Contribution guide for the Mathematical Components library. https://github.com/math-comp/math-comp/blob/mathcomp-1.9.0/CONTRIBUTING.md.

[43] Myra B. Cohen, Joshua Snyder, and Gregg Rothermel. Testing across configurations: Implications for combinatorial testing. *Software Engineering Notes*, 31(6):1–9, 2006. https://doi.org/10.1145/1218776.1218785.

[44] CompCert Team. CompCert - publications. https://compcert.org/publi.html.

[45] Coq Development Team. The Coq proof assistant, version 8.10.0. https://doi.org/10.5281/zenodo.3476303.

[46] Coq Development Team. Building a Coq project. https://coq.inria.fr/distrib/V8.10.2/refman/practical-tools/utilities.html.

[47] Coq Development Team. The Gallina specification language. https://coq.inria.fr/distrib/V8.10.2/refman/language/gallina-specification-language.html.

[48] Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61(1):423–453, 2018. https://doi.org/10.1007/s10817-018-9458-4.

[49] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *International Symposium on Software Reliability Engineering*, pages 201–211, 2014. https://doi.org/10.1109/ISSRE.2014.11.

[50] Ermira Daka, José Miguel Rojas, and Gordon Fraser. Generating unit tests with descriptive names or: Would you name your children Thing1 and Thing2? In *International Symposium on Software Testing and Analysis*, pages 57–67, 2017. https://doi.org/10.1145/3092703.3092727.

[51] Leonardo de Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. Elaboration in dependent type theory. *arXiv preprint arXiv:1505.04324*, 2015. https://doi.org/10.48550/arXiv.1505.04324.

[52] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388, 2015. https://doi.org/10.1007/978-3-319-21401-6_26.

[53] David Delahaye. A tactic language for the system Coq. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 85–95, 2000. https://doi.org/10.1007/3-540-44404-1_7.

[54] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. https://doi.org/10.48550/arXiv.1810.04805.

[55] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. TOGA: A neural method for test oracle generation. In *International Conference*

on Software Engineering, pages 2130–2141, 2022. https://doi.org/10.1145/3510003.3510141.

[56] Christian Doczkal and Joachim Bard. Completeness and decidability of converse PDL in the constructive type theory of Coq. In *Certified Programs and Proofs*, pages 42–52, 2018. https://doi.org/10.1145/3167088.

[57] Christian Doczkal and Gert Smolka. Regular language representations in the constructive type theory of Coq. *Journal of Automated Reasoning*, 61:521–553, 2018. https://doi.org/10.1007/s10817-018-9460-x.

[58] Christian Doczkal, Jan-Oliver Kaiser, and Gert Smolka. Regular language representations in Coq. https://github.com/coq-community/reglang.

[59] Manuel Eberl, Gerwin Klein, Tobias Nipkow, Larry Paulson, and René Thiemann. Archive of formal proofs. https://www.isa-afp.org.

[60] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP*, pages 1536–1547, 2020. https://doi.org/10.18653/v1/2020.findings-emnlp.139.

[61] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *International Conference on Software Engineering*, pages 416–419, 2011. https://doi.org/10.1145/2025113.2025179.

[62] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A generative model for code infilling and synthesis. In *International Conference on Learning Representations*, 2023. https://openreview.net/forum?id=hQwb-lbM6EL.

[63] Emilio Jesús Gallego Arias. SerAPI: Machine-friendly, data-centric serialization for Coq. Technical report, MINES ParisTech, 2016. https://hal-mines-paristech.archives-ouvertes.fr/hal-01384408.

[64] Sa Gao, Chunyang Chen, Zhenchang Xing, Yukun Ma, Wen Song, and Shang-Wei Lin. A neural model for method name generation from functional description. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 414–421, 2019. https://doi.org/10.1109/SANER.2019.8667994.

[65] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. TacticToe: Learning to reason with HOL4 tactics. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 46, pages 125–143, 2017. https://doi.org/10.29007/ntlb.

[66] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. TacticToe: Learning to prove with tactics. *Journal of Automated Reasoning*, 65:257–286, 2021. https://doi.org/10.1007/s10817-020-09580-x.

[67] GitHub, Inc. GitHub Copilot: Your AI pair programmer. https://github.com/features/copilot.

