AUTOMATING THE FORMAL VERIFICATION OF SOFTWARE

A Dissertation Presented

by

EMILY FIRST

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2023

Robert and Donna Manning College of
Information and Computer Sciences

AUTOMATING THE FORMAL VERIFICATION OF SOFTWARE

A Dissertation Presented

by

EMILY FIRST

Approved as to style and content by:

DEDICATION

To my husband, Subbu.

ACKNOWLEDGMENTS

Thank you to my advisor and chair of my dissertation committee, Yuriy Brun. Thank you for your support and guidance and for helping me overcome the many challenges along the way. You've also helped me to believe in myself and my abilities, and for that I am truly grateful.

Thank you to George Avrunin, Arjun Guha, and Talia Ringer for being part of my dissertation committee and for providing invaluable feedback and support over the past few years.

Thank you to the folks at the N2Formal team at Google for hosting me for part of the past year and for helping me with what has become the final chapter of my dissertation. Thank you, especially, to Markus N. Rabe for being a great manager and spending a lot of time helping to finish the work.

Thank you to the folks in LASER lab. Thank you to Alex Sanchez-Stern for all of your guidance, help, and friendship. Thank you to Manish, Heather, Yixue, Zhanna, and Ishita for your support and friendship. Thank you to George, Lee, Laurie, Marius, and Alicia for all of your feedback in LASER meetings over the past few years.

Thank you to my husband, Subbu, for pushing me to finish this program and agreeing to marry me. Thank you to my family for all of your support. Last, but certainly not least, thank you to my parents. You have always been there for me and provided me with so much love, support, and opportunities. I cannot thank you enough.

ABSTRACT

AUTOMATING THE FORMAL VERIFICATION OF SOFTWARE

MAY 2023

EMILY FIRST

B.S., HARVEY MUDD COLLEGE

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Yuriy Brun

Formally verified correctness is one of the most desirable properties of software systems. Despite great progress made toward verification via interactive proof assistants, such as Coq and Isabelle/HOL, such verification remains one of the most effort-intensive (and often prohibitively difficult) software development activities. Recent work has created tools that automatically synthesize proofs either through reasoning using precomputed facts or using machine learning to model proofs and then perform biased search through the proof space. However, models in existing tools fail to capture the richness present in proofs, such as the information the programmer has access to when writing proofs and the natural language contained within variable names. Furthermore, these prior models do not make use of variations in the learning process and advances in large language models.

In this dissertation, I develop tools to improve proof synthesis and to enable fully automating more verification. I first present TacTok, a proof-synthesis tool that models proofs using both the partial proof written thus far and the semantics of the proof state. I then present Diva, a proof-synthesis tool that controls the learning process to produce a diverse set of models and, due to the unique nature of proof synthesis (the existence of the theorem prover, an oracle that infallibly judges a proof's correctness), efficiently combines these models to improve the overall proving power. I then present Passport, a proof-synthesis tool that systematically explores different ways of encoding identifiers in proofs to improve synthesis. Finally, I present Baldur, a proof-synthesis tool that uses transformer-based pretrained large language models fine-tuned on proofs to generate and repair whole proofs at once, rather than one step at a time.

This dissertation contributes new ideas for improving automated proof synthesis and empirically demonstrates that the improvement is significant on large benchmarks consisting of open-source software projects.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Building provably correct systems is critical in high-stakes domains, such as aerospace engineering and software for medical devices. However, most industrial verification tools either aim to simplify the verification process by sacrificing soundness [19] or significantly restrict the programming language in which the system is written [132]. A promising method for building correct software has been to use programming languages that are designed to inherently support program verification, such as interactive theorem provers (ITPs), including Coq [199], Isabelle/HOL [148], and Agda [209].

ITPs have had significant impact on industry. For example, Airbus France uses the Coq-verified CompCert C compiler [121] to ensure safety and improve performance of its aircraft [191]. Chrome and Android both use cryptographic code formally verified in Coq to secure communication [56], while Mozilla has its own verified cryptographic library for Firefox, improving performance [95]. Multiple companies have been successful in using proof assistants to provide formal verification services, including BedRock Systems, who builds formally verified solutions for the healthcare, infrastructure, and financial domains [17], Certora, who formally verifies smart contracts [34], and Galois, Inc., who verifies compiler correctness and hardware design [66]. Meanwhile Amazon successfully applies formal verification to cloud security problems in Amazon Web Services, providing tools for users to detect entire classes of misconfigurations that can potentially expose vulnerable data [10].

With ITPs, the user (a programmer) specifies a theorem about a property of the software and writes a *proof script*, a series of annotated proof tactics, that the interactive theorem prover uses to attempt to construct a proof of the theorem. Still, even with the help of an ITP, the effort required to write proof scripts is often prohibitive. The Coq proof of the C compiler has more than three times as many lines as the compiler code itself and took three person years of work [121]. Meanwhile, it took 11 person years to write the proof script to verify a microkernel in Isabelle/HOL [142]. As a general rule, because of the expense of verification, nearly all software that companies ship is unverified.

However, some formal verification can be fully automated by synthesizing either the underlying proofs or the guiding proof scripts. A series of tools called *hammers* (e.g., CoqHammer [49] and SledgeHammer [156]) use a set of precomputed mathematical facts to attempt to "hammer" out a proof. Evaluated on the CoqGym benchmark [223], CoqHammer can automatically prove 26.6% of theorems found in open-source Coq projects, while Sledgehammer proves 25.7% of the PISA benchmark [100] for Isabelle/HOL. But hammers are restricted by their precomputed facts and cannot reason about proof approaches such as induction, greatly limiting their power. To overcome these limitations, researchers have used machine learning to model existing proof scripts, and then, given a new theorem, applied that model to guide metaheuristic search [83] to attempt to synthesize a new proof script [223, 179, 92]. However, models in prior tools fail to take into consideration the complete information the programmer has access to when writing proofs, variations in the learning process, the richness of the existing proof corpora, and advances in large language models. The central goal of this dissertation is to develop tools to improve these prior proof synthesis techniques and to enable fully automating more verification.

I first present TacTok, a proof-synthesis tool that models proofs using both the partial proof written thus far and the semantics of the proof state. In this work, I observe that when programmers use interactive theorem provers, they use both the feedback from the theorem prover — the proof state — and the partial proof script already written. I show that by carefully modeling and combining the partial proof script already written and the proof assistant's feedback, we can automatically prove different theorems than models that only use one source of input.

I then present Diva, a proof-synthesis tool that controls the learning process to produce a diverse set of models and efficiently combines these models to improve the overall proving power. I make two observations that enable this improvement. First, proof synthesis has a correctness oracle, namely, the theorem prover. Second, variations in the models can change the search-based synthesis of a proof script such that two models can potentially produce different scripts for the same theorem, which can lead to models proving complementary sets of theorems. Due to the first observation, they can be combined without sacrificing their power.

I then present Passport, a proof-synthesis tool that systematically explores different ways of encoding identifiers in proofs to improve synthesis. Prior tools in this space have simplified the data extracted from proof corpora, and so they have sacrificed modeling the depth of information available. This tool extracts and models these identifiers to improve proof synthesis.

Finally, I present Baldur, a proof-synthesis tool that uses transformer-based pretrained large language models fine-tuned on proofs to generate whole proofs for theorems at once, rather than one step at a time. I also combine this proof generation model with a fine-tuned repair model to repair generated proofs, further increasing proving power. Prior tools using transformer-based large language models invoke hammers or use a search-based approach. I instead show that whole-proof generation using transformers is possible and is as effective as search-based techniques without requiring costly search. Additionally, giving the learned model additional context, such as a prior failed proof attempt and the ensuing error message, results in proof repair and further improves automated proof generation.

This dissertation contributes new ideas for improving automated proof synthesis, and the implementation of these ideas in tools TacTok, Diva, Passport, and Baldur empirically demonstrates that the improvement is significant on large benchmarks. I implemented TacTok, Diva, and Passport for Coq and evaluated them on the CoqGym benchmark [223], which has a test set of over 10K theorems. Following prior work using transformer-based approaches, I implemented Baldur for Isabelle and evaluated it on the PISA benchmark [100], which has a test set of over 6K theorems.

This dissertation is structured as follows. Chapter 2 provides background on interactive theorem proving. Chapter 3 describes TacTok, Chapter 4 describes Diva, Chapter 5 describes Passport, and Chapter 6 describes Baldur. Chapter 7 places my work in the context of related research. Chapter 8 summarizes my dissertation and details potential future directions.

CHAPTER 2

BACKGROUND ON INTERACTIVE THEOREM PROVING

This chapter provides background on using an interactive theorem prover, specifically the Coq proof assistant [199]. Using the Isabelle proof assistant [94] is a similar experience. Section 2.1 details different aspects of Coq and defines key terms used throughout the dissertation. Section 2.2 provides an illustrative example of using Coq. This chapter is adapted directly from the background sections in the publications corresponding to the next chapters.

2.1 Theorem Proving in Coq

Coq is a dependently-typed language with a small kernel, which provides a high assurance that Coq-verified programs are truly correct. However, program verification in Coq is not automatic. To prove a theorem in Coq, a programmer must write a proof script (in Ltac), which, when executed, helps automatically generate a proof (in Gallina) of the theorem.

When a theorem is proven in Coq, this means that in Gallina (Coq's internal language), a *proof term* of the desired type has been constructed. The type of this term is the theorem itself. A programmer could write the Gallina proof term themselves, but this can be a long, unforgiving process [170]. To simplify this task, Coq has a meta-programming language called Ltac in which programmers can write *proof scripts*, which when completed and run, generate the Gallina proof term automatically.

Programmers use a *proof assistant* (e.g., CoqIDE or Proof General) to write proof scripts, which consist of a sequence of *proof tactics*. The proof assistant executes a proof script, even a partial one, and provides immediate human-readable feedback after each tactic's execution. This feedback is Coq's internal *proof state*, which includes the *goals* to prove, the *local context* of assumptions, and the *environment* of proven-so-far set of facts. The programmer can even ask to see the intermediate Gallina proof term by writing and executing the `Show Proof` command in their proof script. When starting to prove a theorem, Coq's proof state is a single goal, which is the theorem itself (the corresponding proof term is `?Goal`). The aim is to manipulate the proof state through the use of tactics until the goal is proven and thus removed from the proof state. Since the search space of goal manipulation is too large, a programmer helps manage the exploration by using the current proof state to select a sequence of proof tactics to try.

The proof assistant checks that a partially-written proof script is valid and updates the current proof state, allowing the programmer to incrementally develop a proof script. The programmer can choose a tactic, examine the output from the proof assistant, and then choose the next tactic. If the programmer chooses an invalid tactic, the proof assistant displays an error. If the programmer chooses tactics that are valid, but do not make progress, they can use the proof assistant to backtrack to an earlier proof state and try a different approach. The programmer continues selecting tactics until the proof assistant prints *no more subgoals*, and then uses `Qed` to complete the proof script.

Alternatively, metaheuristic search techniques [83] can automatically search for a proof script, thus alleviating the burden for the programmer [223, 62, 179]. However, metaheuristic search is only as good as the predictive model that is used to bias the search. Building a good predictive model is the focus of most of this dissertation.

2.2 Using Coq to Prove Addition Is Associative

```
1  Inductive nat : Set :=
2    | O : nat
3    | S : nat -> nat.
4
5  Fixpoint add (n : nat) (m : nat) : nat :=
6    match n with
7      | O -> m
8      | S p -> S (add p m)
9    end
10   where "n + m" := (add n m) : nat_scope.
11
12 Theorem assoc : forall n m p : nat,
13    n + (m + p) = (n + m) + p.
14 Proof.
15   intros.
16   induction n.
17   reflexivity.
18   simpl.
19   rewrite -> IHn.
20   reflexivity.
21 Qed.
```

Figure 2.1: The definition of the natural numbers (lines 1–3), the function to add unary natural numbers (lines 5–10), a theorem that the function is associative (lines 12–13), and a proof script of that theorem (lines 14–21).

We demonstrate Coq's use with a small, simple example. We define the natural numbers and then prove that addition is associative.[1] We define the natural numbers (`nat`) using a unary encoding (Figure 2.1, lines 1–3), where `Z` is the natural number 0, and the rest are defined using the successor operator (`S n`).

Lines 5–9 in Figure 2.1 define a recursive function to add two numbers (`add n m`), where addition is evaluated from the left. Line 10 defines the notation `n + m` as a shorthand for `add m n`. The function itself maps the answer to `m` if `n = O` (line 7), and otherwise to `S (add p m)`, where `S p = n` (line 8). In other words, the recursive step returns the successor of the sum of `m` and the number whose successor is `n`.

[1]While Coq's standard library already includes these definitions, this basic example is simple enough to effectively demonstrate our approach.

The `assoc` Theorem (lines 1–3) defines the associativity property, `n + (m + p)= ( n + m)+ p`, for all `n, m, p`.

To prove this property, Coq needs help. Coq maintains an internal *proof state*, which includes the goals that Coq wants to prove and a list of assumptions. When starting to prove a theorem, Coq's proof state is a single *goal*: the theorem itself (shown in fig. 2.2a). Coq manipulates the proof state (with the help of proof tactics, described next) until the goal is proven and can be removed from the proof state. These manipulations are enabled by *proof tactics*. Some tactics create new subgoals (e.g., `induction`) and other tactics manipulate the current goal in various ways (e.g., `simpl`, which simplifies complex terms in the goal, and `rewrite`, which transforms a term in the goal into an equivalent term). The search space of goal manipulation is too large, and a human helps Coq by writing a sequence of proof tactics that help Coq manage the exploration. Lines 14–21 are the human-written sequence of tactics, one per line, that help Coq prove associativity.

To write down the appropriate sequence of tactics, the programmer needs to understand Coq's internal proof state. The interactive proof assistant checks that a partially-complete proof script is valid and shows the current proof state to the programmer. This makes it possible for the programmer to incrementally develop a proof script, instead of writing a complete proof script in a single step. The programmer can choose a tactic, examine the output from the proof assistant, choose the next tactic, and so on. If the programmer chooses an invalid tactic, the proof assistant displays an error. (For example, the `induction` tactic signals an error if its argument is not inductively defined.) If the programmer chooses tactics that are valid, but do not make progress, they can use the proof assistant to backtrack to an earlier proof state and try a different approach.

We now step through the proof script of associativity and discuss both the proof tactic entered by the programmer and the proof state displayed by the proof assistant.

```
1 subgoal
______________________________________(1/1)
forall n m p : nat, n + (m + p) = n + m + p
```

(a) Proof state at the start of the proof script (after line 14 in Figure 2.1).

```
1 subgoal
n, m, p : nat
______________________________________(1/1)
n + (m + p) = n + m + p
```

(b) Proof state after line 15 in Figure 2.1 (`intros` tactic).

```
2 subgoals
m, p : nat
______________________________________(1/2)
0 + (m + p) = 0 + m + p
______________________________________(2/2)
S n + (m + p) = S n + m + p
```

(c) Proof state after line 16 in Figure 2.1 (`induction n` tactic).

```
1 subgoal
n, m, p : nat
IHn : n + (m + p) = n + m + p
______________________________________(1/1)
S n + (m + p) = S n + m + p
```

(d) Proof state after line 17 in Figure 2.1 (`reflexivity` tactic).

```
1 subgoal
n, m, p : nat
IHn : n + (m + p) = n + m + p
______________________________________(1/1)
S (n + (m + p)) = S (n + m + p)
```

(e) Proof state after line 18 in Figure 2.1 (`simpl` tactic).

```
1 subgoal
n, m, p : nat
IHn : n + (m + p) = n + m + p
______________________________________(1/1)
S (n + m + p) = S (n + m + p)
```

(f) Proof state after line 19 in Figure 2.1 (`rewrite -> IHn` tactic).

Figure 2.2: The proof state after the execution of each tactic of the `add` function's associativity proof script.

At the start of the proof script, the proof state has a single goal, which is the theorem itself (fig. 2.2a). Since the goal is a statement that should hold `forall` naturals `n`, `m`, and `p`, the natural next step is to assume the existence of some arbitrary `n`, `m`, and `p`. The `intros` tactic eliminates the `forall` quantifier and introduces three new variables as *assumptions* (line 5 in fig. 2.2b). Since `add` is inductively defined, it is natural to perform induction on one of these variables (e.g., `induction n`), which leads to two subgoals, one for the base case and the other for the inductive case (fig. 2.2c). Coq first focuses on proving the base case (line 11 in fig. 2.2c) by labeling it subgoal 1 of 2 (line 10) and only presenting assumptions (line 9) relevant for the base case. Note that in the base case, `n` is no longer in the list of assumptions, and has been replaced with the value `O` in the subgoal (line 11 in fig. 2.2c). The `reflexivity` tactic does some basic simplification and solves the base case, since both sides are essentially identical. Therefore, the next proof state has just one goal, which is the inductive case (fig. 2.2d). In this state, `n` has been replaced with `S n`, and the proof assistant shows the inductive assumption (`IHn`). However, `IHn` is not immediately applicable because it contains `n` instead of `S n`, and so trying to use it will produce an error. Instead, the `simpl` tactic tries to simplify the goal while keeping it readable (though many Coq users find that `simpl` is a complicated tactic that produces unpredictable results [199]). After choosing `simpl`, the left-hand side of the goal has an expression that is identical to the left-hand side of the inductive assumption (fig. 2.2e). Therefore, the `rewrite` tactic can replace that expression with the right-hand side of the inductive assumption (fig. 2.2f), which leads to a trivial equality that `reflexivity` can solve. Finally, the proof assistant prints *no more subgoals* and `Qed` completes the proof script.

CHAPTER 3
TACTOK : SEMANTICS-AWARE PROOF SYNTHESIS

The work in this chapter was done in collaboration with Arjun Guha and Yuriy Brun. The following is adapted directly from published work [62].

3.1 Introduction

Researchers have developed a variety of tools and techniques to make writing proof scripts in Coq (and other interactive theorem provers) easier to write [86, 89, 109, 92, 223]. Inspired by the success of statistical code completion tools [91, 90, 208], Hellendoorn et al. observed that the sequence of tokens in Coq proofs is somewhat predictable [86], and this could hypothetically be used to build statistical proof script completion tools. Other researchers have used the current proof state to build proof script synthesis tools [223, 92]. For example, ASTactic [223], a deep learning model, takes as input the current goal, local context, and environment, and predicts the next step of the proof script.

In this work, we observe that when programmers use interactive theorem provers, they use both the feedback from the theorem prover — the proof state — and the partial proof script already written. We argue that models that learn from the proof state and the partial proof script, together, can be more effective at synthesizing proof scripts. When programmers write Coq proof scripts themselves, they critically rely on the feedback detailing the internal proof state provided by the interactive proof assistant to choose the next proof tactic to try [173]. In fact, writing proof scripts without the proof assistant's feedback is often *impossible*. For example, Coq has several sophisticated proof tactics that can make partial progress that is hard to predict. Moreover, even some simple tactics automatically generate new variable names that programmers need to know for subsequent tactics, and thus they rely on the proof assistant to learn these generated names. In a way, the programmer and the proof assistant are engaged in a conversation. However, the existing methods of predicting the next tactic from the internal proof state alone [223] and predicting the next tactic from the previous tactics alone [86] only consider distinct sides of the conversation. In this work, we show that by carefully modeling and combining the partial proof script already written and the proof assistant's feedback, we can automatically prove different theorems than models that only use one source of input.

We present TacTok, an open-source implementation of our approach to proof script synthesis that incorporates both proof state and the partial proof script already written. We apply our approach to a benchmark of 122 open-source software projects in Coq, with over 68K theorems. We demonstrate that the inclusion of both proof state and the partial proof script in a proof synthesis model can improve its effectiveness and allow it to synthesize different proof scripts than other tools.

We compare our approach to five approaches, two existing tools, CoqHammer and ASTactic, and three new ones we create, SeqOnly, WeightedRandom, and Weighted-Greedy. CoqHammer [49] is a state-of-the-art proof synthesis technique and ASTactic [223] is a proof script synthesis technique that models only proof state. SeqOnly is a proof script synthesis technique that models only the partial proof script and the initial theorem, and WeightedRandom and WeightedGreedy are proof script synthesis techniques that attempt to construct a proof script via metaheuristic search biased by using the frequency distribution of tactics as they appear in the training data. All of the above techniques, except CoqHammer, are metaheuristic search techniques [83] that search through the proof-script space using a model that biases the search toward the likely proof script steps.

Evaluated on a benchmark of 26 software projects in Coq with 10,782 theorems, we find that each of these techniques can successfully generate proof scripts that prove theorems, which is strong evidence for the success of metaheuristic search in proof script synthesis. TacTok can automatically prove more theorems than ASTactic (12.9% versus 12.3%). Both techniques outperform SeqOnly, which can only prove 10.0% of the theorems. Even without explicitly modeling the theorem being proven, WeightedRandom proves 6.2% of the theorems and WeightedGreedy proves 9.7% of the theorems. Importantly, TacTok is complementary to prior tools. When used together with ASTactic, TacTok can prove 20% more theorems than ASTactic alone. CoqHammer, a very different approach, is able to synthesize proofs for more theorems than TacTok (26.6%), but TacTok and CoqHammer together can prove 11.5% more theorems than CoqHammer alone.

Overall, our experiments show that there is promise to using metaheuristic search and modeling the partial proof scripts and proof state, together, for synthesizing proof scripts for formal verification. We investigate the effect of using various kinds of information encoded in the partial proof script on verification efficacy and produce guiding information for improving future proof script synthesis tools.

The main contributions of this work are:

- A novel automated proof script synthesis technique, TacTok, that, unlike prior work, models the combination of the partial proof script already written and the proof state to automate proof script synthesis and formal verification. TacTok is open-source.

- Three other new metaheuristic search proof script synthesis techniques, SeqOnly, WeightedRandom, and WeightedGreedy.

- An application of ASTactic, a state-of-the-art proof script synthesis tool, independently reproducing prior evaluation results [223].

- An evaluation of the value added by using both the partial proof script and proof state over using each one alone, and a comparison to two state-of-the-art proof synthesis tools and three new tools, showing that TacTok proves theorems no prior tool can prove.

- A public release of TacTok — `https://github.com/LASER-UMASS/TacTok` — and our experimental scripts and data [61].

- An exploration of the information TacTok models to improve verification efficacy.

3.2 Illustrative Examples

Recall that Section 2.2 introduced a small Coq program for adding two numbers and proved, using Coq, that the function is associative. Section 3.2.1 shows how TacTok would generate such a proof script automatically. Section 3.2.2 then introduces a more complex theorem expressed in a higher-order logic that TacTok is able to prove, whereas prior tools do not.

3.2.1 Proving Associativity with TacTok

Recall the example of proving additive associativity with Coq from Section 2.2. Note that the proof script (lines 15–20 of Figure 2.1) is almost unintelligible without examining the internal proof state, for several reasons:

1. Some tactics generate new variable names (e.g., `induction`), which subsequent tactics (e.g., `rewrite`) use. The programmer needs to know these names to use them in the proof script.

2. Some tactics apply heuristics that are hard to predict. In the example in Figure 2.1, `simpl` (line 18) uses heuristics to produce readable output. Without looking at the resulting proof state in fig. 2.2e, the programmer would not know

what form the goal is transformed into and whether `IHn` can be invoked without an error.

3. When a proof script has multiple goals, it may not be obvious where one goal ends and the next goal begins. In the example in Figure 2.1, the base case is addressed by the tactic in line 17, while the inductive case is addressed by the tactics in lines 18–20. However, without examining the proof state fig. 2.2d, it would not be clear that `reflexivity` completely solved the base case.

4. After a few steps of the proof script, the current goal to prove has likely evolved significantly from the original goal (the theorem). In the example in Figure 2.2, the programmer is trying to prove the theorem fig. 2.2a line 3. After three steps in the proof script, the programmer must now prove fig. 2.2d line 18 using assumptions lines 15–16. The goal has changed significantly and there are now assumptions that didn't exist at the start. Knowing the new goal and the assumption `IHn` are helpful to the programmer at this point in deciding the next tactic: the programmer sees that the goal needs to be transformed using `simpl` to apply the assumption. Without seeing the current proof state in fig. 2.2d, the programmer would not know how to proceed.

Since the programmer needs to examine the proof state to choose the next tactic, it is a good idea to consider whether models of next tactic prediction might also need and benefit from having access to information in the proof state. However, since proof scripts, like programs, build upon the commands executed thus far [91, 90, 86], a model of next tactic prediction may also benefit from having access to the previous tactics in the proof script.

TacTok builds a model of next-tactic prediction for Coq. The model is learned using a corpus of existing proof scripts. TacTok decomposes each of the existing proof scripts by stepping through them, one tactic at a time, and computing the resulting intermediate proof states. The model automatically learns an embedding of the proof states and the partially written proof script and maps these to an abstract syntax tree (AST) of the next line in the proof script. Thus, the model's inputs are the partially written proof script and the proof state. For example, one input to the model is the intermediate proof state in fig. 2.2d and the partial proof script [`intros`, `induction n`, `reflexivity`] that achieved that proof state. The model's output is an AST that TacTok decodes to the next predicted tactic and its arguments.

Given a model trained on existing proof scripts (Section 3.3.2.4 presents the training process), TacTok automatically generates the proof script of associativity (lines 14–21). To start, TacTok takes as input the proof state in fig. 2.2a and the partially written proof script, which is only [`Proof`]. For the sake of illustration, suppose the model correctly predicts `intros` as the next tactic. TacTok executes this partial proof script using the Coq interactive proof assistant, and checks for two things. First, that executing the partial proof script with the new tactic does not return an error. And second, that the new returned proof state is different from the prior observed proof states in this partial proof script, meaning the tactic had an affect and produced something new. Here, TacTok will see no error and the updated proof state is that in fig. 2.2b. TacTok will next explore growing the proof script further. TacTok's next input to the model is the proof state in fig. 2.2b and the partial proof script [`Proof`, `intros`]. Suppose, again, the model correctly predicts the next tactic and argument is `induction n`; there is no error adding this tactic to the partial proof script and the new proof state is that in fig. 2.2c.

If TacTok proceeds in this way to produce the partial proof script [`Proof`, `intros`, `induction n`, `reflexivity`] and the proof state in fig. 2.2d, but as the next tactic and arguments, TacTok predicts `rewrite -> IHn`. Adding this tactic results in an error. TacTok goes back to the model and asks it for the next most likely tactic and arguments (TacTok would do the same thing if there were no error but the proof state

```
 1 Definition prod : forall (n : nat) (f : nat -> nat), nat.
 2 intros.
 3 induction n as [| n Hrecn].
 4 exact 1.
 5 exact (f n * Hrecn).
 6 Defined.
 7
 8 Lemma sameProd : forall (n : nat) (f g : nat -> nat),
 9        (forall m : nat, m < n -> f m = g m) ->
10        prod n f = prod n g.
```

Figure 3.1: Lines 1–6 defines the function *prod* n f, which calculates $1 \cdot f(0) \cdot \cdots \cdot f(n-1)$. Lines 8–10 define the `sameProd` theorem, which states that if $f(m) = g(m)$ for all $m < n$, then *prod* n f = *prod* n g.

was a duplicate of an earlier state). Suppose the model suggests `simpl`, which turns out to return no error and changes the state.

TacTok will proceed in this way, performing a depth-first search through the space of proof scripts until, either, it reaches the proof state “no more subgoals” (meaning the proof script is completed successfully) and adds `Qed` to the proof script, or it times out and fails to complete the proof script.

Of course, TacTok can only be successful if its model is fairly accurate in its predictions.

3.2.2 Proving Higher-Order Logic Theorems

The associativity theorem we just discussed is an example of a theorem expressed in first-order logic. Theorems expressed in a higher-order logic are likely to be more complex, and so their proof scripts are more difficult to generate. One example of higher-order logic in Coq is the presence of a nested `forall`. Figure 3.1 defines the function `prod n f`, which calculates $1 \cdot f(0) \cdot \cdots \cdot f(n-1)$. The same figure defines the `sameProd` theorem, which states an intuitive property: for functions f and g, if $f(m) = g(m)$ for all $m < n$, then *prod* n f = *prod* n g. TacTok is able to generate a proof script for `sameProd`. CoqHammer [49] fails to find a proof in ten minutes, and

ASTactic [223] cannot find a proof script either. Section 3.5.5 will further discuss theorems that TacTok proves that prior tools cannot.

We next describe prior work on training language models and our improvements over the state of the art.

3.3 The TacTok Approach

Section 3.3.1 describes language modeling methods and Section 3.3.2 details how TacTok builds on language modeling approaches to generate proof scripts. Our TacTok implementation is publicly available at `https://github.com/LASER-UMASS/TacTok/`.

3.3.1 Language Modeling

Language models estimate the probability of a particular instance in a language. For example, a language model can estimate the likelihood that the words “I went to the” are followed by “store”. When applied to natural languages, these models have aided speech recognition, machine translation, spelling correction, and other natural language processing tasks [192, 26, 108].

While language models can be created in many ways, including manually, a typical approach is to train a language model on a large corpus of example sentences in a language. There are two main classes of language models widely used in modern natural language processing research. The first consists of statistical language models, often called n-gram language models. These count-based models work on the Markov assumption that the conditional probability of a word is dependent on a fixed number (n) of previous words in a sentence. The second class of models are based on neural network architectures. Neural models often perform better on natural language applications than statistical models, though are less human-readable [195, 136].

3.3.1.1 Statistical Language Models

N-gram models predict the next word based on a sequence of the previous $n-1$ words [104]. Given a vocabulary of words, during training, an n-gram model counts the number of times in the training corpus that each word follows each possible sequence of $n-1$ words. At prediction time, given a sequence of $n-1$ words, the model returns the probability distribution of words that follow that sequence. Formally, given a sequence S of $n-1$ words, $S = \langle w_1, w_2, \dots, w_{n-1}\rangle$, the n-gram model estimates $P(w_n|w_1, w_2, \dots, w_{n-1})$, the probability distribution of words in the nth place.

The maximum likelihood estimate for this probability, $P_{ML}(w_n|w_1,\ w_2, \dots, w_{n-1})$, is the number of times $\langle w_1, w_2, \dots, w_n\rangle$ appears in the training corpus, divided by the number of times $\langle w_1, w_2, \dots,\ w_{n-1}\rangle$ appears in the corpus.

$$P_{ML}(w_n|w_1, w_2, \dots, w_{n-1}) = \frac{count(w_1, w_2, \dots w_n)}{count(w_1, w_2, \dots, w_{n-1})}$$

This can be used as an estimate but because training data may not contain every possible sequence $\langle w_1, w_2, \dots, w_n\rangle$, it would end up with a lot of zero probability estimates. So rather than underestimate all $P(w_n|w_1, w_2, \dots, w_{n-1})$ as 0, linear interpolation [192] uses smaller subsequences of length 1 to $n-1$ to estimate the probability:

$$P(w_n|w_1, w_2, \dots, w_{n-1}) = \sum_{i=1}^{n} \lambda_i \times P_{ML}(w_n|w_{n-i+1}, \dots, w_{n-1})$$

where λ_i is a normalized weight given to the value $P_{ML}(w_n|w_{n-i+1},\ \dots, w_{n-1})$. These weights can be considered as *fallback* weights to fallback and look at smaller subsequences if the larger subsequence is not present in the training corpus.

Discounting smooths the probability distribution over the vocabulary by reallocating the probability mass from the training dataset by subtracting a fixed discount d from the counts of each word in the training dataset and then reassigning it to the fallback probability. The state-of-the-art n-gram language model is the Modified Kneser-Ney model [107], which implements a more involved discounting technique.

As described in Section 4.1, recent work has applied n-gram models (as well as other models) to Coq proof scripts to predict tokens, which can, in theory, be used to build statistical proof script completion tools.

3.3.1.2 Neural Language Models

Feed-forward neural networks [175] can also model language. These models are also trained to predict the next word given a sequence of previous words but they do this using a different architecture than that of n-gram models.

Given the previous $n-1$ words $\langle w_1, \ldots, w_{n-1} \rangle$, these models first create real-valued vectors $\langle r_1, \ldots, r_{n-1} \rangle$ that represent the words. This is commonly done by converting each of the words into a one-hot vector representation and then passing them through a linear embedding layer. A one-hot vector representation is constructed by first initializing a vector of the size of vocabulary V of the language and then setting the element at the index corresponding to the given word to 1. The linear embedding layer consists of $|V|$ weights W_e and a bias term b_e. The model then concatenates the real valued embeddings to form a summary vector r. The model passes r through a predetermined number of neural network layers, together called NN. The output $h = NN(r)$ passes through a softmax layer to produce a probability distribution over the entire vocabulary. In summary, the training algorithm is:

```
1 $o_i = onehot(w_i)$
2 $r_i = o_i.W_e + b_e$
3 $r = \oplus(r_1 \ldots r_{n-1})$
4 $h = NN(r)$
5 $P = softmax(h)$
```

Recently, recurrent neural network (RNN) architectures have been shown to perform well in language modeling tasks [18, 136, 195]. The main difference between this approach from a simple feed-forward neural network is the way the sequence of real valued vector representations, $\langle r_1, \ldots, r_{n-1} \rangle$, are handled to obtain the final hidden layer representation h. An RNN performs a series of steps that each process the sequence up until an index. A simplified version of this process can be described as follows. At each index i of the sequence, an RNN maintains a hidden state h_i, which is a vector representing the summary of the real valued vectors before index i. It then concatenates h_i with r_i and sends the resultant vector through a linear neural network layer with weights W_r and W_h and bias b_r. The output is used as the new summary and hidden state at index $i + 1$, h_{i+1}. The summary of the whole sequence is hence the hidden state after processing all the $n - 1$ real valued vectors. Note that h_1 is a zero vector since there are no elements to summarize before the first element in the sequence. Following those steps, we obtain h and then, a softmax layer converts this into the probability distribution over the vocabulary. In summary, the RNN training algorithm is:

```
1 $o_i = onehot(w_i)$
2 $r_i = o_i.W_e + b_e$
3 $h_{i+1} = RNN(r_i, h_i) = r_i.W_r + h_i.W_h + b_r$
4 $h = RNN(r_{n-1}, h_{n-1})$
5 $P = softmax(h)$
```

Long short-term memory networks (LSTMs) extend recurrent neural networks to learn long-term dependencies by tackling a problem that arises with basic RNNs called the vanishing gradient problem [70, 73]. For long sequences, the RNN framework scales down the gradient over each word going backwards and the value of the gradient become negligible, making it difficult for the model to learn. To address this problem, LSTMs redesign the basic building block of the neural network to include components called *gates*. At each step in the sequence, the model makes decisions based on three gates: the input gate, the output gate, and the forget gates. These gates control whether the model should remember or forget the summary of the sequence so far through a set of parameters that are learned jointly with the parameters of the RNN during training. Note that the hidden state h is still calculated sequentially like a normal RNN, except with the model choosing to ignore the summary vector sometimes. The exact implementation of the gates can be found in literature and are beyond the scope of this work.

A Bidirectional LSTM [157] runs the sequence input in two ways, backwards and forward, allowing the output layer to see both directions of the information simultaneously. This improves upon the LSTM by capturing more information.

Transformers [54] are the current state-of-the-art in neural language models, but they only perform well when there is a very large amount of data for training. Thus, LSTMs and variants like gated recurrent neural networks (GRUs) [44] are sufficient for neural language models that do not have millions of sequences for training. They have been shown to do well in machine translation and sentiment analysis [12, 190].

While RNN language models use sequences of words to predict the next word in the sequence, RNNs in general can be used to model other types of sequential objects for any prediction task. In particular, we are interested in using sequential data — sequences of proof script tokens — to predict the next tactic, which is from a different vocabulary. An RNN allows encoding a sequence of n real-valued vectors as one representative vector h through the following steps.

```
1  $h_1 = [0, \ldots, 0]$
2  $h_{i+1} = RNN(r_i, h_i) = r_i.W_r + h_i.W_h + b_r$
3  $h = RNN(r_n, h_n)$
```

LSTMs and GRUs can hence be used to effectively model general sequences of data when there are only thousands of training sequences, and so we use them to model the sequence of proof script tokens.

3.3.2 TacTok Language Modeling

TacTok first trains a model on existing proof scripts, and then applies the model to synthesize new proof scripts. Figure 3.2 details how TacTok trains a model on a set of proof scripts. Given a set of proof scripts, each proof script is broken down

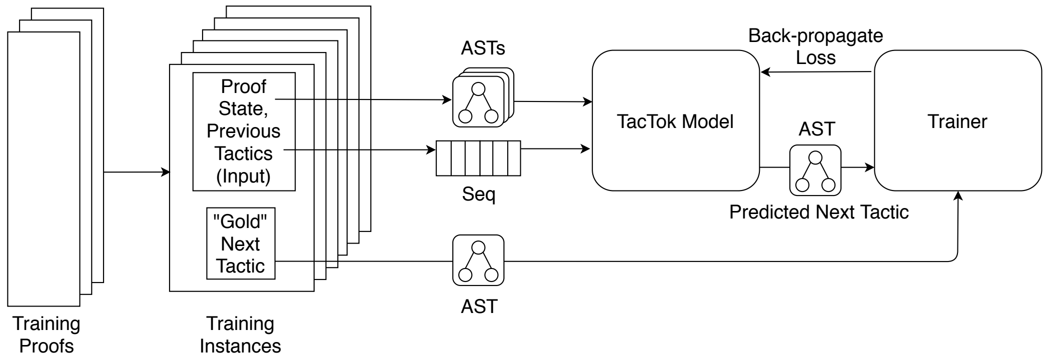


Figure 3.2: TacTok training process. Given a set of proof scripts, TacTok breaks down each proof script into training instances, consisting of the input to the model (proof state and the proof script thus far up to this state), and the next step of the proof script. The TacTok model jointly learns embeddings for the proof state ASTs and the proof script thus far, and uses these embeddings to predict the next tactic in the form of its AST. The learning process back-propagates the difference between the predicted tactic AST and the (expected) next step as the loss.

into training instances. A training instance consists of the input to the model, which is the proof state after a tactic in the proof script is executed and the proof script up to the point of the executed tactic, and the next step of the proof script. The proof state is made up of the current goal, local context, and environment. Each term in the proof state has an underlying AST. The proof script is represented as a sequence of tokens. The TacTok model jointly learns embeddings for these ASTs and sequences. The TacTok model uses these embeddings to output a predicted next proof script step, in the form of an AST, and sends that along with the AST form of the ground-truth next tactic to the trainer. The trainer then compares these tactics and back-propagates the loss.

Figure 3.3 details how TacTok synthesizes proof scripts, using the proof script for the proof of associativity of the add function (recall Section 3.2.1) as an example. The figure shows TacTok at the point where it has already generated [`intros`, `induction n`, `reflexivity`, `simpl`] of the proof script. TacTok takes as input the proof state and the partial proof script so far. The proof state encoder (Section 3.3.2.1) takes the

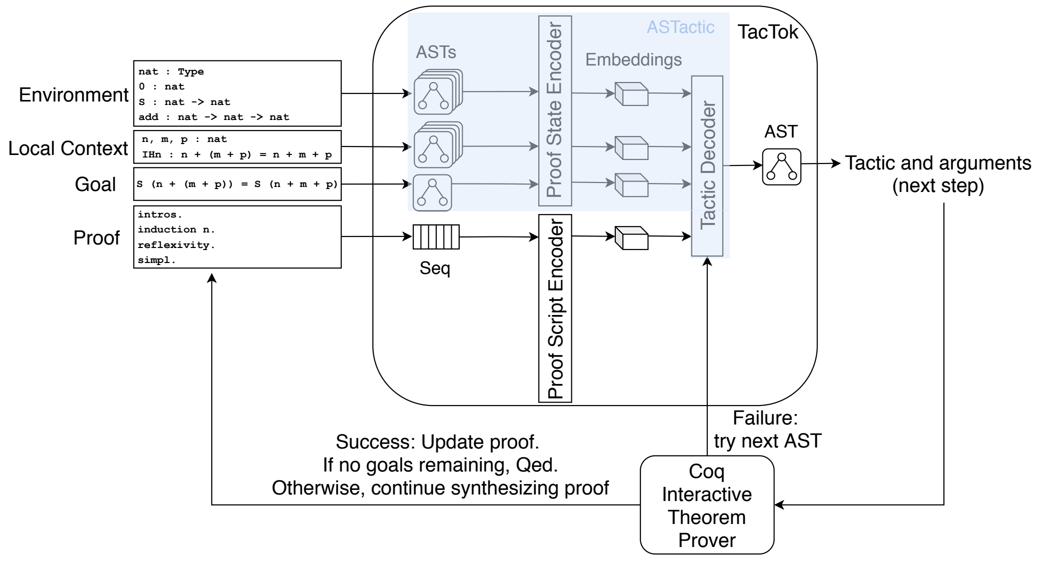
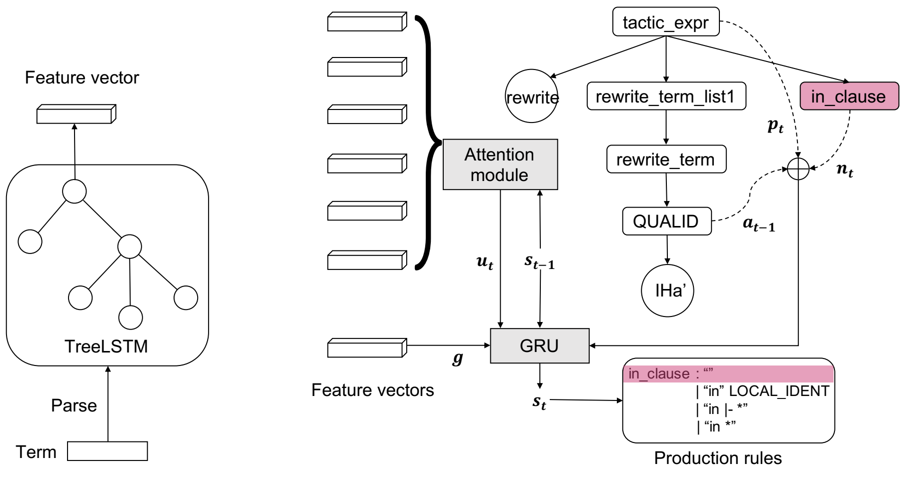


Figure 3.3: TacTok architecture. TacTok, in the process of completing the proof script of associativity for the add function (recall Section 3.2), after the execution of `simpl`. TacTok's input is the proof state (goal, local context, and environment) and the partial proof script synthesized so far. Each term in the proof state has an underlying AST, and the proof state encoder generates embeddings for each of these ASTs, as done in ASTactic. The proof script is represented as a sequence, and the proof script encoder generates embeddings from this sequence. The tactic decoder, modified from ASTactic, uses these embeddings to generate the next predicted tactic and its arguments in the form of an AST. TacTok interfaces with the Coq Interactive Theorem Prover to execute the higher level expression associated with this predicted tactic. Upon failure, TacTok resamples the tactic and its arguments and tries again. Upon success, TacTok updates the proof script and proof state to reflect the execution of the tactic. This process continues until the proof script is complete or times out.