[68] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *International Conference on Software Engineering*, pages 225–234, 2010. https://doi.org/10.1145/1806799.1806835.

[69] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. Automatic generation of oracles for exceptional behaviors. In *International Symposium on Software Testing and Analysis*, pages 213–224, 2016. https://doi.org/10.1145/2931037.2931061.

[70] Georges Gonthier. Formal proof—the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008. http://www.ams.org/notices/200811/tx081101382p.pdf.

[71] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pages 163–179, 2013. https://doi.org/10.1007/978-3-642-39634-2_14.

[72] Google. google-java-format: Reformats Java source code to comply with Google Java style. https://github.com/google/google-java-format.

[73] Google. Bard. https://bard.google.com.

[74] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*, 2021. https://openreview.net/forum?id=jLoC4ez43PZ.

[75] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, Quang Truong Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, Thi Hoai An Ta, Nam Trung Tran, Thi Diep Trieu, Josef Urban, Ky Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5:e2, 2017. https://doi.org/10.1017/fmp.2017.1.

[76] Sangmok Han, David R. Wallace, and Robert C. Miller. Code completion from abbreviated input. In *Automated Software Engineering*, pages 332–343, 2009. https://doi.org/10.1109/ASE.2009.64.

[77] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *Transactions on Software Engineering*, 36(2):226–247, 2009. https://doi.org/10.1109/TSE.2009.71.

[78] Jingxuan He, Cheng-Chun Lee, Veselin Raychev, and Martin Vechev. Learning to find naming issues with big code and small supervision. In *Conference on Programming Language Design and Implementation*, pages 296–311, 2021. https://doi.org/10.1145/3453483.3454045.

[79] Vincent J. Hellendoorn, Premkumar T. Devanbu, and Mohammad Amin Alipour. On the naturalness of proofs. In *International Symposium on the Foundations of Software Engineering, New Ideas and Emerging Results*, pages 724–728, 2018. https://doi.org/10.1145/3236024.3264832.

[80] Jónathan Heras and Ekaterina Komendantskaya. ML4PG in computer algebra verification. In *International Conference on Intelligent Computer Mathematics*, pages 354–358, 2013. https://doi.org/10.1007/978-3-642-39320-4_28.

[81] Jónathan Heras and Ekaterina Komendantskaya. Proof pattern search in Coq/Ssreflect. *arXiv preprint arXiv:1402.0081*, 2014. https://doi.org/10.48550/arXiv.1402.0081.

[82] Jónathan Heras and Ekaterina Komendantskaya. Recycling proof patterns in Coq: Case studies. *Mathematics in Computer Science*, 8(1):99–116, 2014. https://doi.org/10.1007/s11786-014-0173-1.

[83] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *International Conference on Soft-*

ware Engineering, pages 837–847, 2012. https://doi.org/10.1109/ICSE.2012.6227135.

[84] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012. https://doi.org/10.48550/arXiv.1207.0580.

[85] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. https://doi.org/10.1162/neco.1997.9.8.1735.

[86] Josie Holmes, Iftekhar Ahmed, Caius Brindescu, Rahul Gopinath, He Zhang, and Alex Groce. Using relative lines of code to guide automated test generation for Python. *Transactions on Software Engineering and Methodology*, 29(4):1–38, 2020. https://doi.org/10.1145/3408896.

[87] HoTT Team. HoTT conventions and style guide. https://github.com/HoTT/HoTT/blob/V8.10/STYLE.md.

[88] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *International Conference on Program Comprehension*, pages 200–210, 2018. https://doi.org/10.1145/3196321.3196334.

[89] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, 25:1–39, 2019. https://doi.org/10.1007/s10664-019-09730-9.

[90] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. GamePad: A learning environment for theorem proving. In *International Conference on Learning Representations*, 2019. https://openreview.net/forum?id=r1xwKoR9Y7.

[91] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019. https://doi.org/10.48550/arXiv.1909.09436.

[92] Iris Team. Iris style guide. https://gitlab.mpi-sws.org/iris/iris/blob/iris-3.2.0/StyleGuide.md.

[93] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Een, Francois Chollet, and Josef Urban. DeepMath - deep sequence models for premise selection. In *Conference on Neural Information Processing Systems*, pages 2235–2243, 2016. http://papers.nips.cc/paper/6280-deepmath-deep-sequence-models-for-premise-selection.pdf.

[94] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Annual Meeting of the Association for Computational Linguistics*, pages 2073–2083, 2016. https://doi.org/10.18653/v1/P16-1195.

[95] Werner Janjic and Colin Atkinson. Utilizing software reuse experience for automated test recommendation. In *International Workshop on Automation of Software Test*, pages 100–106, 2013. https://doi.org/10.1109/IWAST.2013.6595799.

[96] JavaParser Team. JavaParser: Java 1-17 parser and abstract syntax tree for Java with advanced analysis functionalities. https://github.com/javaparser/javaparser.

[97] JUnit4 Team. JUnit4 homepage. https://junit.org/junit4.

[98] JUnit5 Team. JUnit5 homepage. https://junit.org/junit5.

[99] Cezary Kaliszyk and Josef Urban. Learning-assisted theorem proving with millions of lemmas. *Journal of Symbolic Computation*, 69:109–128, 2015. https://doi.org/10.1016/j.jsc.2014.09.032.

[100] Cezary Kaliszyk, Lionel Mamane, and Josef Urban. Machine learning of Coq proof guidance: First experiments. In *International Symposium on Symbolic Computation in Software Science*, pages 30:27–34, 2014. https://doi.org/10.29007/lmmg.

[101] Nitish Shirish Keskar, Bryan McCann, Lav R Varshney, Caiming Xiong, and Richard Socher. CTRL: A conditional transformer language model for controllable generation. *arXiv preprint arXiv:1909.05858*, 2019. https://doi.org/10.48550/arXiv.1909.05858.

[102] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015. https://arxiv.org/abs/1412.6980.

[103] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander Rush. OpenNMT: Open-source toolkit for neural machine translation. In *Annual Meeting of the Association for Computational Linguistics, System Demonstrations*, pages 67–72, 2017. https://doi.org/10.18653/v1/P17-4012.

[104] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In *Conference on Neural Information Processing Systems*, 2022. https://openreview.net/forum?id=e2TBb5y0yFf.

[105] Ekaterina Komendantskaya, Jónathan Heras, and Gudmund Grov. Machine learning in Proof General: Interfacing interfaces. In *International Workshop On User Interfaces for Theorem Provers*, pages 15–41, 2013. https://doi.org/10.4204/EPTCS.118.2.

[106] Daniel Kühlwein, Twan van Laarhoven, Evgeni Tsivtsivadze, Josef Urban, and Tom Heskes. Overview and evaluation of premise selection techniques for large theory mathematics. In *International Joint Conference on Automated Reasoning*, pages 378–392, 2012. https://doi.org/10.1007/978-3-642-31365-3_30.

[107] Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. MaSh: Machine learning for Sledgehammer. In *International Conference on Interactive Theorem Proving*, pages 35–50, 2013. https://doi.org/10.1007/978-3-642-39634-2_6.

[108] Ivan Kuraj, Viktor Kuncak, and Daniel Jackson. Programming with enumerable sets of structures. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 37–56, 2015. https://doi.org/10.1145/2814270.2814323.

[109] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. In *International Conference on Software Engineering*, pages 795–806, 2019. https://doi.org/10.1109/ICSE.2019.00087.

[110] Jaeseong Lee, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. On the naturalness of hardware descriptions. In *International Symposium on the Foundations of Software Engineering*, pages 530–542, 2020. https://doi.org/10.1145/3368089.3409692.

[111] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *International Conference on Software Engineering*, 2023. To apper, preprint: https://www.carolemieux.com/codamosa_icse23.pdf.

[112] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. https://doi.org/10.1145/1538788.1538814.

[113] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, 2020. https://doi.org/10.18653/v1/2020.acl-main.703.

[114] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. Code completion with neural attention and pointer networks. In *International Joint Conference on Artificial Intelligence*, pages 4159–4165, 2018. https://doi.org/10.24963/ijcai.2018/578.

[115] Jiwei Li, Michel Galley, Chris Brockett, Jianfeng Gao, and Bill Dolan. A diversity-promoting objective function for neural conversation models. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 110–119, 2016. https://doi.org/10.18653/v1/n16-1014.