AST form of the proof state inputs and outputs embeddings for each. The proof script encoder (Section 3.3.2.2) takes the sequence of tokens in the proof script and outputs an embedding for this sequence. The tactic decoder (Section 3.3.2.3) uses these embeddings to predict an AST, which represents a tactic and its arguments. TacTok then interfaces with the Coq ITP to execute this tactic and its arguments and interprets the Coq ITP output as either a *failure* or a *success*. A failure is when the Coq ITP produces an error or the execution of the tactic produces a duplicate proof state. In this case, TacTok resamples the tactic and its arguments and interfaces with the Coq ITP again. Otherwise, it is a success and the proof script and proof state



(a) Proof State Encoder. (b) Tactic Decoder.

Figure 3.4: ASTactic architecture from [223]. The proof state encoder (a), takes as input the goal, local context, and environment terms in AST form and generates embeddings (feature vectors) for each term. The tactic decoder (b) concatenates the input embeddings and generates a tactic in the form of an AST, conditioned on these inputs.

update to include the tactic and its arguments and resulting proof state from the tactic's execution. This process continues until the proof script is complete or times out.

We now describe each of the TacTok components.

3.3.2.1 Proof State Encoder

Figure 3.4a details the encoder in ASTactic [223], which is the proof state encoder in TacTok. The inputs are the goal, local context, and environment in AST form. It uses a TreeLSTM network [198], which allows for encoding a tree, to generate embeddings for each proof state term.

3.3.2.2 Proof Script Encoder

The proof script consists of tokens that are either tactics, arguments to tactics, or other symbols. The proof script encoder parses the sequence of these tokens in two

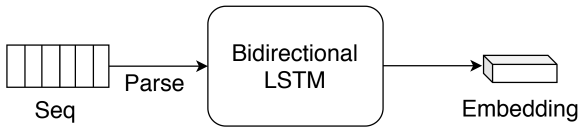


Figure 3.5: Proof Script Encoder. The proof script encoder's input is a sequence of proof script tokens, which it parses and then uses to generate an embedding using a Bidirectional LSTM.

different modes. One mode of parsing the sequence is to include the most common Coq tactic tokens appearing in the training proof scripts, excluding custom tactics, and obscure arguments so that their names are not learned. When the proof script encoder parses in this way, the model it trains is the *Tac* model. Another mode of parsing the sequence is to include the entire token sequence, only excluding punctuation. When the proof script encoder parses in this way, the model it trains is the *Tok* model. TacTok is comprised of both the Tac and Tok models, trained separately, and uses either one when attempting to synthesize a proof script.

We encode the parsed sequence of previous tokens using a Bidirectional LSTM, as described in Section 3.3.1.2, which generates an embedding for the sequence. Figure 3.5 shows the proof script encoder in TacTok.

3.3.2.3 Tactic Decoder

Figure 3.4b shows the tactic decoder in ASTactic [223], which is the tactic decoder in TacTok. It is conditioned on the sequence of input embeddings. In TacTok, the embeddings are a concatenation of the embeddings generated from both the proof state and proof script encoders. The decoder generates a tactic in the form of a program by sequentially growing an AST [226]. At a non-terminal node in the AST, it chooses a production rule from the specified context free grammar (CFG) of the tactic space. At a terminal node, it synthesizes an argument based on semantic constraints. This process of growing the tree is controlled by a GRU [42, 44], which uses the input embeddings of the partially generated AST to update its hidden state.

The tactic decoder has learnable embeddings for all production rules and symbols in the tactic CFG. For example, at time t, n_t is the symbol of the current node, a_{t-1} is the production rule for expanding the prior node, p_t is the production rule for expanding the parent node concatenated with the parent's state, g is the concatenation of the input embeddings, and u_t is the weighted sum of the premises in the environment and local context. The decoder concatenates a_{t-1}, p_t, n_t, g, and u_t, and uses a GRU controller to combine them with the information from the partial tree s_{t-1} to update the decoder state s_t. Then, the decoder uses s_t to predict which production rule to apply.

3.3.2.4 Training and Search

TacTok is trained on a set of proof scripts. Each proof script in the set is broken down into training instances. A training instance details the proof state after a tactic is executed and the proof script up to the point of the executed tactic. TacTok is trained in same way as the ASTactic model, except for that it also jointly trains a language model over the previous tokens in a proof script.

For automated theorem proving, TacTok uses depth-first search, just like ASTactic. TacTok samples a fixed number of the most likely tactics across all search tree nodes at the same level, and then uses these tactics to search for a complete proof script. TacTok backtracks when it detects a duplicate proof state, or when the Coq compiler fails to check the new attempted proof script step.

3.4 Proof and Proof Script Synthesis Tools

Our evaluation will compare TacTok to five tools, CoqHammer, ASTactic, SeqOnly, WeightedRandom, and WeightedGreedy. This section describes these tools.

The most powerful, general-purpose techniques for automating verification in an ITPs, such as the Coq ITP, are called *hammers* [25]. A hammer performs efficient automated reasoning using facts from a preexisting library. It uses machine-learning techniques to select the facts that are likely to be needed to prove a theorem. Then, it translates these selected facts and the theorem from the ITP logic to a form accepted by Automated Theorem Provers (ATPs), which are theorems provers for first-order logic. The ATPs take the resulting translation and try to find a proof. Lastly, the hammer processes the proof found by an ATP, and tries to reconstruct the proof of the theorem in the ITP logic. CoqHammer [49] is one such hammer for Coq. As in prior evaluations of automated verification tools that compare to CoqHammer [223] our experiments configure CoqHammer to use four ATPs: Z3 [52], Vampire [110], CVC4 [16], and E Prover [181].

ASTactic is a search-based automated verification tool that uses deep learning to learn to predict the next step of a proof script solely from the current proof state [223]. To make a prediction for a given, incomplete proof script, ASTactic uses the current goal, local context, and environment, and generates tactics in the form of ASTs (just like TacTok).

Our central goal in developing TacTok is to demonstrate the effect of modeling both the proof state and the partial proof script on proof script synthesis. Since ASTactic [223] models proof state alone to synthesize proof scripts, a direct comparison between TacTok and ASTactic demonstrates the effect of adding the partial proof script to the model. A natural complement is then a tool that models only the partial proof script, to measure the effect of adding the proof state to the model. However, such a tool, without knowing any of the goals, including the theorem being proven, would follow the same pattern for every theorem and would prove very few, if any, of them. (Anecdotally, we implemented such a tool and confirmed that it proves a small number of theorems.) Instead, we developed SeqOnly, a tool that follows the same basic structure as TacTok and models the partial proof script, but instead of modeling the entire proof state, it only models the theorem being proven. SeqOnly uses the

TacTok proof script encoder (recall Section 3.3.2.2) to generate embeddings for the proof script sequence, but its tactic decoder (recall Section 3.3.2.3) is conditioned only on the proof script sequence embedding and the initial theorem. SeqOnly has the potential to be effective because language models (n-grams and neural networks) have been shown to be able to predict next tokens in Coq proof scripts [86]. For example, using an RNN to represent the sequence of tokens in a proof script allows for 56.6% accuracy in predicting the next token.

Additionally, we build two more proof script synthesis tools, WeightedRandom and WeightedGreedy, that use the frequencies of tactic ASTs occurring in the training proofs to predict the next likely proof step. WeightedRandom computes a probability distribution over the entire set of tactic ASTs by measuring the frequency of each tactic AST in the training set of proof scripts. It then performs a search, just as TacTok, but using only that probability distribution to make predictions over the next tactic AST. To make the top-k predictions at each step of the search algorithm, WeighedRandom samples k tactic ASTs from this distribution without replacement.

WeightedGreedy uses the same probability distribution as WeightedRandom, but always selects the top-k most frequent tactic ASTs from the training set of proof scripts, as opposed to sampling the distribution. Otherwise, it uses search to construct a proof script in the same way as WeightedRandom.

3.5 Evaluation

We evaluate TacTok by comparing its ability to fully automatically synthesize proof scripts to that of two state-of-the-art synthesis tools, CoqHammer [49] and ASTactic [223], and our own three techniques, SeqOnly, WeightedRandom, and WeightedGreedy. CoqHammer attempts to automatically prove theorems using external ATP systems (recall Section 3.4). ASTactic, like TacTok, learns from existing proof scripts but uses only the proof state to predict the next step of the proof script. Finally, SeqOnly is a tool we created ourselves that learns from existing proof scripts but uses only the partially written proof script and the initial theorem to predict the next step of the proof script. WeightedRandom is a model that constructs a probability distribution over the set of tactics in the training dataset, while WeightedGreedy is a model of the maximum likelihood prediction based on the distribution of tactics in the training dataset.

Our implementations, experimental scripts, and data are publicly available in our replication package [61].

TacTok uses search in the proof-script space, whereas CoqHammer produces proofs in Coq's logic (Gallina) using a fundamentally different approach. When the Coq compiler executes a proof script, it generates a proof. Proof scripts can be wrong — e.g., it is possible to write a proof script that concludes with *Proof completed*, but that is not a valid proof when it is checked by the Coq compiler — but proofs cannot. As such, it is reasonable to compare TacTok (and other proof script synthesis tools) to CoqHammer in terms of the theorems they are able to prove, but as their approaches are so fundamentally different, it should be no surprise that the tools are likely to excel for different theorems, and be complementary. For some simpler classes of theorems, CoqHammer and TacTok are likely to perform similarly well, whereas for others, (e.g., proofs that require induction) CoqHammer will be at a fundamental disadvantage. CoqHammer will be perhaps more predictable in the types of theorems it is able to prove, whereas proof script synthesis tools may provide more surprises. Section 3.5.5 discusses some theorems TacTok was able to prove but other prior tools, including CoqHammer, could not.

Our evaluation uses the state-of-the-art CoqGym benchmark [223] and answers three research questions:

RQ1: Does modeling proof state and the partially written proof script, together, improve automatic verification efficacy, as compared to modeling only the proof state or only the partial proof script?

RQ2: Does modeling the proof state and partial proof script, together, improve automatic verification efficacy over state-of-the-art ATP-using CoqHammer?

RQ3: Does modeling of other (non-tactic) tokens together with the proof state improve automatic verification efficacy as compared to modeling the tactic sequences together with proof state?

We next describe our evaluation methodology (Section 3.5.1), and then answer each of the research questions (Sections 3.5.2–3.5.4). We end with a discussion of proof scripts of complex theorems that TacTok generates (Section 3.5.5).

3.5.1 Evaluation Methodology

This section describes the dataset and metrics our experiments use, and the tools to which we compare TacTok's performance.

3.5.1.1 Dataset

In our evaluation, we use the CoqGym benchmark [223]. In total, we use 68,501 theorems from 122 open-source software projects in Coq. CoqGym is the state-of-the-art benchmark used in prior evaluations of formal verification tools [223].

The CoqGym benchmark comes with a preselected training set of 96 projects with 57,719 human-written proof scripts. The preselected testing set in CoqGym is comprised of 27 projects. We were unable to reproduce prior results for ASTactic's performance [223] for one project, coq-library-undecidability, due to internal Coq errors when processing the proof scripts within that project. Accordingly, we exclude this project from our evaluation. We were able to reproduce the results for the remaining 26 projects of 10,782 theorems, and this is the testing set our evaluation uses. Each project in the testing set has between 2 and 2.1K theorems, summarized in Figure 3.9. Following the methodology in prior evaluations [223], our experiments use the training set to train our models and the testing set to measure efficacy of automated verification. When the training instances are extracted from the proof scripts, there are 189,824 training instances.

3.5.1.2 Metrics

Our experiments measure two key metrics, *success rate* and *added value.* The success rate of a tool is the fraction of all theorems that tool fully automatically generate a proof script for. Success rate is widely used in prior evaluations [92, 223]. The added value of tool A as compared to tool B is the relative increase in the fraction of *new* theorems tool A proves that tool B fails to prove, as compared to the total number of theorems tool B proves. For example, if Tool B proves 10 theorems, and tool A only proves 5, but tool B cannot prove any one of these 5, then tool A's added value is 50% (5 new theorems out of 10 previously provable theorems). In other words, together, tools A and B can prove 15 theorems, a 50% improvement over tool B's 10.

Because we view our approach as complementary to the prior work, our hope is that TacTok will prove some theorems the prior approaches fail to prove (as opposed to strictly improving the absolute success rate). The added value metric captures this measure.

3.5.2 RQ1: Does Modeling Proof State and Partial Proof Script Improve Verification Efficacy?

TacTok uses both the partial proof script and the proof state to predict the next step in a proof script. To understand if this combination of information improves the efficacy of automated proof script verification, we compare TacTok to ASTactic, which

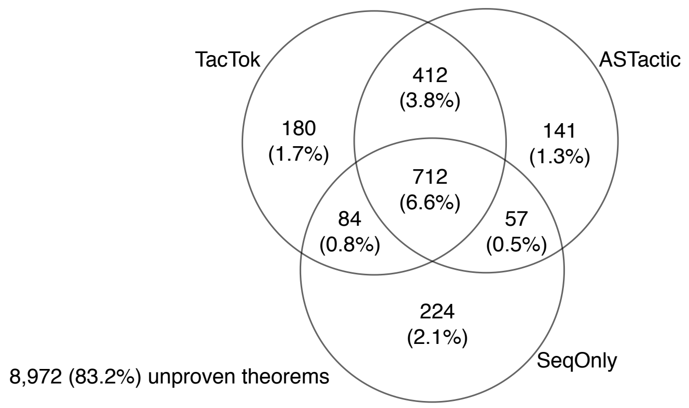


Figure 3.6: Theorems proven by TacTok, ASTactic, and SeqOnly out of the 10,782 theorems in CoqGym's test dataset. Figure 3.9 shows the by-project breakdown of the data. TacTok has the highest success rate of the three tools, proving 1,388 (12.9%) theorems, proving 264 theorems that ASTactic cannot (an added value of 20.0%). TacTok and ASTactic outperform SeqOnly, each with an added value of about 50%. This suggests that proof state together with the partial proof script can achieve better automated verification than either alone.

uses only proof state, and SeqOnly, which uses only the partial proof script and the initial theorem, as well as to our WeightedRandom and WeightedGreedy approaches.

Figure 3.6 compares the number of theorems proven by TacTok, ASTactic, and SeqOnly. TacTok proves 1,388 theorems, for a success rate of $\frac{1{,}388}{10{,}782} = 12.9\%$. ASTactic, trained only on the proof state, has a success rate of $\frac{1{,}322}{10{,}782} = 12.3\%$, and SeqOnly has a success rate of $\frac{1{,}077}{10{,}782} = 10.0\%$.

TacTok proves 264 theorems that ASTactic fails to prove, so its value added compared to ASTactic is $\frac{264}{1{,}322} = 20.0\%$.

TacTok proves 592 theorems that SeqOnly fails to prove, so its value added compared to SeqOnly is $\frac{592}{1{,}077} = 55.0\%$. ASTactic proves 553 theorems that SeqOnly fails to prove, so its value added is $\frac{553}{1{,}077} = 51.3\%$.

SeqOnly is able to prove theorems that ASTactic and TacTok fail to prove, for an added value of $\frac{308}{1{,}322} = 23.3\%$ and $\frac{281}{1{,}388} = 20.2\%$, respectively. This suggests that proof state and the partial proof script are helpful for automated verification when used together and separately.

tool	theorems proven	TacTok value added over tool
WeightedRandom	671 (6.2%)	799 (120%)
WeightedGreedy	1,049 (9.7%)	504 (48%)

Figure 3.7: Theorems proven by the WeightedRandom and WeightedGreedy, out of the 10,782 theorems in CoqGym's test dataset, and TacTok's added value over these baselines. Both WeightedRandom and WeightedGreedy underperform TacTok, which proves 1,388 theorems (12.9%).

Figure 3.7 shows the success rates of WeightedRandom and WeightedGreedy. WeightedRandom proves 671 theorems, for a success rate of $\frac{671}{10{,}782} = 6.2\%$. WeightedGreedy proves 1,049 theorems, for a success rate of $\frac{1{,}049}{10{,}782} = 9.7\%$. Both WeightedRandom and WeightedGreedy underperform TacTok, which proves 1,388 theorems (12.9%).

TacTok proves 799 theorems that WeightedRandom fails to prove, so its added value compared to WeightedRandom is $\frac{799}{671} = 120\%$. WeightedRandom adds little beyond TacTok, proving 82 theorems that TacTok fails to prove, for an added value of $\frac{82}{1{,}388} = 5.9\%$.

TacTok proves 504 theorems that Greedy fails to prove, so its added value compared to Greedy is $\frac{504}{1{,}049} = 48\%$. Greedy proves 165 theorems that TacTok fails to prove, for an added value of $\frac{165}{1{,}388} = 11.9\%$.

RA1: The data suggest that using proof state together with the partial proof script creates significant added value and improves verification efficacy beyond using either kind of information alone. TacTok proves more theorems on its own than ASTactic. Together, they prove 20.0% more than ASTactic alone, and are, therefore, complementary. Meanwhile TacTok and ASTactic outperform SeqOnly, each with an added value of about 50%. WeightedRandom and WeightedGreedy also underperform TacTok.

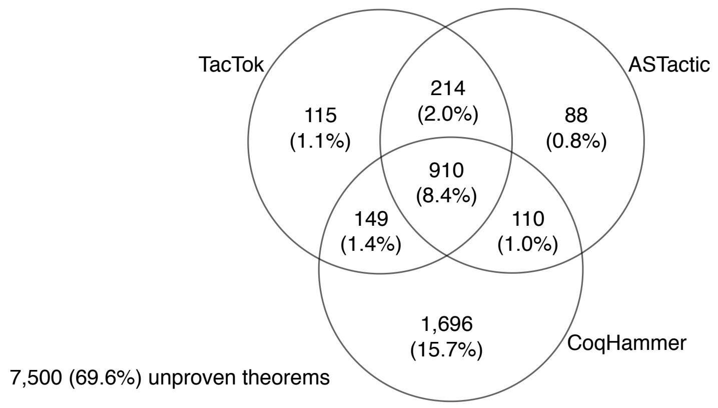


Figure 3.8: Theorems proven by TacTok, CoqHammer, and ASTactic out of the 10,782 theorems in CoqGym's test dataset. Figure 3.9 shows the by-project breakdown of the data. CoqHammer proves more theorems than TacTok and ASTactic, but they each prove different theorems. Together, they can prove more theorems overall.

3.5.3 RQ2: Does TacTok Improve CoqHammer's Verification Efficacy?

To evaluate whether TacTok improves automatic verification efficacy over other state-of-the-art tools, we compare TacTok to CoqHammer. We also compare to the combination of CoqHammer and ASTactic to understand if TacTok provides value beyond the combination of prior tools.

Figure 3.8 compares the number of theorems proven by TacTok, CoqHammer, and ASTactic. CoqHammer has a success rate of $\frac{2{,}865}{10{,}782} = 26.6\%$, which dominates the success rates of TacTok and ASTactic, 12.9% and 12.3%, respectively. The success rate of CoqHammer and ASTactic together is $\frac{3{,}167}{10{,}782} = 29.4\%$.

However, TacTok proves 115 theorems that CoqHammer and ASTactic fail to prove, so its added value compared to the combination of CoqHammer and ASTactic is $\frac{115}{3{,}167} = 3.6\%$.

Figure 3.9 shows the by-project breakdown of the success rates for each tool, while Figure 3.10 details the by-project value added of TacTok compared to CoqHammer and ASTactic.

success rate: project	TacTok	ASTactic & CoqHammer	ASTactic	CoqHammer	Tac	Tok	total theorems
weak-up-to	18 (12.9%)	36 (25.9%)	23 (16.5%)	30 (21.6%)	12 (8.6%)	12 (8.6%)	139
buchberger	76 (10.5%)	192 (26.5%)	70 (9.7%)	166 (22.9%)	70 (9.7%)	65 (9.0%)	725
jordan-curve-theorem	22 (3.5%)	168 (26.8%)	19 (3.0%)	165 (26.3%)	16 (2.5%)	22 (3.5%)	628
dblib	45 (25.0%)	67 (37.2%)	41 (22.8%)	55 (30.6%)	41 (22.8%)	35 (19.4%)	180
disel	89 (14.0%)	194 (30.6%)	83 (13.1%)	185 (29.2%)	63 (9.9%)	72 (11.4%)	634
zchinese	5 (11.6%)	13 (30.2%)	5 (11.6%)	12 (27.9%)	4 (9.3%)	4 (9.3%)	43
zfc	33 (13.9%)	70 (29.5%)	33 (13.9%)	64 (27.0%)	28 (11.8%)	28 (11.8%)	237
dep-map	11 (25.9%)	16 (37.2%)	9 (20.9%)	14 (32.6%)	7 (16.3%)	8 (18.6%)	43
chinese	35 (26.7%)	58 (44.3%)	31 (23.7%)	56 (42.7%)	27 (20.6%)	30 (22.9%)	131
UnifySL	180 (18.6%)	367 (37.9%)	189 (19.5%)	303 (31.3%)	126 (13.0%)	153 (15.8%)	968
hoare-tut	5 (27.8%)	6 (33.3%)	1 (5.5%)	6 (33.3%)	3 (16.7%)	3 (16.7%)	18
huffman	28 (8.9%)	81 (25.8%)	25 (7.9%)	74 (23.6%)	22 (7.0%)	21 (6.7%)	314
PolTac	112 (30.9%)	308 (84.9%)	118 (32.5%)	289 (79.6%)	87 (24.0%)	110 (30.3%)	363
angles	4 (6.5%)	15 (24.2%)	4 (6.5%)	15 (24.2%)	3 (4.8%)	4 (6.5%)	62
coq-procrastination	6 (75.0%)	5 (62.5%)	5 (62.5%)	3 (37.5%)	5 (62.5%)	6 (75.0%)	8
tree-automata	111 (13.4%)	311 (37.6%)	96 (11.6%)	292 (35.3%)	83 (10.0%)	96 (11.6%)	828
coquelicot	100 (6.8%)	299 (20.4%)	95 (6.5%)	273 (18.6%)	77 (5.2%)	78 (5.3%)	1,467
fermat4	10 (7.7%)	47 (36.2%)	13 (10.0%)	47 (36.2%)	9 (6.9%)	8 (6.2%)	130
demos	53 (77.9%)	55 (80.9%)	50 (73.5%)	54 (79.4%)	49 (72.1%)	52 (76.5%)	68
coqoban	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	2
goedel	67 (11.1%)	128 (21.1%)	53 (8.7%)	120 (19.8%)	57 (9.4%)	58 (9.6%)	606
verdi-raft	121 (5.7%)	351 (16.5%)	117 (5.5%)	337 (15.8%)	99 (4.7%)	97 (4.5%)	2,127
verdi	47 (8.2%)	127 (24.7%)	37 (7.2%)	122 (23.7%)	37 (7.2%)	42 (8.2%)	514
zorns-lemma	12 (8.1%)	21 (14.1%)	10 (6.7%)	18 (12.1%)	10 (6.7%)	7 (4.7%)	149
coqrel	183 (71.5%)	191 (74.6%)	184 (71.9%)	128 (50.0%)	163 (63.7%)	146 (57.0%)	256
fundamental-arithmetic	15 (10.6%)	41 (28.9%)	11 (7.8%)	37 (26.1%)	10 (7.0%)	9 (6.3%)	142
total	1,388 (12.9%)	3,167 (29.4%)	1,322 (12.3%)	2,865 (26.6%)	1,108 (10.3%)	1,166 (10.8%)	10,782

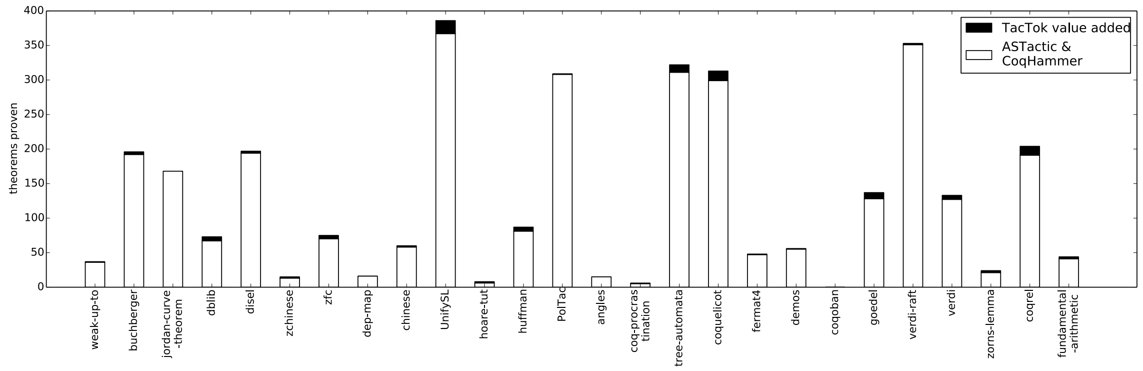


Figure 3.9: Success rates, as evaluated on the CoqGym benchmark [223], the three tools, TacTok, ASTactic [223], and CoqHammer [49], prove more theorems together than each does on its own. For 22 of the 26 subjects, TacTok proves theorems the other two tools cannot. Overall, together, TacTok proves 115 more theorems than the prior tools combined, a 3.6% relative improvement. Figure 3.10 demonstrates this improvement for each project.

For 22 of the 26 projects, TacTok has added value over the combination of CoqHammer and ASTactic. The greatest increase in the number of theorems proven in a project is 19, for UnifySL; the average increase is 4.4.

added value: project	TacTok over ASTactic & CoqHammer		TacTok over ASTactic		TacTok over CoqHammer		total theorems
	theorems proven	added value	theorems proven	added value	theorems proven	added value	
weak-up-to	1 (0.7%)	2.7%	2 (1.4%)	8.6%	3 (2.2%)	10.0%	139
buchberger	4 (0.6%)	2.1%	10 (1.4%)	14.3%	27 (3.7%)	16.3%	725
jordan-curve-theorem	0 (0.0%)	0.0%	5 (0.8%)	26.3%	1 (0.2%)	0.6%	628
dblib	6 (3.3%)	9.0%	11 (6.1%)	26.8%	11 (6.1%)	20.0%	180
disel	3 (0.5%)	1.5%	24 (3.9%)	28.9%	9 (1.4%)	4.9%	634
zchinese	2 (4.7%)	15.4%	2 (4.7%)	40.0%	2 (4.7%)	16.7%	43
zfc	5 (2.1%)	7.1%	5 (2.1%)	15.2%	6 (2.5%)	9.4%	237
dep-map	0 (0.0%)	0.0%	2 (4.7%)	22.2%	2 (4.7%)	14.3%	43
chinese	2 (1.5%)	3.4%	9 (6.9%)	29.0%	4 (3.1%)	7.1%	131
UnifySL	19 (2.0%)	5.2%	37 (3.8%)	19.6%	56 (5.8%)	18.5%	968
hoare-tut	2 (11.1%)	33.3%	4 (22.2%)	400.0%	2 (11.1%)	33.3%	18
huffman	6 (1.9%)	7.4%	8 (2.5%)	32.0%	11 (3.5%)	14.9%	314
PolTac	1 (0.3%)	0.3%	4 (1.1%)	3.4%	19 (5.2%)	6.6%	363
angles	0 (0.0%)	0.0%	0 (0.0%)	0.0%	0 (0.0%)	0.0%	62
coq-procrastination	1 (12.5%)	20.0%	1 (12.5%)	20.0%	3 (37.5%)	100.0%	8
tree-automata	11 (1.3%)	3.5%	30 (3.6%)	31.3%	24 (2.9%)	8.2%	828
coquelicot	14 (1.0%)	4.7%	28 (1.9%)	29.5%	33 (2.2%)	12.1%	1,467
fermat4	1 (0.8%)	2.1%	2 (1.5%)	15.4%	1 (0.8%)	2.1%	130
demos	1 (1.5%)	1.8%	4 (5.9%)	8.0%	2 (2.9%)	3.7%	68
coqoban	0 (0.0%)	0.0%	0 (0.0%)	0.0%	0 (0.0%)	0.0%	2
goedel	9 (1.5%)	7.0%	16 (2.6%)	30.2%	17 (2.8%)	14.2%	606
verdi-raft	2 (0.1%)	0.6%	19 (0.9%)	16.2%	11 (0.5%)	3.3%	2,127
verdi	6 (1.2%)	4.7%	16 (3.1%)	43.2%	9 (1.8%)	7.4%	514
zorns-lemma	3 (2.0%)	14.3%	3 (2.0%)	30.0%	6 (4.0%)	33.3%	149
coqrel	13 (5.1%)	6.8%	17 (6.6%)	9.2%	64 (25.0%)	50.0%	256
fundamental-arithmetic	3 (2.1%)	7.3%	5 (3.5%)	45.5%	6 (4.2%)	16.2%	142
total	115 (1.1%)	3.6%	264 (2.4%)	20.0%	329 (3.1%)	11.5%	10,782

Figure 3.10: TacTok added value, as evaluated on the CoqGym benchmark [223]. TacTok proves 115 theorems that both ASTactic [223] and CoqHammer [49] fail to prove, so its added value over the two tools combined is 3.6%. The added value of TacTok over ASTactic is 20.0%. While CoqHammer proves more theorems than TacTok, the added value of TacTok over CoqHammer is 11.5%.

Meanwhile, for 24 of the 26 projects, TacTok has added value over CoqHammer and ASTactic, individually. TacTok proves 264 theorems that CoqHammer fails to prove, so its added value compared to CoqHammer is $\frac{329}{2{,}865} = 11.5\%$ (and 20.0% over ASTactic, as discussed in Section 3.5.2).

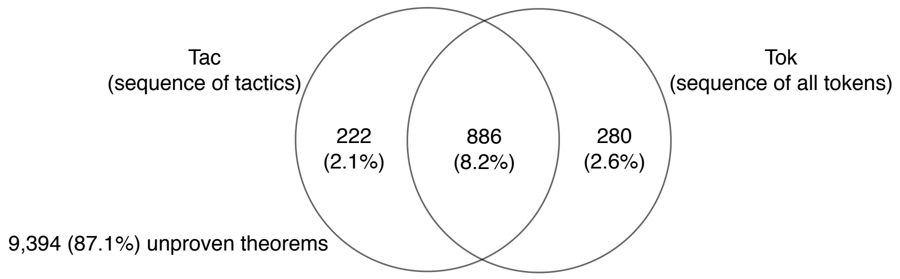


Figure 3.11: Theorems proven separately by TacTok's two ways of encoding partial proof scripts. Tac encodes a sequence of tactics, whereas Tok encodes all tokens, in addition to the tactics. The two ways of encoding complement each other in proving theorems.

RA2: While CoqHammer proves more theorems, TacTok can prove 11.5% additional theorems CoqHammer fails to prove. Of these 329 newly proven theorems, 115 cannot be proven by ASTactic, so TacTok adds 3.6% value beyond the two state-of-the-art tools combined.

3.5.4 RQ3: Do Non-Tactic Tokens Improve Verification Efficacy?

TacTok is comprised of two underlying models, Tac and Tok (recall Section 3.3.2.3). The two models both encode proof state the same way, but they differ in how they encode partial proof scripts. Tac models the partially written proof scripts by just the sequence of tactics used thus far. Tok, meanwhile, models the partially written proof scripts using all the proof tokens, including the tactics and their arguments. This research question focuses on understanding the contributions Tac and Tok make to TacTok.

Figure 3.11 compares the number of theorems proven by Tac and Tok. Tac proves 1,108 theorems, for a success rate of $\frac{1{,}108}{10{,}782} = 10.3\%$, while Tok has a success rate of $\frac{1{,}166}{10{,}782} = 10.8\%$. Figure 3.9 shows the by-project breakdown of the success rates for Tac and Tok.

Tac and Tok are clearly complementary. Tac outperforms Tok for 8 of the projects, while Tok outperforms Tac for 13 of the projects. Tac is able to prove 222 theorems that Tok fails to prove, so its value added compared to Tok is $\frac{222}{1{,}166} = 19.0\%$. Tok proves 280 theorems that Tac fails to prove, so its value added compared to Tac is $\frac{280}{1{,}108} = 25.3\%$. Their complementarity is the prime reason TacTok relies on both, improving its efficacy.

RA3: The data suggest that modeling the sequence of tactics in partial proof scripts is complementary to modeling all tokens. Each approach helps prove around a fifth of the theorems the other cannot. Some proof scripts appear to require the knowledge encoded in the tactic arguments of proof scripts, whereas others are able to be generated accurately (with the arguments) without encoding the arguments as an input to the model. Combining the two produces the highest verification efficacy.

3.5.5 Does TacTok Prove More Complex Theorems That Other Tools Do Not?

To understand if TacTok is able to prove more complex theorems than prior tools, we manually examined the theorems TacTok was able to prove that prior tools were unable to. We observed several examples of higher-order logic that TacTok proved, but prior tools could not.

One class of examples is theorems that have nested `forall` quantifiers. Figure 3.1 shows an example of such a theorem that TacTok is able to prove, but ASTactic and CoqHammer were not. Higher-order logic is more difficult for a programmer to reason through, and so the use of a proof script or proof synthesis tool is likely to be more helpful. However, it is not the case that TacTok is better at proving theorems with higher-order logic than other tools. Rather, TacTok can sometimes prove ones that they could not, making it complementary.

Another class of examples is theorems that require induction to prove. Figure 3.12 shows one example such a theorem. The theorem uses the $n^{th}\ l\ n\ a$ relation, which

```
1 Lemma Nth_app :
2    forall A (l : list A) l' a n,
3      Nth l n a ->
4      Nth (l ++ l') n a.
5
6 Proof.
7 intro. intro. intro. intro. intro. clear. intro.
8 induction H. constructor. constructor. assumption.
9 Qed.
```

Figure 3.12: Lines 1–4 defines the `Nthapp` theorem, and lines 6–9 is the proof script that TacTok generates for this theorem. TacTok correctly applies the `induction` tactic.

holds when a is the nth element of the list l. If so, the theorem states that it is also the element of a longer list that is prefixed by l. TacTok can generate the inductive proof script of this theorem, but prior tools cannot. CoqHammer does not try to prove theorems that require induction [49], and so CoqHammer is automatically at a disadvantage for proving this class of theorems. TacTok's modeling of proof state and partial proof script, together, was able to capture sufficient context to properly apply induction.

3.6 Contributions

This work is the first to explore the value of combining modeling of the proof state and of the partially written proof script to synthesize proof scripts, from scratch. We reify this modeling in TacTok, an open-source automated proof script synthesis technique. Evaluating TacTok against CoqHammer [49], ASTactic [223], and other metaheuristic search-based approaches we create, we find that TacTok is complementary to other tools: it successfully synthesizes proof scripts for theorems prior tools cannot for 24 out of the 26 projects on which we evaluate. With TacTok, 11.5% more theorems can be proven automatically than by CoqHammer alone, and 20.0% than by ASTactic alone. Compared to a combination of CoqHammer and ASTactic, TacTok can prove an additional 3.6% more theorems, proving 115 theorems no tool could previously prove.

Overall, our experiments provide evidence that partial proof script and proof state semantics, together, provide useful information for proof script modeling. We create a concrete approach for modeling the two types of information together, which can serve as a basis for further progress creating automatic verification tools.

CHAPTER 4
DIVA: DIVERSITY-DRIVEN AUTOMATED VERIFICATION

The work in this chapter was done in collaboration with Yuriy Brun. The following text is adapted directly from published work [59].

4.1 Introduction

Recent work has created tools that can fully automate some formal verification by synthesizing either the underlying proofs or the guiding proof scripts. Tools called *hammers* (e.g., CoqHammer [49]) use a set of precomputed mathematical facts to attempt to "hammer" out a proof. Evaluated on the CoqGym benchmark [223], CoqHammer can automatically prove 26.6% of theorems found in open-source Coq projects. But hammers are restricted by their precomputed facts and cannot reason about proof approaches such as induction, greatly limiting their power. To overcome these limitations, researchers have used machine learning to model existing proof scripts, and then, given a new theorem, applied that model to guide metaheuristic search [83] to attempt to synthesize a new proof script [223, 62, 179].

While these tools tend to prove fewer theorems, e.g., ASTactic proves 12.3% [223] and TacTok proves 12.9% [62], they are capable of applying higher-order proof approaches learned from existing proofs, including induction, and so are complementary to hammers. Together with CoqHammer, they prove 30.4% of the theorems. The central goal of this work is to improve on this fraction, particularly focusing on the tools that model existing proofs.

We make two key observations that enable us to improve the proving power of proof-script-synthesis techniques. First, the formal verification domain is a unique application of machine learning because it has a correctness oracle. In most machine learning applications, it is not known when the model is correct. This is why models are typically evaluated for precision or accuracy. In the formal verification domain, however, the interactive theorem prover can use a synthesized proof script to determine whether it truly proves the underlying theorem. If the prover can get to `Qed`, then the synthesized proof script must be correct. Thus, proof-script-synthesis systems always have a precision of 100%: they never return a failed script, instead continuing the search or timing out. While recall may be low, precision is always perfect. Second, variations in the models can alter the search-based synthesis of a proof script enough that two models can potentially produce different scripts for the same theorem. This, in turn, can, hypothetically, lead to models that prove complementary sets of theorems. And because of our first observation, they can be combined without sacrificing their power. The combined system can synthesize successful proof scripts for all theorems each one of the models can prove individually; if one model fails to synthesize a successful script, the theorem prover unequivocally tells us so, and we instead use the other model's successful script. Thus, if one can learn models that differ in a way to produce different scripts, potentially, this set of models may be able to prove far more theorems than a single model. The central question this work answers is whether model diversity can be created to improve the proving power of proof-script-synthesis techniques, and whether such an approach improves on the state-of-the-art automated formal verification techniques. We find that the answer to both questions is "yes."

As we will demonstrate on a benchmark of 68,501 theorems from 122 open-source software projects in Coq, we are able to create a set of 62 models by varying learning parameters and learning data that, together, prove 68% more theorems than TacTok and 77% more than ASTactic, despite using the same search method. Combining our approach, Diva, with CoqHammer [49], we can prove 33.8% of all the theorems, the highest such result to date. Diva proves 364 theorems that none of the prior tools have been able to prove. The difficulty of manually writing proof scripts for formal verification is so great, that even small improvements in proving power can be significant, and the savings in human effort that our approach represents are quite substantial.

Our insights enable for a completely new way to combine machine learning models. Of course, the idea of combining models is not new. Ensemble learning allows weighting the results of multiple models to improve the precision or recall of a single model [176]. And stacking uses a classifier to decide which model to apply to each input [55]. While both these methods can improve precision and recall in practice, they can also, hypothetically, reduce them, and often cannot properly amplify the correct results of a small minority of models. By contrast, in our domain, our method for combining models can never produce a wrong result or ignore the correct result produced by even a single model. This represents a killer app for ensemble learning and stacking. We are the first to combine the idea of ensemble learning with an oracle to produce optimal stacking.

This work explores nine dimensions for learning diverse models, and identifies which dimensions lead to the most useful diversity. Altering the types of information (the proof script, state, and term) the model learns from resulted in the greatest diversity, while varying the depth of the proof script and the learning rate provided the second most diversity. As running a large number of models can be inefficient, we develop a model interrupts optimization that speeds up Diva's execution by 40×.

The main contributions of our work are:

- A novel approach for combining varied machine learning models to formally verify software properties.

- A systematic exploration of which learning dimensions provide usable model diversity.
- An implementation of our approach, Diva, that proves 68% more theorems than TacTok and 77% more than ASTactic, the prior work most closely related to ours. Diva is open-source and is available at `https://github.com/LASER-UMASS/Diva/`.
- An optimization for improving Diva's performance.
- A platform for evaluating models and rerunning experiments, and all data and source code used in our experiments for replications [60].

4.2 Proof script modeling

Prior proof script synthesis tools, such as ASTactic and TacTok, use the predictions from learned proof script models to bias the metaheuristic search for a proof script. Such a model is learned from a set of existing, successful proof scripts to predict the next proof step (tactic and arguments) of an incomplete proof script. There are three relevant aspects of proof scripts we may want to encode to serve as input to such a model: the proof state, the proof script, and the Gallina proof term. ASTactic only encodes the proof state, while TacTok encodes both the proof state and the proof script. There has yet to be a proof script synthesis tool that encodes the Gallina proof term. Next, Sections 4.2.1, 4.2.2, and 4.2.3 describe how to encode the proof state, proof script, and Gallina proof term, respectively.

4.2.1 Encoding the proof state

The proof state consists of the goals to be proven, local context, and the environment. While the programmer sees them in a human-readable format, each term of the proof state has an underlying abstract syntax tree (AST) representation. ASTactic and TacTok serialize these ASTs and encode them using a neural model, specifically a TreeLSTM [198]. Prior work has empirically argued that neural models are much more effective than other architectures [18, 136, 195].

4.2.2 Encoding proof script features

The proof script is comprised of a sequence of tokens in Ltac. For the model to encode these tokens, each proof script needs to be preprocessed to remove high-frequency low-signal tokens, such as punctuation. Then, encoding such a sequence is traditionally done using a language model [18, 136, 195]. Language models are widely used in natural language processing tasks [12, 190]. The primary function of a language model is to predict the next token in a sequence of tokens. While prior work has used n-grams to model Coq [86], TacTok found neural language models work better to encode the sequence of tokens because it can generate a representative vector (embedding) for the sequence, that can then be combined with other types of inputs. Among the most extensively used neural language models are transformers [54] and RNNs [157]. TacTok uses an RNN (specifically a Bidirectional LSTM [157]) trained from scratch. Transformers require massive amounts of data to train from scratch [54].

4.2.3 Encoding the proof term

Prior tools have not encoded proof terms, but, conceptually, the Gallina sequence is similar to the proof script Ltac sequence, and we encode it in a similar way using a Bidirectional LSTM [157]. This allows all three, the proof state, proof script, and proof term, to be encoded with a single model.