[116] 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 Vil-

legas, 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! *arXiv preprint arXiv:2305.06161*, 2023. https://doi.org/10.48550/arXiv.2305.06161.

[117] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. https://doi.org/10.1126/science.abq1158.

[118] Yuding Liang and Kenny Qili Zhu. Automatic generation of text descriptive comments for code blocks. In *AAAI Conference on Artificial Intelligence*, pages 5229–5236, 2018. https://doi.org/10.1609/aaai.v32i1.11963.

[119] Chin-Yew Lin and Franz Josef Och. ORANGE: A method for evaluating automatic evaluation metrics for machine translation. In *International Conference on Computational Linguistics*, pages 501–507, 2004. https://aclanthology.org/C04-1072.

[120] Chin-Yew Lin and Franz Josef Och. Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics. In *Annual Meeting of the Association for Computational Linguistics*, pages 605–612, 2004. https://doi.org/10.3115/1218955.1219032.

[121] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. NL2Bash: A corpus and semantic parser for natural language interface to the Linux operating system. In *International Conference on Language Resources and Evaluation*, 2018. https://aclanthology.org/L18-1491.

[122] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In *International Conference on Software Engineering*, pages 1–12, 2019. https://doi.org/10.1109/ICSE.2019.00019.

[123] Yu Liu, Pengyu Nie, Owolabi Legunsen, and Milos Gligoric. Inline tests. In *Automated Software Engineering*, pages 57:1–13, 2022. https://doi.org/10.1145/3551349.3556952.

[124] Yu Liu, Zachary Thurston, Alan Han, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. pytest-inline: An inline testing tool for Python. In *International Conference on Software Engineering, Demonstrations*, 2023. To appear, preprint: https://arxiv.org/abs/2305.13486.

[125] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Conference on Neural Information Processing Systems, Systems Datasets and Benchmarks*, 2021. https://openreview.net/forum?id=6lE4dQXaUcb.

[126] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. ReACC: A retrieval-augmented code completion framework. In *Annual Meeting of the Association for Computational Linguistics*, pages 6227–6240, 2022. https://doi.org/10.18653/v1/2022.acl-long.431.

[127] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Empirical Methods in Natural Language Processing*, pages 1412–1421, 2015. https://doi.org/10.18653/v1/d15-1166.

[128] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, 2022. https://doi.org/10.5281/zenodo.7118596.

[129] Yaniv Markovski. Understanding Codex training data and outputs. https://help.openai.com/en/articles/5480054-understanding-codex-training-data-and-outputs.

[130] Mathematical Components Team. Missing lemmas in seq. https://github.com/math-comp/math-comp/pull/41.

[131] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of ACM*, 3(4):184–195, 1960. https://doi.org/10.1145/367177.367199.

[132] Richard J. Miara, Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman. Program indentation and comprehensibility. *Communications of ACM*, 26(11): 861–867, 1983. https://doi.org/10.1145/182.358437.

[133] Mike Mirzayanov. Codeforces. https://codeforces.com.

[134] Manish Motwani and Yuriy Brun. Automatically generating precise oracles from structured natural language specifications. In *International Conference on Software Engineering*, pages 188–199, 2019. https://doi.org/10.1109/ICSE.2019.00035.

[135] MSRA and ISCAS. NL2Code. https://nl2code.github.io.

[136] Dennis Müller, Florian Rabe, and Claudio Sacerdoti Coen. The Coq library as a theory graph. In *International Conference on Intelligent Computer Mathematics*, pages 171–186, 2019. https://doi.org/10.1007/978-3-030-23250-4_12.

[137] Yutaka Nagashima and Yilun He. PaMpeR: Proof method recommendation system for Isabelle/HOL. In *Automated Software Engineering*, pages 362–372, 2018. https://doi.org/10.1145/3238147.3238210.

[138] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, German Delbianco, and Anton Trunov. The PCM library. https://github.com/imdea-software/fcsl-pcm.

[139] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *International Conference on Software Engineering*, pages 69–79, 2012. https://doi.org/10.1109/ICSE.2012.6227205.

[140] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. Suggesting natural method names to check name consistencies. In *International Conference on Software Engineering*, pages 1372–1384, 2020. https://doi.org/10.1145/3377811.3380926.

[141] Pengyu Nie, Rishabh Rai, Junyi Jessy Li, Sarfraz Khurshid, Raymond J. Mooney, and Milos Gligoric. A framework for writing trigger-action todo comments in executable format. In *International Symposium on the Foundations of Software Engineering*, pages 385–396, 2019. https://doi.org/10.1145/3338906.3338965.

[142] Pengyu Nie, Karl Palmskog, Junyi Jessy Li, and Milos Gligoric. Learning to format Coq code using language models. In *The Coq Workshop*, 2020. https://arxiv.org/abs/2006.16743.

[143] Pengyu Nie, Karl Palmskog, Junyi Jessy Li, and Milos Gligoric. Deep generation of Coq lemma names using elaborated terms. In *International Joint Conference on Automated Reasoning*, pages 97–118, 2020. https://doi.org/10.1007/978-3-030-51054-1_6.

[144] Pengyu Nie, Karl Palmskog, Junyi Jessy Li, and Milos Gligoric. Roosterize: Suggesting lemma names for Coq verification projects using deep learning. In *International Conference on Software Engineering, Demonstrations*, pages 21–24, 2021. https://doi.org/10.1109/ICSE-Companion52605.2021.00026.

[145] Pengyu Nie, Jiyang Zhang, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. Impact of evaluation methodologies on code summarization. In *Annual Meeting of the Association for Computational Linguistics*, pages 4936–4960, 2022. https://doi.org/10.18653/v1/2022.acl-long.339.

[146] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. Learning deep semantics for test completion. In *International Conference on Software Engineering*, 2023. To appear, preprint: https://arxiv.org/abs/2302.10166.

[147] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. CodeGen2: Lessons for training LLMs on programming and natural languages. In *Deep Learning for Code Workshop*, 2023. https://dl4c.github.io/assets/pdf/papers/29.pdf.

[148] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An open large language model for code with multi-turn program synthesis. In *International Conference on Learning Representations*, 2023. https://openreview.net/forum?id=iaYcJKpY2B_.

[149] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021. https://doi.org/10.48550/arXiv.2112.00114.

[150] OCamlverse. Metaprogramming and PPX. http://ocamlverse.net/content/metaprogramming.html.

[151] Naoto Ogura, Shinsuke Matsumoto, Hideaki Hata, and Shinji Kusumoto. Bring your own coding style. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 527–531, 2018. https://doi.org/10.1109/SANER.2018.8330253.

[152] OPAM Team. OCaml package manager. https://opam.ocaml.org.

[153] OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023. https://doi.org/10.48550/arXiv.2303.08774.

[154] Oracle and/or its Affiliates. Lambda expression. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

[155] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Conference on Neural Information Processing Systems*, pages 27730–27744, 2022. https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf.

[156] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *International Conference on*

Software Engineering, pages 75–84, 2007. https://doi.org/10.1109/ICSE.2007.37.

[157] Sheena Panthaplackel, Milos Gligoric, Raymond J. Mooney, and Junyi Jessy Li. Associating natural language comment and source code entities. In *AAAI Conference on Artificial Intelligence*, pages 8592–8599, 2020. https://doi.org/10.1609/aaai.v34i05.6382.

[158] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond J. Mooney. Learning to update natural language comments based on code changes. In *Annual Meeting of the Association for Computational Linguistics*, pages 1853–1868, 2020. https://doi.org/10.18653/v1/2020.acl-main.168.

[159] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J. Mooney. Deep just-in-time inconsistency detection between comments and source code. In *AAAI Conference on Artificial Intelligence*, pages 427–435, 2021. https://doi.org/10.1609/aaai.v35i1.16119.

[160] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Annual Meeting of the Association for Computational Linguistics*, pages 311–318, 2002. https://doi.org/10.3115/1073083.1073135.

[161] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *Autodiff Workshop*, 2017. https://openreview.net/forum?id=BJJsrmfCZ.

[162] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In *International Workshop on the Implementation of Logics*, pages 2:1–11, 2012. https://doi.org/10.29007/36dt.

[163] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12: 2825–2830, 2011. https://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf.