4.3 Diva: Diversity-Driven Synthesis

Machine learning models can be sensitive to noise in the training data [144, 205] and to parameters applied during the learning process [72]. This sensitivity can cause great variability in the accuracy of models. Of course, this can hurt the generalizability

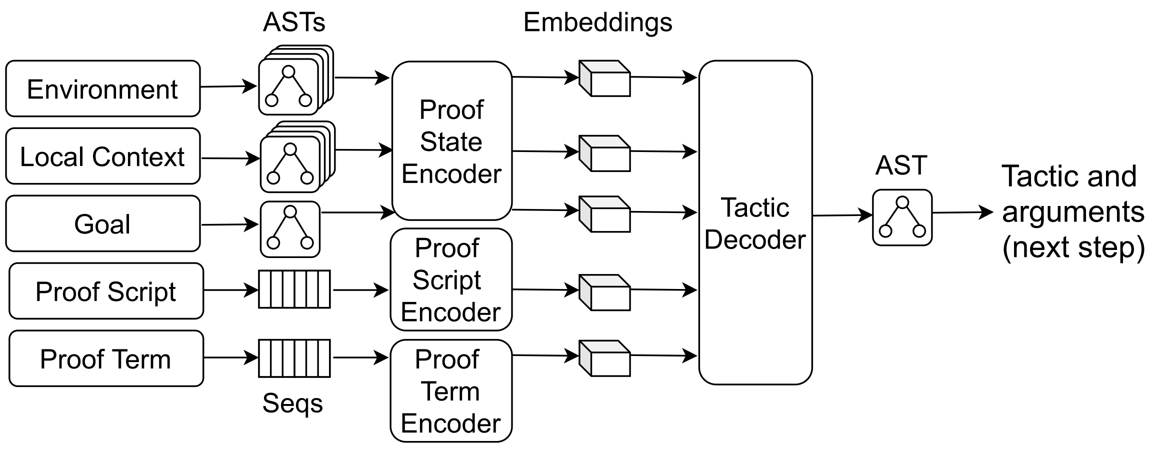


Figure 4.1: Model of proof script, which Diva uses internally to drive search.

of machine learning results, but we posit that in the right domain, this sensitivity, and the diversity of models it can produce, can provide a significant benefit.

In the formal verification domain, tools such as ASTactic [223], Proverbot9001 [179], and TacTok [62] use a learned model of a proof script to guide metaheuristic search toward synthesizing a proof script for a theorem. Variations in the models can alter the search, resulting in potentially different attempted synthesized scripts. The key uniqueness of this domain is that an interactive theorem prover can act as an oracle for each proof script. If the proof script leads the theorem prover to generate a proof terminating in `Qed`, then the proof script is, by definition, correct. This allows a synthesis tool to try applying many different models to bias the search in different ways, and then pick out just the successful synthesis attempts, discarding the failed ones.

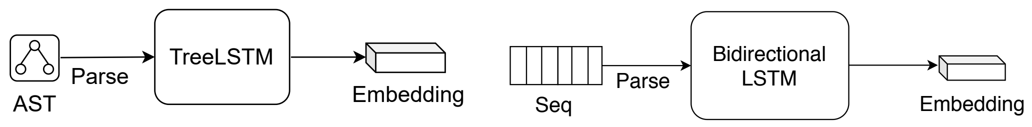
This is not the typical case in applications of machine learning. Ensemble learning [176] and stacking [55] attempt to combine the results of multiple machine learning models to improve precision or recall. However, without an oracle, ensembles and stacks are unlikely to always pick the correct result, especially when relatively few of the diverse models produce it. By contrast, in our domain, with the theorem prover acting as an oracle, even a single model producing a correct proof script can establish an answer.

To demonstrate this insight, we develop Diva, a proof-script-synthesis tool that uses the diversity in machine learning to significantly improve its proving power. Diva is open-source and is available at `https://github.com/LASER-UMASS/Diva/`.

Diva's key contributions are the generation of a diverse set of models capable of proving complementary sets of theorems, a mechanism for combining the benefits of the models, and an optimization to make running a large number of searches using independent models feasible.

To automate proof script synthesis, Diva uses a learned model of a proof script to guide metaheuristic depth-first search. During this search, Diva samples a fixed number of the most likely tactics, predicted by the model, across all search tree nodes at the same level, and then uses these tactics to search for a complete proof script. Diva backtracks when the Coq compiler fails to check the attempted proof script step or detects a duplicate proof state. Diva uses the same search configuration (width of 20, search depth limit of 5, and a timeout of 10 minutes) as ASTactic and TacTok.

To intentionally produce a diverse set of models that prove complementary sets of theorems, control over the learning process is key. When training a model of proof scripts, Diva varies the learning parameters and which features of the training data to encode. Next, Section 4.3.1 describes what a Diva model looks like; Sections 4.3.2 and 4.3.3 detail how Diva generates a diverse set of models by controlling learning parameters and the encoded features of the training data, respectively; and Section 4.3.4 explains our Diva efficiency optimization.



(a) Proof state encoder (b) Proof script and term encoders

Figure 4.2: The neural models used to encode the proof state AST, proof script sequence, and the proof term sequence.

4.3.1 Diva's learned model

Figure 4.1 illustrates Diva's proof script model, learned from a set of existing proof scripts. Diva uses the predictions from this model to drive the search for a complete proof script.

Figure 4.2 details the encoders used in the Diva model to encode relevant aspects of proof scripts. Figure 4.2a presents the proof state encoder, which Diva uses to encode the goal, local context, and environment, in AST form. To encode a tree, it uses a TreeLSTM network [44], which generates embeddings for each proof state term. Figure 4.2b shows the proof script encoder, which Diva uses to encode the proof script sequence. We encode the parsed sequence of previous tokens using a Bidirectional LSTM, which generates an embedding for the sequence. A Bidirectional LSTM improves on the LSTM by capturing more contextual information by processing the input sequence in two ways, forward and backward [157], allowing the output layer to simultaneously see both directions of information. Diva encodes the Gallina proof term (the first synthesis tool to do this) using the same encoder in Figure 4.2b. Similar to the proof script sequence encoding, we choose to encode the sequence of proof term tokens using a Bidirectional LSTM, generating an embedding. Diva jointly learns embeddings for the sequences and ASTs.

Diva's tactic decoder is modified from the tactic decoder first used in ASTactic and, later, TacTok. This tactic decoder is conditioned on the sequence of embeddings. In Diva, however, the embeddings are a concatenation of a subset of the embeddings generated from the proof script, proof term, and proof state encoders. This allows for modeling of more relevant proof script aspects and the choice of which subset to combine allows us to create variability in the models (see Section 4.3.3). The tactic decoder then generates a tactic by sequentially growing an AST [226]. It chooses a production rule from the context free grammar of the tactic space at a non-terminal node in the AST, while it synthesizes arguments based on semantic constraints at a terminal node. A GRU [42, 44] controls this process of growing the tree, as it updates its hidden state using the input embeddings of the partially generated AST.

Diva trains the model on a set of existing proof scripts. Each proof script in this set is broken down into training instances, which are the inputs to the model. A training instance is comprised of the proof state before the tactic execution, the proof script up to the tactic execution, the Gallina proof term before the tactic execution, and the next step of the proof script. The Diva model jointly learns embeddings for the proof state ASTs, the proof script, and proof term sequence, and then uses these embeddings to predict the next proof script step in the form of an AST. The model sends the predicted AST along with ground-truth next tactic AST to the trainer, where the trainer compares these tactic ASTs and back-propagates the loss.

Unlike prior tools, Diva jointly trains a language model over the tokens in the proof term. Section 4.3.2 details further modifications in this training process for creating Diva's diverse models.

4.3.2 Diversity via varying learning parameters

One way in which we create a diverse set of models is by varying the learning parameters, which affects the model's size and the learning algorithm itself. For this, we start with the Tac model from TacTok, and explore varying six dimensions: sequence tactic depth, sequence token depth, the learning rate, the embedding size, the number of layers, and the order of the training data.

Tactic and token sequence depth The sequence depth denotes the size of the input the learning algorithm considers. When training, the model can consider the entire proof script written so far, or part of it, such as only the most recent tactic and its arguments, or several most recent tactics with arguments, or only several most recent tokens. The proof script encoder considers only that portion of the proof script (and, symmetrically, the decoder will consider the same depth when decoding the next proof step). Diva varies the sequence depth along both tactics and tokens, from a depth of 0, which does not consider the proof script at all (it considers only proof state, making the model equivalent to ASTactic's model), to the entire proof script. Diva considers sequence depth sizes (excluding the start token) of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 29 and 30.

Learning rate During training, the algorithm updates the model's weights in every iteration. The learning rate is a hyperparameter that determines how much the weights can be changed in each iteration. A larger learning rate is less likely to result in the training getting stuck in a local optimum, but may also take longer to converge or fail to explore a region long enough to find an optimal solution. Accordingly, the models produced by varying the rate can be quite different. Diva considers learning rates of 3×10^k for $k \in \{-2, -3, -4, -5, -6, -7, -8\}$.

Model size (embedding and layers) The model's size is defined by two hyperparameters, the number of model layers and embedding size, which is the size the vector space in which a proof aspect is embedded. Diva varies the proof script encoder size by trying 1, 2, 3, 4, and 5 layers and embedding sizes of 64, 128, 256, and 512.

Training data order The order of the training data can affect the model [72]. We vary the order in which Diva sees the training instances by creating ten random orders.

4.3.3 Diversity via varying training data

The second way in which we create a diverse set of models is by varying aspects of the training data available to the learning algorithm. There are three types of data in the training proof scripts: the proof state, the proof script tactics and tokens, and the proof term that the proof assistant generates when it executes the proof script (recall Section 4.2). When training a model, we either include each of these three types of data or we exclude them. For the proof script, we include either the tactics or the tokens, since they encode fundamentally the same information. This leads us to a total of 11 models. For the models that do not include proof state, when we encode the training instance that represents the very start of proof-script synthesis, we include the theorem being proven (otherwise the model would not know what it is trying to prove). Similarly, at test time, when synthesizing the first proof script step, we include the theorem being proven.

4.3.4 Efficiently combining model executions

Executing a large set of models in sequence is slow since Diva has to wait for a model to finish its proof script synthesis attempt before it can try the next one. We develop model interrupts to improve Diva's efficiency. In model interrupts, given a set of models, Diva assigns an arbitrary order of model application. In order, each model will be given a specified amount of time to try to synthesize a proof script. Once the time runs out, the next model attempts to synthesize a proof script from scratch. Figure 4.3 illustrates this concept. The first model attempts synthesis from scratch for X seconds, at which point, if a complete proof script is not generated, the partial proof script is stored and the second model attempts to synthesize a proof script from scratch for X seconds. And so on. Once each model is given an opportunity to try for X seconds and a complete proof script is not found, the models will be given

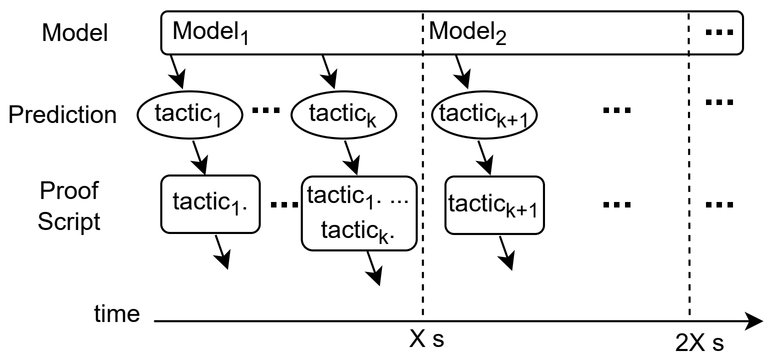


Figure 4.3: Model interrupts allows Diva to let models take turns synthesizing proof scripts from scratch.

more time to synthesize a proof script starting from the stored partial proof script associated with the model.

4.4 Evaluation

We evaluate Diva to measure how much diversity of models of proof scripts can increase the effectiveness of proof script generation. We follow the methodologies of prior evaluations of proof-script synthesis tools [62, 223], in terms of the dataset (Section 4.4.1.1) and metrics (Section 4.4.1.2) used; we compare to two state-of-the-art proof-script-synthesis tools, ASTactic [223] and TacTok [62], which use the same metaheuristic search for proof-script synthesis as Diva. We further compare Diva to the state-of-the-art proof-synthesis tool CoqHammer [49].

Our evaluation answers four research questions:

RQ1: Does diverse-modeling significantly improve proof-script synthesis over state-of-the-art approaches CoqHammer, ASTactic, and TacTok?

RQ2: How much model diversity results from varying the model learning parameters sequence depth, learning rate, number of layers, size of embeddings, and training order, and how does this model diversity affect proof script synthesis effectiveness?

RQ3: How much model diversity results from varying which aspects of the training proofs — tactics, tokens, proof state, Gallina proof terms — are available to the learning process and how does this model diversity affect proof script synthesis effectiveness?

RQ4: How effective is our interrupts mechanism for improving Diva efficiency?

All of our evaluation data and the source code to reproduce our results are available [60].

4.4.1 Evaluation methodology

We first describe the dataset and metrics we use to evaluate Diva.

4.4.1.1 Dataset

In our evaluation, we use CoqGym [223], the state-of-the-art benchmark used in prior evaluations of formal verification tools [223, 62, 126]. The benchmark consists of 70,856 theorems from 123 open-source software projects in Coq. The CoqGym benchmark comes with a preselected training set of 96 projects with 57,719 human-written proof scripts, and test set of the remaining 13,137 theorems from 27 projects.

Our earlier TacTok evaluation [62] was unable to reproduce prior results for ASTactic's performance [223] for one project, coq-library-undecidability, due to internal Coq errors when processing the proof scripts. Accordingly, we exclude this project from our evaluation. We were able to reproduce the results for the remaining 26 projects of 10,782 theorems. In total, our training and test sets have 68,501 theorems from 122 projects.

4.4.1.2 Metrics

We measure four quantities in answering our research questions: success rate, added value, diversity, and mean time to prove a theorem.

Success Rate. The success rate of a tool, widely used in prior evaluations [92, 223, 62], is the fraction of all theorems for which the tool generates a succesful proof script.

Added Value. The added value of tool A over tool B is the number of new theorems tool A proves that tool B does not, divided by the number of theorems tool B proves.

Diversity. Given a set of models, we wish to know how much diversity they yield with respect to their ability to prove theorems. And so, we think of the diversity of a set of models as the diversity of the corresponding sets of theorems that the models prove. Our goal with the diversity measure is to be able to compare how much diversity results from various methods for creating models, so that we can compare the different methods.

Informally, given a set of sets of objects (theorems) we define a family of diversity functions, such that the k diversity function, d_k, measures the relative increase in objects contained in k sets, as compared to $k-1$ sets. So, for example, for a set of models, d_5 denotes the fraction of the additional theorems (out of all the theorems proved by at least one model) that are able to be proved by adding a fifth model to a set of four models, on average.

More formally, let T be a set of objects and let M be a set of subsets of T such that the union of all sets in M is equal to T. Then, for each $k \in \{1, 2, 3, \ldots, |M|\}$, the k diversity function $d_k \colon 2^T \to \mathbb{R}$ is the average increase, in terms of the fraction of T, that the union of k elements of M contains over the union of $k-1$ elements of M. Thus, for all $M_k \subseteq M$, such that $|M_k| = k$, and for all $M_{k-1} \subseteq M$, such that $|M_{k-1}| = k-1$, $d_k(M)$ is the average value of $\frac{|M_k \setminus M_{k-1}|}{|T|}$.

tool	theorems proven	Diva's value added
ASTactic	1,322 (12.3%)	908 (76.9%)
TacTok	1,388 (12.9%)	842 (68.4%)
CoqHammer	2,865 (26.6%)	781 (27.3%)
all 3 prior tools	3,282 (30.4%)	364 (11.1%)
Diva	2,338 (21.7%)	—
Diva & CoqHammer	3,646 (33.8%)	—

Figure 4.4: Theorems proven by and the success rate of Diva, ASTactic, TacTok, CoqHammer, and the combination of these tools out of the 10,782 theorems in CoqGym's test dataset. Diva provides value added over each of these tools, and 11.1% value added over the combination of all three.

Given a set of models, we compute the diversity functions empirically. We use each model to attempt to synthesize proof scripts to prove theorems. We then compute T, the set of all theorems that can be proven by at least one model. Then, to compute d_k, we, for each model, compute how many *additional* theorems it proves compared to each set of $k-1$ models. We then compute the average of those numbers, and divide it by $|T|$ for normalization. In the end, $d_k(M)$ is the average fraction of theorems proven by adding a k model to a set of $k-1$ models. Note that the sum of d_k for all k is 1, and that diversity is monotonically non-increasing with respect to k (that is, $d_{k-1} \leq d_k$.)

Mean Time to Prove a Theorem. To measure efficiency, we compute the mean time it takes to generate a proof script for a theorem, averaged over all the theorems for which we produce a successful proof script, and over all the possible orderings of the models used in the metaheuristic search.

4.4.2 RQ1: Does diversity help Diva outperform the state-of-the-art?

We created models by varying learning parameters and aspects of proof scripts to encode (recall the models described in Sections 4.3.2 and 4.3.3). Overall, we generated these 62 models for Diva to use.

We compare Diva to the state-of-the-art synthesis tools, ASTactic [223], TacTok [62], and CoqHammer [49]. ASTactic and TacTok, like Diva, learn from existing proof scripts to predict the next step of the proof script. CoqHammer uses a fundamentally different approach. Whereas CoqHammer produces proofs in Coq's logic (Gallina), Diva searches the proof-script space. When the Coq compiler executes a proof script, it generates a proof. Proofs cannot be wrong, while proof scripts can be (e.g., a proof script that concludes with *Proof completed*, may not lead to a valid proof when it is checked by the Coq compiler). Thus, it is reasonable to compare proof script synthesis tools, such as Diva, to CoqHammer with respect to the theorems they are able to prove. However, since their approaches are so fundamentally different, it is expected that these tools are likely to be complementary, performing well for different theorems. While CoqHammer and Diva are likely to perform similarly well for some simpler classes of theorems, CoqHammer is at a fundamental disadvantage, though, for other classes of theorems, such as ones that require induction to prove.

On our evaluation set of 10,782 theorems, ASTactic proves 1,322 (12.3%) and TacTok proves 1,388 (12.9%) theorems. CoqHammer proves 2,865 (26.6%) theorems. Prior to performing our evaluation, we expected that Diva would prove strictly more theorems than ASTactic and TacTok (though how many more remained an important question), that it would not prove more theorems than CoqHammer (given that CoqHammer proves much more than ASTactic or TacTok can prove alone), but that it would prove some complementary theorems, thus providing significant added value compared to CoqHammer, as was the case in ASTactic and TacTok evaluations [62, 223].

Figure 4.4 shows the success rates, as well as the raw number of theorems proven by the four tools, and the value Diva adds over each tool, as well as their combination. Diva proves 2,338 (21.7%) of the theorems. This means Diva proves $\frac{2{,}338-1{,}322}{1{,}322} = 76.9\%$ more theorems than ASTactic and $\frac{2{,}338-1{,}388}{1{,}388} = 68.4\%$ more theorems than TacTok.

Since these tools use the same search mechanism, these significant improvements are due entirely to the use of model diversity.

While CoqHammer proves more theorems than Diva, Diva proves 781 theorems that CoqHammer does not, an added value of $\frac{781}{2{,}865} = 27.3\%$. Figure 4.5 shows a Venn diagram of the theorems Diva, ASTactic, TacTok, and CoqHammer prove. Together, these four tools prove 3,646 theorems, for a success rate of 33.8%, whereas without Diva, the other three tools prove 3,282 theorems. (Because ASTactic and TacTok have an added value of 0% over Diva, CoqHammer and Diva prove the 3,282 theorems on their own, without the other tools' help.) Diva adds a value of 11.1% over the combined state of the art, and proves 364 theorems no tool has previously proven.

RA1: Our Diva diversity mechanisms are successful in creating model diversity sufficient to significantly improve the proving power of metaheuristic-search-based tools (68%–77% added value). Diva also generates 27.3% added value over CoqHammer, and proves 364 theorems no prior tool has proven. Together with CoqHammer, Diva reaches a new milestone, proving over one third of all theorems completely automatically.

4.4.3 RQ2: Learning-parameter diversity

To investigate the effectiveness of varying learning parameters on generating diverse models, we conduct a series of experiments by generating models varying those parameters, using the resulting models to synthesize proof scripts, and then measuring the *diversity* of the sets of theorems the models prove. As Section 4.3.2 described, the factors we investigate are sequence depth, learning rate, number of layers, embedding size, and training order. Figure 4.6 details how much diversity Diva produces by varying learning parameters in training its models.