[164] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Learning approximate execution semantics from traces for binary function similarity. *Transactions on Software Engineering*, 49(4):2776–2790, 2022. https://doi.org/10.1109/TSE.2022.3231621.

[165] Raphael Pham, Yauheni Stoliar, and Kurt Schneider. Automatically recommending test code examples to inexperienced developers. In *International Symposium on the Foundations of Software Engineering, New Ideas and Emerging Results*, pages 890–893, 2015. https://doi.org/10.1145/2786805.2803202.

[166] Xinyu Pi, Qian Liu, Bei Chen, Morteza Ziyadi, Zeqi Lin, Qiang Fu, Yan Gao, Jian-Guang Lou, and Weizhu Chen. Reasoning like program executors. In *Empirical Methods in Natural Language Processing*, pages 761–779, 2022. https://aclanthology.org/2022.emnlp-main.48.

[167] George Pîrlea and Ilya Sergey. Mechanising blockchain consensus. In *Certified Programs and Proofs*, pages 78–90, 2018. https://doi.org/10.1145/3167086.

[168] Sebastian Proksch, Johannes Lerch, and Mira Mezini. Intelligent code completion with Bayesian networks. *Transactions on Software Engineering and Methodology*, 25(1):1–31, 2015. https://doi.org/10.1145/2744200.

[169] Ruixiang Qian, Yuan Zhao, Duo Men, Yang Feng, Qingkai Shi, Yong Huang, and Zhenyu Chen. Test recommendation system based on slicing coverage filtering. In *International Symposium on Software Testing and Analysis, Tool Demonstrations*, pages 573–576, 2020. https://doi.org/10.1145/3395363.3404370.

[170] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[171] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21:1–67, 2020. https://www.jmlr.org/papers/volume21/20-074/20-074.pdf.

[172] Musfiqur Rahman, Dharani Palani, and Peter Rigby. Natural software revisited. In *International Conference on Software Engineering*, pages 37–48, 2019. https://doi.org/10.1109/ICSE.2019.00022.

[173] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Conference on Programming Language Design and Implementation*, pages 419–428, 2014. https://doi.org/10.1145/2594291.2594321.

[174] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Symposium on Principles of Programming Languages*, pages 111–124, 2015. https://doi.org/10.1145/2676726.2677009.

[175] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. Quickly generating diverse valid test inputs with reinforcement learning. In *Interna-*

tional Conference on Software Engineering, pages 1410–1421, 2020. https://doi.org/10.1145/3377811.3380399.

[176] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: A method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020. https://doi.org/10.48550/arXiv.2009.10297.

[177] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at large: A survey of engineering of formally verified software. *Foundations and Trends in Programming Languages*, 5(2-3):102–281, 2019. https://doi.org/10.1561/2500000045.

[178] Romain Robbes and Michele Lanza. How program history can improve code completion. In *Automated Software Engineering*, pages 317–326, 2008. https://doi.org/10.1109/ASE.2008.42.

[179] Romain Robbes and Michele Lanza. Improving code completion with program history. *Automated Software Engineering*, 17(2):181–212, 2010. https://doi.org/10.1007/s10515-010-0064-x.

[180] Stephen E. Robertson, Steve Walker, and Micheline Beaulieu. Experimentation as a way of life: Okapi at TREC. *Information Processing & Management*, 36 (1):95–108, 2000. https://doi.org/10.1016/S0306-4573(99)00046-1.

[181] Brian Robinson, Michael D Ernst, Jeff H Perkins, Vinay Augustine, and Nuo Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *Automated Software Engineering*, pages 23–32, 2011. https://doi.org/10.1109/ASE.2011.6100059.

[182] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. https://doi.org/10.1038/323533a0.

[183] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. Generating correctness proofs with neural networks. In *International Workshop on Machine Learning and Programming Languages*, pages 1–10, 2020. https://doi.org/10.1145/3394450.3397466.

[184] Rylan Schaeffer, Brando Miranda, and Sanmi Koyejo. Are emergent abilities of large language models a mirage? *arXiv preprint arXiv:2304.15004*, 2023. https://doi.org/10.48550/arXiv.2304.15004.

[185] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527*, 2023. https://doi.org/10.48550/arXiv.2302.06527.

[186] Abigail See, Peter J. Liu, and Christopher D. Manning. Get to the point: Summarization with pointer-generator networks. In *Annual Meeting of the Association for Computational Linguistics*, pages 1073–1083, 2017. https://doi.org/10.18653/v1/P17-1099.

[187] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Annual Meeting of the Association for Computational Linguistics*, pages 1715–1725, 2016. https://doi.org/10.18653/v1/P16-1162.

[188] Iulian V. Serban, Alessandro Sordoni, Yoshua Bengio, Aaron Courville, and Joelle Pineau. Building end-to-end dialogue systems using generative hierarchical neural network models. In *AAAI Conference on Artificial Intelligence*, pages 3776–3783, 2016. https://doi.org/10.1609/aaai.v30i1.9883.

[189] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *Conference on Programming Language Design and Implementation*, pages 77–87, 2015. https://doi.org/10.1145/2737924.2737964.

[190] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. In *Symposium on Principles of Programming Languages*, pages 28:1–30, 2017. https://doi.org/10.1145/3158116.

[191] Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. Natural language to code translation with execution. In *Empirical Methods in Natural Language Processing*, pages 3533–3546, 2022. https://aclanthology.org/2022.emnlp-main.231.

[192] Ben Shneiderman and Don McKay. Experimental investigations of computer program debugging and modification. In *Human Factors Society Annual Meeting*, volume 20, pages 557–563, 1976. https://doi.org/10.1177/154193127602002401.

[193] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. pages 3104–3112, 2014. https://proceedings.neurips.cc/paper_files/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf.

[194] Jun Suzuki and Masaaki Nagata. Cutting-off redundant repeating generations for neural abstractive summarization. In *Conference of the European Chapter of the Association for Computational Linguistics*, pages 291–297, 2017. https://doi.org/10.18653/v1/e17-2047.

[195] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: AI-assisted code completion system. In *International Conference on Knowledge Discovery and Data Mining*, pages 2727–2735, 2019. https://doi.org/10.1145/3292500.3330699.

[196] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. IntelliCode compose: Code generation using transformer. In *International Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020. https://doi.org/10.1145/3368089.3417058.

[197] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *International Conference on Software Testing, Verification, and Validation*, pages 260–269, 2012. https://doi.org/10.1109/ICST.2012.106.

[198] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers. *arXiv preprint arXiv:2009.05617*, 2020. https://doi.org/10.48550/arXiv.2009.05617.

[199] Inigo Jauregi Unanue, Ehsan Zare Borzeshi, and Massimo Piccardi. A shared attention mechanism for interpretation of neural automatic post-editing systems. In *Workshop on Neural Machine Translation and Generation*, pages 11–17, 2018. https://doi.org/10.18653/v1/w18-2702.

[200] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Conference on Neural Information Processing Systems*, pages 6000–6010, 2017. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[201] Ke Wang and Zhendong Su. Blended, precise semantic program embeddings. In *Conference on Programming Language Design and Implementation*, pages 121–134, 2020. https://doi.org/10.1145/3385412.3385999.

[202] Ke Wang, Zhendong Su, and Rishabh Singh. Dynamic neural program embeddings for program repair. In *International Conference on Learning Representations*, 2018. https://openreview.net/forum?id=BJuWrGW0Z.

[203] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. In *Conference on Neural Information Processing Systems*, pages 2786–2796, 2017. http://papers.nips.cc/paper/6871-premise-selection-for-theorem-proving-by-deep-graph-embedding.pdf.

[204] Simin Wang, Liguo Huang, Amiao Gao, Jidong Ge, Tengfei Zhang, Haitao Feng, Ishna Satyarth, Ming Li, He Zhang, and Vincent Ng. Machine/deep learning for software engineering: A systematic literature review. *Transactions on Software Engineering*, 49(3):1188–1231, 2022. https://doi.org/10.1109/TSE.2022.3173346.

[205] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *International Conference on Software Engineering*, pages 461–470, 2008. https://doi.org/10.1145/1368088.1368151.

[206] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Empirical Methods in Natural Language Processing*, pages 8696–8708, 2021. https://doi.org/10.18653/v1/2021.emnlp-main.685.