Tactic depth diversity. We vary the tactic sequence depth, considering depths of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 29 and 30; a total of 16 models. (Note that

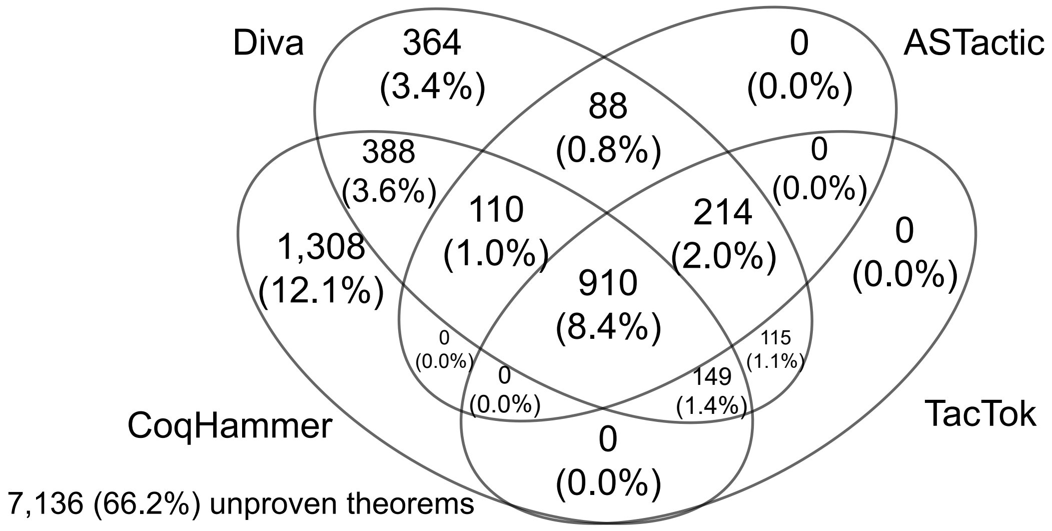
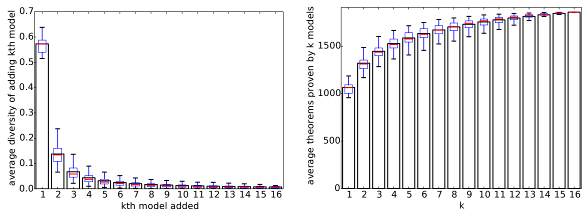
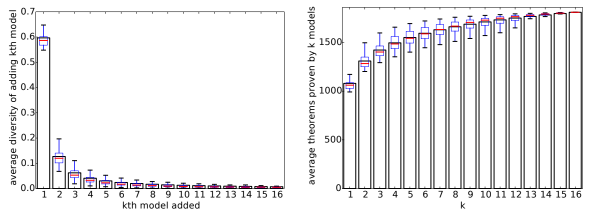
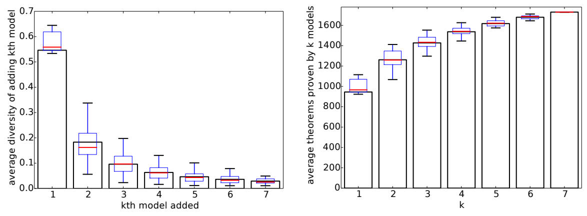


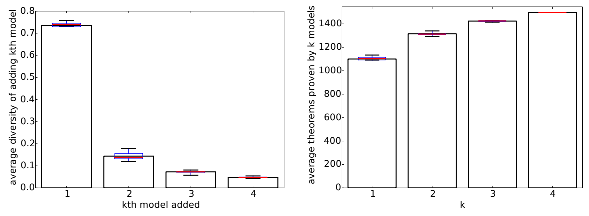
Figure 4.5: The breakdown of how many theorems are proven by each combination of tools. Diva proves 364 theorems no other tool proves.

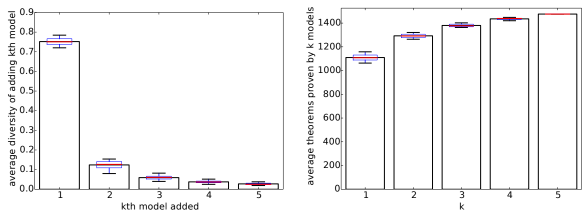
the depth 0 model is equivalent to ASTactic, and the depth 3 model is equivalent to the Tac model in TacTok.) Overall these 16 models prove 1,858 theorems, whereas on average, a single model proves 1,064 theorems. Diva's diversity is responsible for a 74.6% increase in proving power! The left graph in Figure 4.6a shows the diversity of the set of tactic sequence depth models (recall the diversity metric from Section 4.4.1.2). The k bar shows d_k for the 16 models. That is, the k bar states the fraction of *extra* theorems proven by k random models, that a random set of $k-1$ models does not prove. For example, the $k = 1$ bar is simply the effectiveness of using a single model, 0.573 (on average, 57.3% of the theorems proven by all models together are proven by using one random model). The remaining 42.7% need Diva's diversity mechanism. For $k = 2$, the diversity is 0.138, meaning that adding the second model, on average, adds an additional 13.8% of the total theorems proven. Two randomly chosen models prove, on average, 57.3% + 13.8% = 71.1% of all the theorems proven by at least


(a) Tactic sequence depth (total 1,858 theorems)


(b) Token sequence depth (total 1,810 theorems)


(c) Learning rate (total 1,730 theorems)


(d) Embedding size (total 1,496 theorems)


(e) Number of layers (total 1,476 theorems)

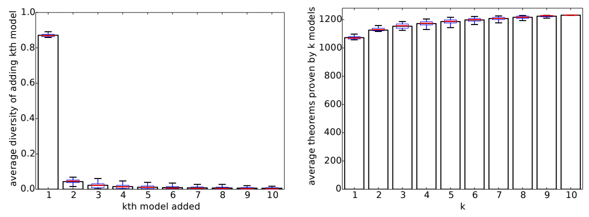

(f) Training data order (total 1,232 theorems)

Figure 4.6: The diversity exhibited by altering learning parameters tactic sequence depth (a), token sequence depth (b), learning rate (c), embedding size (d), number of layers (e), and training data order (f). The left graph in each pair shows the diversity measure, as a function of the number of models (e.g., the $k = 5$ bar is the mean fraction of additional theorems proven by picking a random 5 model that a random disjoint set of 4 models has not proven). The right graph in each pair shows the mean number of theorems proven by k models. The box-and-whiskers indicate the maximum, 75%-, 50%-, and 25%-tiles, and minimum values.

one model. The right graph in Figure 4.6a shows the average number of theorems that k of the tactic sequence depth models prove. The box-and-whiskers indicate the variability in the choice: how important is it to select specific k models, or can they simply be selected at random. For example, a single model can prove between 957 (8.9%) and 1,322 (12.3%) theorems from the test set. We leave developing mechanisms for selecting models to future work.

Token depth diversity. Similar to tactic depth, we considered token depths of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 29 and 30; a total of 16 models. (Note that the depth 0 model is, again, equivalent to ASTactic, and the depth 30 model is equivalent to the Tok model in TacTok.) Overall these 16 models prove 1,810 theorems (slightly fewer than tactic depth diversity models did), whereas on average, a single model proves 1,080 theorems. Diva's diversity is responsible for a 67.6% increase in proving power. The left graph in Figure 4.6b shows the diversity of the token depth models. A single random model proves a slightly larger fraction, 59.7%, of all the proven theorems than was the case for tactic depth models, indicating again that token depth provides slightly less useful diversity. Still, the remaining 40.3% of the theorems require Diva's diversity to be proven. The right graph in Figure 4.6b shows the variability in a selected k models. Here, a single model can prove between 992 (9.2%) and 1,322 (12.3%) theorems from the test set. Overall, token depth provides significant diversity, but less than tactic depth did.

Learning rate. We explore 7 different learning rates: 3×10^k for $k \in \{-2, -3, -4, -5, -6, -7, -8\}$. Overall these 7 models prove 1,730 theorems (slightly fewer than the depth diversity models did), whereas on average, a single model proves 945 theorems. Diva's diversity is responsible for a 83.1% increase in proving power. The left graph in Figure 4.6c shows the diversity of the learning rate models. A single random model proves 54.6% of all the proven theorems. The remaining 45.4% of the theorems require Diva's diversity to be proven. The right graph in Figure 4.6c shows the variability in a selected k models. Here, a single model can prove between 505 (4.7%) and 1,115 (10.3%) theorems from the test set. Overall, learning rate provides significant diversity, and the models are more diverse from one another than the sequence depth models, but, overall, result in slightly less proving power.

Embedding size. We explore 4 different embedding sizes: 64, 128, 256, 512. Overall these 4 models prove 1,496 theorems (fewer than the already discussed models), whereas on average, a single model proves 1,100 theorems. Diva's diversity is responsible for a 36.0% increase in proving power. The left graph in Figure 4.6d shows the diversity of the embedding size models. A single random model proves 73.5% of all the proven theorems. The remaining 26.5% of the theorems require Diva's diversity to be proven. The right graph in Figure 4.6d shows the variability in a selected k models. Here, a single model can prove between 1,056 (9.8%) and 1,134 (10.5%) theorems from the test set. Overall, embedding size provides some diversity, though less than sequence depth and learning rate.

Number of layers. We explore 5 different numbers of layers: 1, 2, 3, 4, and 5. Overall these 5 models prove 1,476 theorems (similar to the embedding size), whereas on average, a single model proves 1,109 theorems. Diva's diversity is responsible for a 33.1% increase in proving power. The left graph in Figure 4.6e shows the diversity of the number of layers models. A single random model proves 75.2% of all the proven theorems. The remaining 24.8% of the theorems require Diva's diversity to be proven. The right graph in Figure 4.6e shows the variability in a selected k models. Here, a single model can prove between 1,063 (9.9%) and 1,158 (10.5%) theorems from the test set. Overall, varying the number of layers provides a similar amount of diversity as embedding size. Both parameters effect the size of the learned model.

Training data order. We explore 10 randomly chosen orderings of the training data. Overall these 10 models prove 1,232 theorems, the smallest number of all the learning parameters, whereas on average, a single model proves 1,073 theorems. Diva's diversity is responsible for a 14.8% increase in proving power. The left graph in Figure 4.6f shows the diversity of the training data order models. A single random model proves 87.1% of all the proven theorems. The remaining 12.9% of the theorems require Diva's diversity to be proven. The right graph in Figure 4.6f shows the variability in a selected k models. Here, a single model can prove between 1,058 (9.8%) and 1,098 (10.2%) theorems from the test set. Overall, even just varying the training data order provided some useful diversity and enabled proving more theorems, though the diversity benefits were much smaller than those of the other parameters.

RA2: Varying learning parameters resulted in significant diversity, which, in turn, led to significant improvement in proving power. Varying the depth of the tactics and tokens the model learned from and the learning rate led to the greatest diversity, while varying the size of the model led to moderate diversity. Varying the order of the training data marginally increased the proving power.

4.4.4 RQ3: Training-data diversity

Recall from Section 4.3.3 that there are three types of data in the training proof scripts: the proof state, the proof script tactics and tokens, and the Gallina proof term. We train models for all possible combinations of these data types, except no model includes both tactics and tokens, and we exclude the model that is the empty combination. In total, we learn 11 models.

We first measure the value added by adding each of the three types of information. The value of adding proof script tactics to a model already encoding the proof state and the Gallina proof term is 134.2%, proving an additional 345 theorems. (The value of adding proof script tokens instead of tactics is similar, 136.6%, 351 theorems). The value of adding Gallina proof term to a model already encoding the proof script and the proof state is much smaller, 8.0%, proving an additional 89 theorems. (If using tokens instead of tactics, the added value is 10.6%, 124 theorems.) Finally, the value of adding proof state to a model already encoding the proof script and the Gallina proof term is 21.8%, proving an additional 135 theorems. (If using tokens instead of tactics, the added value is 53.5%, 281 theorems. We observe that while in previous scenarios, tactics and tokens behaved similarly, here, tokens exhibit much more diversity than tactics.) In all three cases, tokens exhibited greater diversity than tactics in encoding the proof script, suggesting that tokens are a more different representation than tactics of the other types of information. The Gallina proof term contained the least diversity compared to the other types of data, whereas the proof script contained the most.

Overall, these 11 models prove 2,053 theorems, which is significantly more than any of the learning parameter models from Section 4.4.3. A single model, on average, proves 785 theorems. Diva's diversity is responsible for a 161.5% increase in proving power! The left graph in Figure 4.7 shows the diversity of the training-data-types models. A single random model proves 38.3% of all the proven theorems. The remaining 61.7% of the theorems require Diva's diversity to be proven. The right graph in Figure 4.7 shows the variability in a selected k models. Here, a single model can prove between 257 (2.4%) and 1,322 (12.3%) theorems from the test set. Overall, training data types provide the most diversity of all the dimensions we explored, leading to the greatest proving power.

RA3: Including different data types in training resulted in the most diversity of all the dimensions we considered, leading to the greatest proving power increase. Adding proof script tactics or tokens provided the most diversity, followed by the proof state.

4.4.5 RQ4: Synthesis efficiency

To explore improving Diva's efficiency, we implement model interrupts (described in Section 4.3.4). We evaluate the efficiency improvement of model interrupts by

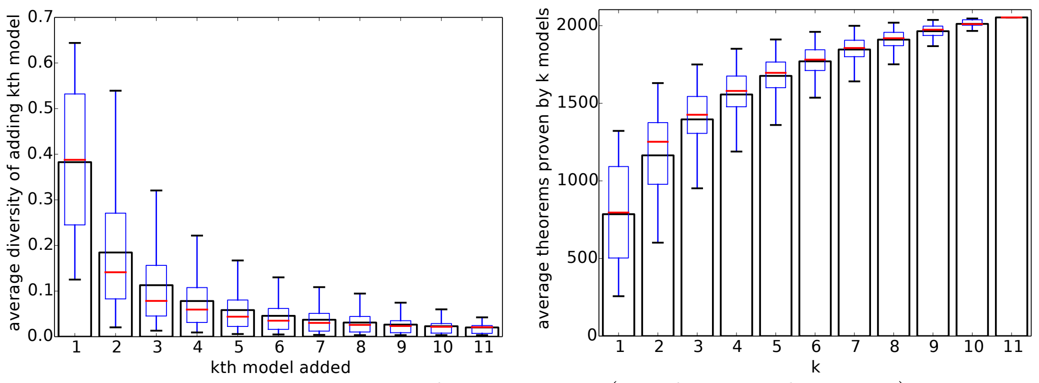


Figure 4.7: Training data aspects (total 2,053 theorems)

measuring the mean time to prove a theorem with and without interrupts. Of course, the order in which Diva considers the models matters. Without interrupts, in the worst case, the last model produces the successful proof script, and Diva wastes 10 minutes on each of the other models, before they time out. For our evaluation, we measure the mean time over a random sample of 20 possible model orderings.

Without interrupts, the mean time to prove a theorem is 685.5 seconds. However, we observe that most models either synthesize the proof script relatively quickly, or don't at all, though with some notable exceptions. Using model interrupts allows us to benefit from proving theorems quickly in the initial burst of each model, without spending the long time in the tail of each model's distribution, unless it is necessary. With model interrupts, we explore 15 different switching times: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, and 60 seconds.

We explore two different interrupt schemes. First, we attempt to synthesize a proof script using each model for X seconds. If none of the models find a proof script in that time, we return and give each model another X seconds. And so on, until each model has attempted its search for 10 minutes. The left graph in Figure 4.8 shows the mean time to prove a theorem for this interrupt scheme. For $X = 1$ second, this interrupt scheme achieves the minimal mean time to prove a theorem of 17.2 seconds, and the proving time increases monotonically for larger X. For $X = 1$, the speed up compared to not using interrupts is 97%, or 40×. This suggests that many theorems are proven very early in the synthesis process, and while some theorems do get proven after a lengthy synthesis search, prioritizing the first seconds of synthesis using the diverse models greatly improves synthesis efficiency.

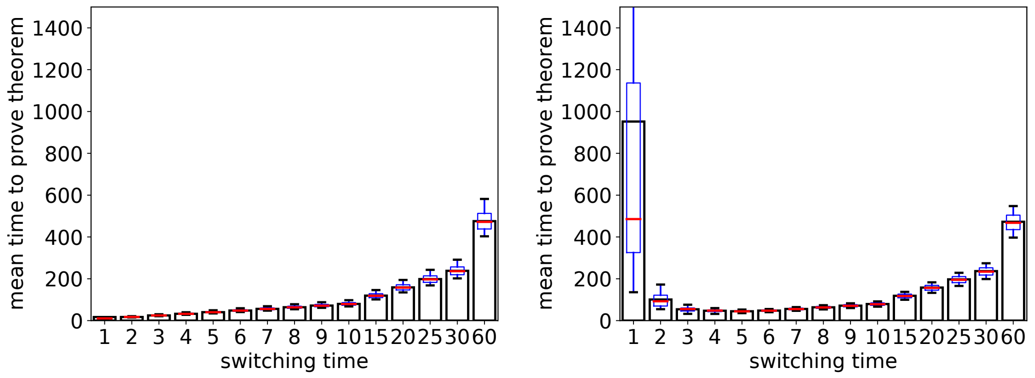


Figure 4.8: Mean time to prove a theorem using the model interrupts optimization for different switching times (in seconds). Executing each model's search for X seconds, and then again each model's search for X seconds, and so on until each model's search has been executed for 600 seconds (left graph), achieves the minimal mean time to prove a theorem of 17.2 seconds when $X = 1$ second. Interrupting each model's search once, first executing each model for X seconds, and then each model for $600 - X$ seconds, (right graph), achieves the minimal mean time to prove a theorem of 44.7 seconds when $X = 5$ seconds. The box-and-whiskers indicate the maximum, 75%-, 50%-, and 25%-tiles, and minimum values over 20 different model orderings.

Second, we allow each model to attempt to synthesize a proof script for X seconds, and then give each model the remainder of its $600 - X$ seconds, thus switching only once per model. The right graph in Figure 4.8 shows the mean time to prove a theorem for this interrupt scheme. For $X = 5$ seconds, this interrupt scheme achieves the minimal mean time to prove a theorem of 44.7 seconds, a speed up of 93%, or 15×, compared to not using interrupts.

RA4: Model interrupts is incredibly effective, cutting down the mean time to prove a theorem by up to 97%.

4.4.6 Threats to validity

The CoqGym benchmark we evaluate our work on has been used by prior evaluations of proof-script synthesis [223, 62] and uses theorems from 122 open-source projects, improving the likelihood that our results generalize. Our analysis focuses on the Coq interactive proof assistant and may not extend to other assistants, such as HOL4 [188] and HOL Light [84]. Transformers have outperformed Bidirectional LSTM in some natural language tasks [54], and may be able to improve Diva's performance beyond what we find here.

4.5 Contributions

We have identified a method for using diversity to significantly improve the proving power of proof-script-synthesis tools. We create Diva, implementing our diversity-based approach, which proves 68% more theorems than TacTok and 77% more than ASTactic, two state-of-the-art proof-script-synthesis tools. Diva automatically proves 364 theorems no existing tool has proved. Together with CoqHammer, Diva proves more than a third of all the theorems in our benchmark of 122 open-source projects, the largest fraction to date. Our model interrupts optimization improves Diva's running time by 40×. Along the way we identify a killer app for ensemble learning, by using the theorem prover as an oracle for optimally aggregating learned model results. Our findings strongly suggest that using diversity for improving automated formal verification is fruitful and warrants further research.

CHAPTER 5

PASSPORT: IMPROVING AUTOMATED FORMAL VERIFICATION USING IDENTIFIERS

The work in this chapter was done in collaboration with Alex Sanchez-Stern, Timothy Zhao, Zhanna Kaufman, Yuriy Brun, and Talia Ringer. The following is adapted directly from work that has been accepted for publication [180].

5.1 Introduction

In recent years, techniques that use machine learning to synthesize proof scripts have shown promise in alleviating some of the effort of verification [179, 62, 59, 223, 154]. These *proof-synthesis* tools learn from corpora of existing proof scripts and theorems to automate the construction of proof scripts for new theorems. In particular, these tools build predictive models of proof scripts, and then use search to explore the proof-script space. This process uses the proof assistant to guide the search and evaluate ultimate success.

In this work, we explore ways of improving these predictive models by better exploiting the richness of the proof data that they learn from. We focus in particular on modeling *identifiers*: the names that uniquely identify theorems, datatypes, functions, type constructors, and local variables. Previous machine-learning-guided proof-synthesis tools have either ignored the names of individual identifiers completely and only encoded basic categorical information about them, or given common identifiers unique indices and marked all others as unknown, without category information. In this work, we develop the Passport approach, which enhances the models used by existing proof-synthesis tools with three new encoding mechanisms for identifiers: category vocabulary indexing, subword sequence modeling, and path elaboration. We implement our approach for tools that synthesize proofs for the Coq proof assistant [199] and show that all three of these encodings improve performance of the end-to-end tool.

The term “Passport approach” refers to our approach of enhancing the model of an existing proof-synthesis tool with identifier information. Most of our evaluation focuses on the application of Passport to a single existing tool, Tok [62]; where unambiguous, we refer to the resulting tool as Passport. Where necessary for clarity, we make explicit the distinction between the approach and the tool resulting from enhancing existing the model of Tok with our approach.

Identifiers in Passport The Passport approach encodes identifiers with three different encoding mechanisms (described in detail in Sections 5.2 and 5.3):

1. **Category Vocabulary Indexing**: we encode each identifier with the category it comes from (global definition, local variable, or type constructor); and for the most common identifiers in each category, we encode indices corresponding to their names. That is, each common identifier is given a unique tag, associating it with all other uses of that exact identifier.

2. **Subword Sequence Modeling**: For all identifiers, we use a subword sequence model to draw bridges between related names. That is, identifiers are broken down into common word-pieces, and processed with a sequence model.

3. **Path Elaboration**: For type constructors and global definitions, we encode their *fully-qualified paths* — the names of directories, files, and modules within which they are contained.

While we focus on Coq in this work, similar techniques should apply for other proof assistants, including Lean [117], Isabelle/HOL [94], and Agda [3].

Results We evaluate the Passport approach using the CoqGym benchmark [223] of 124 open-source Coq projects. We compare to three existing search-based proof-synthesis tools, ASTactic [223], Tac, and Tok [62]. We find that all three of our encoding mechanisms improve tool performance, in terms of being able to prove more theorems fully automatically. For example, adding path elaboration leads to proving 12.6% more theorems. We also measure the impact of adding identifier information to each of the categories of identifiers individually, and find that the Passport approach is useful for each.

Together with the three prior tools, tools enhanced with the Passport approach are able to fully automatically prove 1,820 of the 10,782 theorems in our benchmark test set, whereas without the enhancements, these prior tools combined can prove 1,259 theorems. That is an increase of 45% theorems proven over this prior work.

Contributions The main contributions of our work are:

1. The Passport approach (Section 5.3) consisting of a set of techniques for encoding identifiers in a proof assistant context.