[207] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. In *International Conference on Software Engineering*, pages 1398–1409, 2020. https://doi.org/10.1145/3377811.3380429.

[208] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. A systematic literature review on the use of deep learning in software engineering research. *Transactions on Software Engineering and Methodology*, 31(2):1–58, 2022. https://doi.org/10.1145/3485275.

[209] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. Code generation as a dual task of code summarization. In *Conference on Neural Information Processing Systems*, pages 6563–6573, 2019. https://proceedings.neurips.cc/paper_files/paper/2019/file/e52ad5c9f751f599492b4f087ed7ecfc-Paper.pdf.

[210] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022. https://openreview.net/forum?id=yzkSU5zdwD.

[211] Freek Wiedijk. Statistics on digital libraries of mathematics. *Studies in Logic, Grammar and Rhetoric*, 18(31):137–151, 2009. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d61540ec721b2871256348e0ad22a7e310428aa3.

[212] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989. https://doi.org/10.1162/neco.1989.1.2.270.

[213] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the Raft consensus protocol. In *Certified Programs and Proofs*, pages 154–165, 2016. https://doi.org/10.1145/2854065.2854081.

[214] Sihan Xu, Sen Zhang, Weijing Wang, Xinya Cao, Chenkai Guo, and Jing Xu. Method name suggestion with hierarchical attention networks. In *Workshop on Partial Evaluation and Program Manipulation*, pages 10–21, 2019. https://doi.org/10.1145/3294032.3294079.

[215] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning*, pages 6984–6994, 2019. http://proceedings.mlr.press/v97/yang19a.html.

[216] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie Van Deursen. Mining software repositories to study co-evolution of production & test code.

In *International Conference on Software Testing, Verification, and Validation*, pages 220–229, 2008. https://doi.org/10.1109/ICST.2008.47.

[217] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. Large language models meet NL2Code: A survey. In *Annual Meeting of the Association for Computational Linguistics*, 2023. To appear, preprint: https://arxiv.org/pdf/2212.09420.pdf.

[218] Zhiqiang Zang, Nathaniel Wiatrek, Milos Gligoric, and August Shi. Compiler testing using template Java programs. In *Automated Software Engineering*, pages 23:1–13, 2022. https://doi.org/10.1145/3551349.3556958.

[219] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA + Saarland University, 2019. https://publications.cispa.saarland/3120.

[220] Benwen Zhang, Emily Hill, and James Clause. Towards automatically generating descriptive names for unit tests. In *Automated Software Engineering*, pages 625–636, 2016. https://doi.org/10.1145/2970276.2970342.

[221] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. Leveraging class hierarchy for code comprehension. In *Workshop on Computer-Assisted Programming*, 2020. https://openreview.net/forum?id=WHjjpMNZFoD.

[222] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. CoditT5: Pretraining for source code and natural language editing. In *Automated Software Engineering*, pages 22:1–12, 2022. https://doi.org/10.1145/3551349.3556955.

[223] Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. Multilingual code co-evolution using large language models. In *International Symposium on the Foundations of Software Engineering*, 2023. To appear.

[224] Shu Zhang, Dequan Zheng, Xinchen Hu, and Ming Yang. Bidirectional long short-term memory networks for relation classification. In *Pacific Asia Conference on Language, Information and Computation*, pages 207–212, 2015. https://doi.org/10.18653/v1/p16-2034.

[225] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. CodeGeeX: A pre-trained model for code generation with multilingual evaluations on HumanEval-X. *arXiv preprint arXiv:2303.17568*, 2023. https://doi.org/10.48550/arXiv.2303.17568.

[226] Wen Zhou, Seohyun Kim, Vijayaraghavan Murali, and Gareth Ari Aye. Improving code autocompletion with transfer learning. In *International Conference on Software Engineering, Software Engineering in Practice*, pages 161–162, 2022. https://doi.org/10.1145/3510457.3513061.

[227] Chenqian Zhu, Weisong Sun, Qin Liu, Yangyang Yuan, Chunrong Fang, and Yong Huang. HomoTR: Online test recommendation system based on homologous code matching. In *Automated Software Engineering, Tool Demonstrations*, 2020. https://doi.org/10.1145/3324884.3415296.