2. The Passport implementation of that approach as a standalone tool within an existing proof-synthesis framework. Passport is open-source: `https://github.com/LASER-UMASS/Passport`

3. An evaluation (Section 5.4) showing that (1) the Passport approach improves proof synthesis when applied to three prior tools, (2) each mechanism for encoding identifiers helps model proof scripts more precisely and improves performance of proof synthesis, and (3) encoding each identifier category alone is still an improvement over not encoding any.

4. A forward-looking discussion (Section 5.5) of the challenges that we faced when building Passport (relative to building symbolic proof automation), along with

```
Definition posnat^1 := {n^2 : nat | n > 0}.

Inductive posnatEq^1 : posnat -> posnat -> Prop :=
  | posnatEq_intro^3 : ...

Definition posnatMult^1(p1^2 p2^2 : posnat) : posnat := ...
```

Figure 5.1: Definitions related to the posnat type, a type of pairs of natural numbers and proofs that they are greater than zero. These definitions are found in the Foundational Cryptography Framework (retrieved as part of the Verified Software Toolchain).

potential solutions to those challenges. Our evaluation includes an experiment measuring the impact of nondeterministic training variance (Section 5.4.6).

5.2 Overview of the Passport Approach

The proof state is made up of many Gallina terms; modeling these terms well is key to producing accurate models. However, previous models have left out much of the essential information about identifiers in terms, when they have encoded identifiers at all. Encoding identifiers well is essential because proof corpora in Coq are rich with identifier information. One reason that identifiers are particularly important in Coq is that Coq has no primitive datatypes; *every* referenced type is an identifier. These names can carry a lot of meaning — and that meaning can be reflected in the names of theorems that refer to them. This work describes and evaluates improvements to identifier encodings in the tactic prediction model.

Categories of Identifiers To begin to harness the latent information in identifiers, the Passport approach adds three categories of identifiers to the term model. To

[-1] https://github.com/adampetcher/fcf

[0] https://vst.cs.princeton.edu/

understand these identifier categories, consider the definitions in Figure 5.1, from a verified cryptography library.

1. The identifier `posnat` is a *global definition* (highlighted in `red`[1]), it can be used by datatypes, functions, theorems, or proof scripts, to reference the globally defined `posnat` datatype.

2. The identifier `n` is a *local variable* (highlighted in `orange`[2]), as it can be referenced within the local context of this term, but not outside of it.

3. The identifier `posnatEq_intro` is a *type constructor* (highlighted in `yellow`[3]) as it can be referenced in datatypes, functions, theorems, and proof scripts to construct a new `posnatEq` object.

Appendix A further details these categories of identifiers (global definitions, local variables, and constructor names) and provides intuition through examples for why each category may be useful to encode in a tactic prediction model. Appendix A.3 details the implementation effort required for enriching a model with these three categories of identifiers.

Encodings Figure 5.2 shows a proof over these definitions, `posnatMult_comm`. This proof says that multiplication of posnats is commutative, meaning you can switch the order of the arguments and the result will always be the same. Making progress in this proof state requires understanding several things about the identifiers involved.

1. The `exist` type constructor is a common constructor for sigma (existential) types, and there are specialized tactics (like `exists` and `eexists`) for reasoning with those objects.

2. The goal type, `posnatEq` is related to `posnats` and equality.

3. The `Nat.mul` function is defined in the Coq's standard library, whereas `mult_gt_0` is a theorem about it defined in the current project.

```
Lemma posnatMult_comm^1 : forall
    p1^2
    p2^2,
  (posnatEq (posnatMult p1 p2)
            (posnatMult p2 p1)).
Proof.
  intuition.
  unfold posnatMult.
  destruct p1; destruct p2.
```

(a) A partial proof of posnatMult_comm.

```
x : nat
g : x > 0
x0 : nat
g0 : x0 > 0
============================
posnatEq^1 (exist^3 (fun n^2 : nat =>
     n > 0)
                 (Nat.mul^1 x^2 x0^2)
                 (mult_gt_0^1 g^2
                    g0^2))
          (exist (fun n : nat => n
              > 0)
                 (Nat.mul x0 x)
                 (mult_gt_0 g0 g))
```

(b) The proof state at this point in the proof.

Figure 5.2: A proof using the definitions in Figure 5.1, from the same file.

Understanding these things requires three different approaches: attaching special signifiers to common identifiers, processing the individual pieces of identifiers to understand where they connect to different concepts, and remembering where the definitions being referenced are defined.

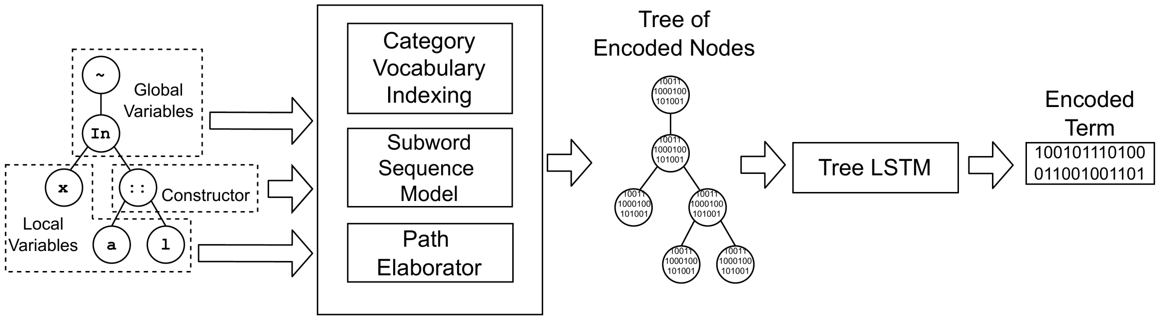


Figure 5.3: The architecture of Passport's identifier processing.

The crux of this work is the enrichment of a proof-synthesis model for Coq with rich information about identifiers. Figure 5.3 shows an overview of how the Passport approach encodes identifiers. To fully take advantage of the richness of these identifiers, our design employs three key encoding mechanisms:

1. *Category Vocabulary Indexing* (Section 5.3.1), which separately considers different kinds of common identifiers in a proof development,

2. *Subword Sequence modeling* (Section 5.3.2), which draws bridges between all identifiers, and

3. *Path Elaboration* (Section 5.3.3), which encodes the location where the object referred to by each identifier is defined.

Category vocabulary indexing allows us to assign unique labels to common identifiers in the code. In this case, that means giving a unique label to the `exist` type constructor, so that we can use knowledge from previous proofs which used that precise constructor. Subword sequence modeling allows us to break identifiers up into common pieces, and process those pieces with a sequence model. In this case, that means breaking the `posnatEq` identifier into the chunks `posnat` and `Eq`, so that we can use knowledge from previous proofs that had identifiers with similar pieces. Finally, path elaboration allows us to consider the directories, files, and modules in which the object referenced by the identifier is defined. Here, that means understanding that the multiply identifier refers to a function defined within `Coq.Init.Nat`, but the `mult_gt_0` refers to a lemma defined in the current file.

Armed with the knowledge from these three encoding mechanisms, our model has everything it needs to suggest tactics that the tool can use to complete the proof of `posnatMult_comm`.

5.3 Passport Encodings

Identifiers are proxies for semantic information not by accident, but *by design*. By taking advantage of the information in identifiers, term models can learn from the design principles the proof engineer has already followed to make proof developments easier to read, understand, and build on. To extract this information from identifiers, the Passport approach uses three encoding mechanisms: **category vocabulary indexing** (Section 5.3.1), **subword sequence modeling** (Section 5.3.2), and **path elaboration** (Section 5.3.3).

5.3.1 Category Vocabulary Indexing

In each identifier category (global definitions, local variables, and type constructors), there are many common identifiers used across proof developments. These identifiers are so common that we can learn a significant amount about how to understand them from their previous uses. For instance, in the example from Figure 5.2, the `exist` type constructor is part of the standard library, and many proofs in our training data reason with it. Even when an identifier is not very common, we can still understand a lot about it by knowing what category it is in.

To take advantage of these properties of identifiers, we developed **category vocabulary indexing**. This encoding mechanism tags every identifier with the category it comes from and, if the identifier is commonly used enough, a unique tag for that particular identifier. By giving common identifiers a unique tag, we can generalize across their many appearances, and predict tactics that worked well with them in the past. And by marking identifiers with their category, either global definition, local variable, or type constructor, we can disambiguate identifiers with the same name from different categories, and learn useful information about even uncommon identifiers.

The models in some previous tools for machine-learning-guided proof-synthesis, such as Proverbot9001 [179] and Tactician [22], use vocabulary indexing for common identifiers, but make no category distinctions. This is a reasonable approach, because in Coq, the names of global definitions, local variables, and type constructors share a common namespace. However, in the Passport approach, we decided to distinguish between identifiers of different categories, in part because manual analysis of the training data revealed different naming conventions for different categories. For example, single-letter identifiers seemed to almost exclusively represent local variables, with uppercase for types (like `A` in Figure A.1), and lowercase for terms (like `x` in Figure 5.2); longer uppercase identifiers generally refer either to sort names (like `Set` or `Prop`) or type constructors (like `Some` or `None`). This means that when human provers see an identifier, even if they have not seen it before, they often have a sense of what category it belongs to.

The models in other previous tools for machine-learning-guided proof-synthesis, such as ASTactic and TacTok, make category distinctions, but do not index vocabulary. We learned early on that the possibility of performance regression due to uninformative local variables like `x` had concerned the ASTactic authors, and contributed to their decision not to encode identifiers.[1] However, upon closer inspection of the data, we determined that even when a particular name does not always refer to the same definition, common names can carry information of their own. For instance, variables named `hd` and `tl` consistently refer to the head and tail of a list. These names, too, can benefit from a unique tag which generalizes across their usages. Our manual inspection determined that this can often hold even for single-character variable names.

Implementation To decide which identifiers are common enough to be indexed, we use our training data set to create a fixed identifier vocabulary. That is, we count the occurrences of each identifier, and include in our vocabulary those whose count is above an experimentally chosen, fixed threshold (see Section 5.4.7 for an evaluation of different thresholds). Using separate vocabularies for each category of identifier allows us to use different thresholds across different categories; since type constructors are

[1]`https://github.com/princeton-vl/CoqGym/discussions/60`

less common overall than local variables, they might require having a lower threshold for being included in the vocabulary.

5.3.2 Subword Sequence Modeling

Identifier information can be useful not just for learning about individual datatypes, theorems, and functions, but also for drawing bridges between them. Developers often organize development using parts of names to group theorems and functions which refer to common definitions. It turns out these naming conventions can be useful to a model, too.

Many variable names are not simply single unique words, but are made up of multiple parts. These parts could be multiple english words in camel case, such as the case in something like `firstItemInList` broken into "first", "item", "in", and "list". Or they could be components of a single word that carry individual meaning, like `prelocalizations` broken into "pre" "local" "ization" "s". By breaking these identifiers into pieces, a model built using the Passport approach can learn the meaning of shared pieces and generalize across identifiers.

In the example from Section 5.2, Passport breaks `posnatMult` into `[pos, nat, Mult]`; with a different subword vocabulary, from a different set of variable occurrences in the training data, it might produce `[posnat, Mult]`. These tokens are processed with a sequence model, so that the identifier's ultimate feature vector reflects the fact that the identifier relates to the "posnat" type, and that it primarily relates to the multiplication operation.

To get a sense for this, let us consider another example. The Coq standard library includes operations about the real numbers `R`, like addition:

```
Rplus¹ : R → R → R.
```

The library contains proofs of theorems about `Rplus`, like this proof (highlighting just one `Rplus` for presentation):

```
Lemma Rplus_eq_compat_l : ∀ (r r1 r2 : R),
    r1 = r2 → Rplus¹ r r1 = Rplus r r2.
Proof.
  intros r r1 r2.
  apply f_equal.
Qed.
```

which proves the theorem that right addition preserves equality.

Suppose we wish to prove the analogous theorem about the natural numbers `nat`, using the addition function `plus` defined over `nat`. We can do this the same way:

```
Lemma plus_eq_compat_l : ∀ (n n1 n2 : nat),
    n1 = n2 → plus¹ n n1 = plus n n2.
Proof.
  intros n n1 n2.
  apply f_equal.
Qed.
```

simply renaming the local variables for style (though the original proof with `r`, `r1`, and `r2` also works with no changes).

The fact that `Rplus` and `plus` are related is explicit in the identifier names: `Rplus` behaves like `plus` over `R`. A model that can draw connections between `plus` and `Rplus` can in some cases reuse proofs about one to derive analogous proofs about the other.

The key here is subword sequence modeling which excels at drawing connections between related words [65, 184]. Subword sequence modeling allows us to break the identifier `Rplus` into the chunks `R` and `plus`, and index them separately, connecting them to the identifier `plus`. By drawing these connections, we expect that a model can suggest `intros` and `f_equal` in the body of `plus_eq_compat_l`, by connecting the hypothesis `plus n n1 = plus n n2` to the hypothesis `Rplus n n1 = Rplus n n2`. With subword sequence modeling, the model can learn all of this with no need for semantic information about what each of the reals and naturals represent, or how their addition functions are related.

In the Passport approach, identifiers are broken into subwords using a byte-pair encoding algorithm (BPE) [65, 184], an algorithm that has seen success in code completion models for program synthesis [103, 196]. The algorithm uses the training corpus to make a list of common subwords by starting with a vocabulary of single characters, and iteratively merging common pairs. Then, each identifier is tokenized by greedily consuming the longest matching vocabulary element.

The Passport approach incorporates these tokens as embeddings in a syntax model. Program syntax can generally be modeled in two ways. The simplest way is to model it as an unstructured sequence of words (or more generally, tokens). The alternative is to parse the syntax into a tree, and use a tree based model to process it. One of the advantages of the former is that you can tokenize strings in a number of different ways, including with multiple tokens per identifier (sub-word tokenization). However, our implementation of Passport builds on a parsed-tree-based model, so there is no existing string tokenizer that could be used for subword tokenization. Instead, we embed a sequence model *within the leaves* of the tree-based syntax model. This means that our subword sequence model only learns how to combine parts of an identifier into a fixed embedding for the identifier, and does not need to learn about other parts of program syntax.

With our category vocabulary indexing, we used separate vocabularies for identifiers of different categories. However, proof developments sometimes demonstrate connections between identifiers from different categories. These connections are lost in using separate vocabularies, so subword encoding is used to maintain these connections. The Passport approach uses a single subword vocabulary, derived from the global variable corpus, to encode identifiers from all categories.

Implementation There are several subtleties to the implementation of our subword tokenization algorithm, and the byte-pair encoding which generates its vocabulary. Sometimes there were several possible ways to implement the approach; in general, we made our choices based on the performance of the resulting tool on our benchmarks.

As indicated by the name, byte-pair tokenization often starts with a vocabulary of bytes, not characters, to allow a reasonable base vocabulary size when working with unicode. However this has the downside of sometimes indicating that two identifiers are similar because they share bytes within a unicode character, even if no characters are in common. In our implementation, we use characters as our base vocabulary. To keep our base vocabulary of a reasonable size, we only include those characters which are present in the training corpus. Since Coq programmers generally only use a small subset of possible unicode characters, this works well. However, there are in rare cases unicode characters present in the test data which are not present in the training data. To address this, our subword tokenizer drops characters which are not present at all in the vocabulary; this behavior can be changed with a flag to instead produce a special `<unknown>` element.

Many different neural architectures have been used to process sequences of tokens. For language modeling, the most effective models are often those with attention and forgetfulness mechanisms, to capture the long-range dependencies present in text. However, the identifiers we work with are generally short, often only a few subwords long, so we instead use the simplest sequence model, a Recurrent Neural Network, without any attention mechanism.

As with any sequence-based model, there is a question of how to cap the size of sequences so that their lengths can be normalized. With Passport, we found empirically that capping at four tokens per identifier during training, but eight tokens per identifier when synthesizing proofs, is most effective on our evaluation suite. Four subwords is enough to encode the entire name of 98.74% of identifiers in our training data, and eight subwords is enough to encode the entire name 99.97% of the time.

We trained the subword encoder end-to-end alongside the rest of the term encoder and tactic decoder, so that the encoder is trained to retain information about subwords particularly relevant to the task of tactic prediction.

5.3.3 Path Elaboration

The final encoding mechanism in the Passport approach is path elaboration: the encoding of fully-qualified paths of different identifiers. By paying attention to the fully-qualified paths of different identifiers, the tools using the Passport approach can take advantage of any grouping of identifiers into common modules and files already used by Coq developers to organize development. Tools using Passport approach can also capitalize on proof development styles that dispatch proofs for entire classes of related theorems using powerful tactics — a proof development style recommended by, for example, the popular Coq textbook Certified Programming with Dependent Types [41].

To gain some intuition for what this means in action, consider this proof of a theorem from the Coq standard library:

```
Theorem not_in_cons A (x a : A) (l : list A):
  ~ In x (a::l) ↔ x<>a ∧ ~ In x l.
Proof.
  simpl. intuition.
Qed.
```

The proof of `not_in_cons` goes through by just two tactics: `simpl` and `intuition`. The `simpl` tactic simplifies the initial goal (no assumptions, with the theorem type as the sole proof obligation) to make it easier to reason about, producing this proof state:

```
A : Type
x, a : A
l : list A
______________________________________(1/1)
~ (a = x ∨ In x l) ↔ x <> a ∧ ~ In x l
```

In this case, the `simpl` tactic has unfolded the `In x (a::l)` on the left side of the identifier into `(a = x ∨ In x l)`.

But the resulting goal is still a bit complex because it chains together a number of logical connectives: if and only if (↔), negation (~), inequality (<>), conjunction (∧), and disjunction (∨). So the `intuition` tactic breaks down logical connectives into simpler subgoals, and dispatches each subgoal automatically.

```
not_in_cons^1
: ∀ (A^2 : Type) (x a^2 : A) (l^2 : list A),
    Coq.Init.Logic.iff^1
      (Coq.Init.Logic.not^1
        (In^1 A x (cons^3 A a l)))
      (Coq.Init.Logic.and^1
        (Coq.Init.Logic.not
          (Coq.Init.Logic.eq^1 A x a))
        (Coq.Init.Logic.not (In A x l))).
```

Figure 5.4: The theorem statement `not_in_cons`, elaborated with paths. Highlighted using the same conventions as in Figure 5.1, with other paths omitted for brevity.

Taking a step back, it is natural to wonder how the proof engineer could have known to use the `intuition` tactic to dispatch the remaining goals. Intuitively, it made sense to use `intuition` here because the goal consisted of simple statements linked by logical connectives, which `intuition` excels at. It turns out that the fact that these operators are logical connectives is explicit in the paths of the identifiers in the goal — they all reside in the `Coq.Init.Logic` module — so we can pass it on to our models by encoding paths.

We can see this by expanding the paths of the identifiers in the theorem statement of `not_in_cons` (Figure 5.4). All of the operators in `not_in_cons` are syntactic sugar for identifiers, which themselves refer to types defined inductively in Coq. For example, conjunction ($\wedge$) refers to the inductive type `and` in the path `Coq.Init.Logic`.

Internally, Coq stores the elaborated theorem with all of these identifiers (like `and`) and their fully-qualified paths (like `Coq.Init.Logic`) explicit.

Inspecting the elaborated version of `not_in_cons` shows that the fact that these are logical connectives requires no semantic understanding to deduce — it is explicit in the grouping of identifiers in the `Logic` module.

We determined that a simple way to pass this intuition on to our models was to encode each of the file and module names inside of fully-qualified paths, taking advantage of the organization of large proof developments to infer tactics used to dispatch related goals.

Implementation To implement this, we created a dedicated vocabulary and corresponding `<unknown>` token for file and module names inside of fully-qualified paths, much like we did for each category of identifier. We then used this vocabulary for encoding paths.

As with identifiers, Coq includes fully-qualified paths inside of the ASTs by default, but TacTok and ASTactic had erased those paths from the AST. For example, in Figure A.3, the fully-qualified path `Coq.Init.Datatypes` of the `option` inductive type shows up in the AST as a `directory_path` node, with data `[Datatypes; Init; Coq]`.

Elaborating paths was thus similar to adding each of the categories of identifiers: First, we modified the post-processing code to avoid erasing paths. Then, we built a separate vocabulary for common files or modules that paths consisted of, like `Datatypes`, `Init`, and `Coq` in Figure A.3. We then encoded each file or module along the path separately, mapping to a dedicated `<unknown>` token for files or modules in paths that occurred less frequently than the chosen threshold.

5.4 Passport Evaluation

We evaluated Passport's ability to successfully prove theorems using the CoqGym benchmark [223], following the evaluation methodology used by several recent papers [223, 62, 59].

In summary, our results show:

- **The Passport approach improves proving power.** By comparing to previous tools — ASTactic and the two base tools, Tac and Tok, that make up TacTok — we measured additional proving power provided by the Passport approach's encoding of identifiers. The combined proving power of the tools enhanced by the Passport approach exceeds that of the original tools by 38%, and combining both the enhanced and un-enhanced tools outperforms the combined un-enhanced tools by 45% (Section 5.4.2).

- **Identifiers improve performance.** All three categories of identifiers improve performance, in aggregate proving 64% more theorems than the individual un-enhanced tool (Section 5.4.3).
- **All three encoding mechanisms improve performance.** All three categories of identifiers in the Passport approach improve performance in Passport with each of the three encoding mechanisms (Sections 5.4.4 and 5.4.5).
- **Our results are meaningful beyond variance introduced by nondeterminism.** Proof synthesis success rate varies by 0.4% for individual tools, and combining many varying runs can improve results by 22% (Section 5.4.6).
- **Hyperparameter choices impact performance.** We choose our hyperparameters experimentally based on these results (Section 5.4.7).

All our experiments are affected by nondeterminism, and while the bulk of our experiments only use a single trial, Section 5.4.6 explores the effect on nondeterminism on the variance of our results and argues that that effect is small.

5.4.1 Experimental Setup

Benchmark The CoqGym benchmark includes 124 open-source Coq projects, split into three sets. For our evaluation, we trained on 97 projects (containing a total of 57,719 theorems) and synthesized proofs for 26 projects (containing a total of 10,782 theorems). We exclude one project, coq-library-undecidability, from our evaluation because TacTok's evaluation [62] was unable to reproduce prior results for ASTactic's performance [223] on that project due to internal Coq errors when processing the proof scripts.

Projects in the CoqGym benchmark are a mixture of mathematical formalizations, proven correct programs, and Coq automation libraries. They include several compilers of varying sizes (such as CompCert [122]), distributed systems (such as Verdi [217]), formalizations of set theory, and more. Some of the projects in CoqGym (such as the automation libraries) do not contain any proofs, but we included them for completeness.

Machines We ran this work's experiments using two clusters: a GPU cluster for training and a CPU cluster for synthesizing proofs.

Each node in the GPU cluster has between two and eight NVIDIA GPU cards. There are four nodes with two NVIDIA Tesla V100 GPUs, and thirty-three nodes with eight NVIDIA RTX 2080ti GPUs. The nodes in the GPU cluster all run on a shared ZFS file system, run CentOS Linux, and use Slurm for job scheduling and resource management. We found that training time varied between 12 and 14 hours per epoch, and did not differ significantly between the Passport implementation and the baseline model.

Each node in the CPU cluster has between 24 and 36 cores, with 4 hyperthreads per core. There are:

- 1 head node with 24 cores of Xeon E5-2680 v4 @ 2.40GHz, 128GB RAM and 200GB local SSD disk.
- 50 compute nodes with 28 cores of Xeon E5-2680 v4 @ 2.40GHz, 128GB RAM and 200GB local SSD disk.
- 50 compute nodes with 28 cores of Xeon Gold 6240 CPU @ 2.60GHz, 192GB RAM and 240GB local SSD disk.
- 5 compute nodes with 56 cores of Xeon E5-2680 v4 @ 2.40GHz, 264GB RAM and 30TB local disk.

The nodes in the CPU cluster also all run on a shared ZFS file system, run CentOS Linux, and use Slurm for job scheduling and resource management. The average inference time for a random sample of generated proofs was 0.4 seconds per tactic for the Passport implementation, compared to 0.3 seconds for the baseline model.

Experimental Parameters Passport attempts to synthesize each proof for a preset amount of time, timing out if it fails to to reach `Qed` in that time. Our evaluation used 10 minutes for this timeout, following the choice made by ASTactic [223] and TacTok [62]. Following a design decision made by ASTactic, we limited our search to a total of 300 attempted tactics, and restrict solutions to be no longer than 50 tactics long. Our experiments use 200 as the default category vocabulary threshold (recall Section 5.3.1) and 4,096 as the default byte-pair merge threshold (recall Section 5.3.2). We use 128 as the default vector dimension for term, grammar, and terminal/non-terminal symbol embeddings, as well as the dimension of the LSTM controller. For all other parameters, we follow those used by ASTactic [223] and TacTok [62].

Implementation Overall, the Passport approach implementation is 1.5K lines of code and took four developers about a year to build. While the conceptual and design aspects of the Passport approach can extend to all prediction-model-driven, search-based, proof-synthesis tools, the current implementation is straightforwardly applicable to all such tools built within the CoqGym environment [223].

This implementation adds three embeddings for category indexes and one for paths, with 428, 136, 27, and 262 items for global definitions, locals, constructors, and paths, respectively. This results in a corresponding increase to the first layer of Tok's term encoder. The new subword embedding contains 4,164 items and is encoded with an RNN using a hidden size of 32. When implementing these new model components, we optimized for simplicity over model size, so we believe that the model size could be decreased further without significantly impacting accuracy.

The original ASTactic, Tok, and Tac models used a 256-float symbol embedding size. However, we observed no significant difference between those models using a 256-float symbol embedding, and using a 128-float symbol embedding. As a result, our model uses 128-float symbol embeddings, and, where appropriate, we compared to versions of other models with a 128-float symbol embedding. Overall, these changes to model size had no significant impact on training time, as described above.

5.4.2 The Passport Approach's Effect on Proof-Synthesis Tools

In this section, we show that the addition of our identifier information improves the end-to-end performance of proof search tools. Since Passport is implemented in the ASTactic/TacTok framework, we were able to evaluate our changes against three base tools: An ASTactic-like[2] tool, Tac, and Tok. ASTactic was developed as part of the CoqGym project [223], and uses only proof contexts as input to their prediction model. By contrast, the models in Tac and Tok (developed as part of the TacTok project [62]) additionally model the proof script up to the current point, with Tac's model encoding the tactics in the proof script, and the Tok's model encoding all the tokens except punctuation in the proof script.

Figure 5.5 shows the results of adding identifier information to all three of these tools. Adding identifiers to each of the three tools significantly improves their ability to prove theorems. Adding identifier information improves our ASTactic-like tool by 29% (304 additional theorems proved), Tac by 14% (136 additional theorems proved), and Tok by 33% (318 additional theorems proved).

Following TacTok's [62] and Diva's [59] evaluations, we also explore how the differences in theorems proven by multiple tools lead to more theorems proven overall, and how adding identifier information increases that improvement. When we compute

[2]We were not able to replicate the original results of ASTactic [223], so for our evaluations we trained a model with the same embedding vector dimensions as our own models. For this reason we are using the term ASTactic-like when we describe our results.

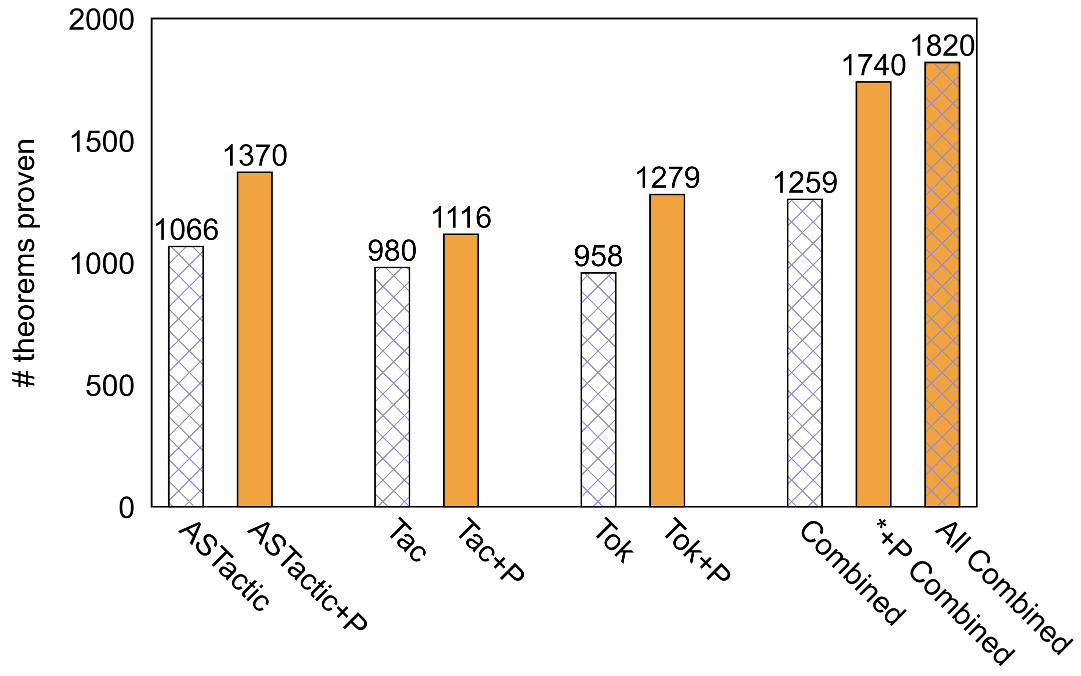


Figure 5.5: The effect of adding all of the three encodings for three identifier types to several proof-synthesis tools. The purple crosshatch bars represent baseline tools based on ASTactic, Tok, and Tac. The orange bars represent our new contributions. The rightmost crosshatch bar, labeled "Combined", is the number of theorems successfully proven by *at least one* of the baseline tools. The orange bar next to that, labeled "*+P Combined", is the number of theorems successfully proven by *at least one* of the tools enhanced by the Passport approach. Finally, the orange *and* crosshatched bar on the far right is the number of theorems proven by at least one of all the presented tools.

the union of the theorems proven by all our tools enhanced by the Passport approach, and compare that set to the union of the theorems proven by the base tools, we find an improvement of 38%. Comparing the union of theorems proven by all the tools to the union of theorems proven by the three base tools, we find an improvement of 45%.

Next, we examine the complexity of the proofs that Passport generated. Using human-written proof-script length as a rough proxy for complexity, we note that Passport successfully synthesized proof scripts for 351 theorems for which the human-written proof scripts were at least 5 tactics long. For 54 of those theorems, the human-written proof scripts were at least 10 tactics long. This observation suggests that Passport is able to synthesize a significant number of nontrivial proofs. For 280 theorems, Passport was able to synthesize proof scripts that were shorter than the human-written ones. In one particular case, the human-written script was 139 tactics long, while Passport's script was only 2 tactics long. The baseline tool produced 239 proofs for which the human-written proof scripts were at least 5 tactics long, so Passport proved 46.9% more theorems with human-written proofs of that length. For theorems with human-written proofs of length 10 or more, the baseline tool produced 37 proofs, so Passport proved 45.9% more such theorems. Finally, the baseline model produced proofs shorter than the human-written proofs for 171 theorems, so Passport did so for 63.7% more theorems.

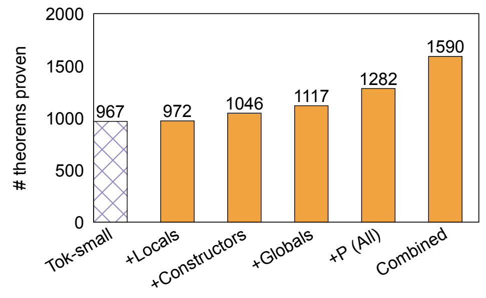
Examining the time it takes Passport to synthesize a proof script, the successfully generated proof scripts took between 0.08 and 86.6 seconds to generate, with the mean of 2.9 seconds.

5.4.3 Identifier Categories

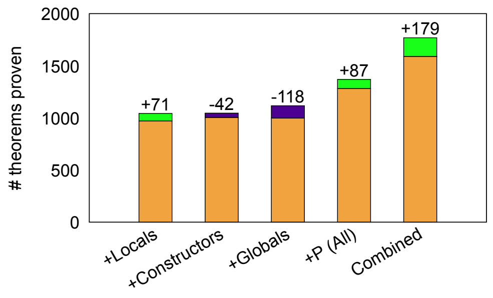
In the Passport approach, we model three categories of identifiers. While the experiment in Section 5.4.2 showed that modeling identifiers from these categories are effective together, we also want to show the utility of the identifier categories individually.

Figure 5.6a shows the individual results of just adding local variables, type constructors, and global definitions. For consistency, this experiment compares to a Tok-like tool with a model with smaller embedding sizes, as Passport uses that model to add identifier information to.

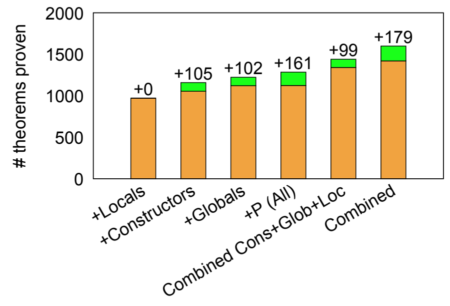
Each of the identifier types added individually increases the number of theorems proven, though the increase from local variables alone is marginal. Adding type constructors alone proves 8% more theorems than the baseline, adding global definitions



(a) The impact of category vocabulary indexing on three identifier categories (without subwords or paths): local variables, type constructors, and global definitions.



(b) The impact of subword encoding on each of the categories of identifiers (with category vocabulary indexing but without paths).



(c) The impact of fully-qualified path encoding of type constructors and global definitions (with category vocabulary indexing but without subwords).

Figure 5.6

alone proves 16% more theorems, and adding local variables alone proves 0.5% more theorems.

However, no identifier category added individually is close to the impact of adding all three. Adding all three identifier types, without subword information, proves 33% more theorems.

Finally, though none of the tools with individual identifier types prove as many theorems as the one with all of them together, some of these individual identifier-enriched tools prove theorems that the all-identifiers-enriched tool does not. The union of the theorems proven by the individual identifier-enriched tools and the all-identifiers-enriched tool contains 64% more theorems than the baseline tool.

These experiments show that each identifier category is useful for producing a more effective proof-synthesis tool, and that the identifier categories help with a diverse set of theorems, so combining the results of adding different subsets of identifiers helps further.

5.4.4 Subwords

Figure 5.6b shows the impact of adding subword encodings to our identifier-enriched tools (Section 5.3.2). Adding the subword encoding does not benefit all types of identifiers individually. In fact, it makes two (type constructors and global definitions) out of the three identifier categories perform worse than when those identifiers are used individually, possibly due to overfitting.

However, when subwords are added to the full tool with all the identifier categories, they improve results by 7%. This improvement is greater than what the cumulative impact of adding subwords to individual identifier-enriched tools, suggesting that subwords particularly help with making connections between multiple identifier types. In fact, even though subword sequence modeling does not help global definitions alone, when global definitions are combined with the other identifier types, removing subword encoding significantly hurts results.

The most likely explanation for these results is that for subwords to be effective, a sufficiently large number of identifiers is necessary to encounter a non-trivial number of repeated subwords, allowed for learning semantics of those subwords. Adding subwords to only a single type of identifier likely does not meet that threshold, but using all identifiers leads to a significant improvement in the tool's proving power.

5.4.5 Paths

Figure 5.6c shows the impact of removing path elaboration (Section 5.3.3) from various identifier types in the Passport model. Since local variables do not have paths, there is no impact of removing path elaboration. Subwords were not included in this experiment, as we wanted to isolate the impact of paths.

Path elaboration benefits both type constructors and global definitions: increasing theorems proven for type constructors alone by 10% and increasing theorems proven for global definitions alone by 9%. The union of the theorems proven using these categories alone and the theorems proven with local variables alone (for which the paths improvement is 0%), is 7% larger than without path elaboration. However, when we add path elaboration to Passport's model with *all three* identifier categories, it increases the number of theorems proven by 12.6%.

These results indicate that the impact of adding path elaboration to a model that implements local variables, type constructors, and global definitions is greater than the combined effect on individual models. Similarly to the subword experiment above, these results suggest that encoding fully-qualified paths helps connect identifiers across categories; learning about how type constructors from a particular module behave helps in dealing with global definitions from that module, and visa versa. However, unlike the subword experiment, paths seem to benefit all identifiers for which they are implemented individually as well as in combination.

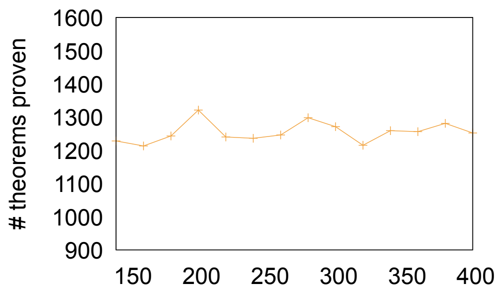
5.4.6 Nondeterministic Model Variance

During the course of our evaluation, we found that models trained in the ASTactic framework had significant variance in their downstream proof-synthesis success rate, even when the model code and training data were identical. While part of this variance could be attributed to different hardware and other hard-to-control factors (see Section 5.5), even when controlling for all those factors, there was still variance. After months of investigation, we found that the cause was nondeterminism at the hardware and framework level, some of it undocumented [169, 67].

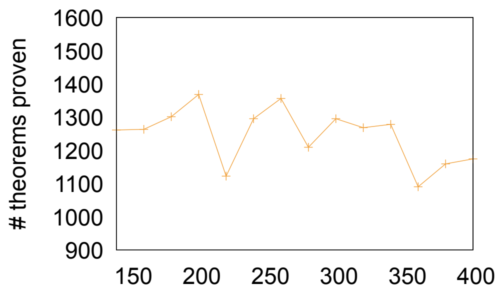
Nondeterminism in model training is not specific to proof search, and has in fact been documented in the ML community at large [159, 186, 165]. However, it is not immediately obvious how these effects would impact proof search, since they are usually measured as inaccuracy in the top prediction of a model, while proof-search tools generally use multiple model predictions, smoothing out some inaccuracy.

To measure the impact of nondeterministic training variance on proof search, we trained our model with identifiers added to Tok's model 20 times. On average, the tool using one of these models proved 11.9% (1,279 theorems), with the maximum proving 12.0% (1,294 theorems) and the minimum proving 11.6% (1,256 theorems). The 0.4% spread (38 theorems) shows that training the same model can lead to small differences in overall success rates. Our result for adding local variables alone (with no other identifiers) and without subword encoding is within this variance range. However, the impact of local variables is better captured with the addition of subwords and together with other identifiers, which yields results significantly outside of this range.

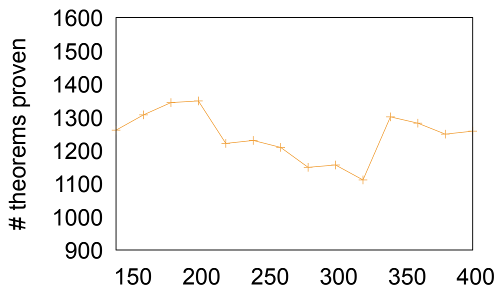
Interestingly, the union of the theorems proven by the tool using these 20 models is 14.5% (1,564 theorems), an improvement of 22% over the average. This demonstrates



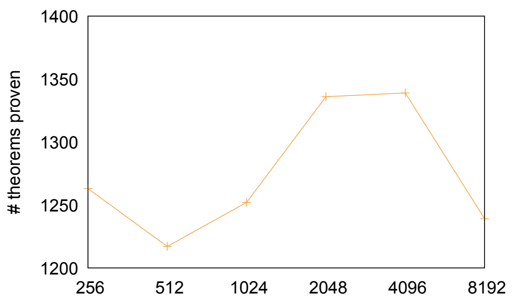
(a) Global definitions



(b) Local variables



(c) Type constructors



(d) BPE merges

Figure 5.7: The impact of different vocabulary thresholds for the various categories of identifiers. A smaller threshold means the vocabulary is larger.

that the scale of the differences in *which* theorems model-based tools can prove as a result of nondeterministic training variance is much larger than the scale of the differences in *how many* they prove. Thus, the variance from training nondeterminism serves as a dimension for model diversity, which can be used to improve proof synthesis, similarly to the approach taken by Diva [59].

5.4.7 Hyperparameters

As discussed in Section 5.3.1, each of the identifier types we add has a vocabulary of the most common identifiers of that type, giving a fixed encoding of those identifiers in addition to the subword encoding. We count the occurrences of the identifiers in the training set to determine which identifiers occur more than a specified threshold, and then only include those identifiers in our vocabulary. For example, if we have a threshold of 100, then all the identifiers that occur at least 100 times in the training set will be included in the vocabulary. That threshold is a hyperparameter that we can vary for each type of identifier, and it determines the size of the vocabulary.

Figure 5.7 shows the performance impact of different values of that hyperparameter for different identifiers. As you can see the performance of various vocabulary sizes for global definitions, local variables, and type constructors are all fairly jagged, though they all peak at around 200 occurrences, which we set as the default in the rest of our experiments.

It is interesting to note that, while the thresholds which produce the best results are the same for the different identifier categories, this results in drastically different vocabulary sizes: 427 global definitions meet the threshold, but only 135 local variables and 26 type constructors do. This justifies our decision to use a fixed occurrence threshold to pick vocabulary rather than using the n most common identifiers from each category.

However, there are signs that our method of picking vocabulary to index could be improved. Sometimes, adding identifiers with fewer occurrences, such as the global definitions with between 180 and 200 occurrences, helps; while adding those with more occurrences, such as the global definitions with between 200 and 220 occurrences, hurts. This suggests that the number of occurrences does not monotonically predict the usefulness of indexing a particular identifier, even though it is the most common approach. Future systems should investigate new metrics to pick vocabulary for indexing. Finally, these experiments indicate that the model — and therefore the proof-search tool — is sensitive to small changes in hyperparameters, similar to how model-based tool performance varies greatly from nondeterminism at the hardware level in model training.

The subword encoding we use also has several hyperparameters which can be varied; principle among these is the number of byte-pair merges, which determines the size of the subword vocabulary. Figure 5.7d shows the effect of different subword vocabulary sizes on success rate. The default byte-pair merge threshold of 4,096 is represented as the the highest point on the graph.

5.5 Discussion

We believe that it is prudent to broaden the discourse around machine learning for proofs to consider not just the tool produced, but also the development processes in building these tools. It is for this reason that we step back and discuss our experiences, centering challenges that we encountered in three areas: the feedback cycle, reproducibility, debugging.

Feedback Cycle The feedback cycle for developing Passport was slow. Every time we changed an encoding, we had to retrain the model, a process that took around two days. Mistakes in the code or in the training parameters would often not manifest until evaluation, at which point we would need to retrain once more. This slow feedback cycle quickly added up, so that even a small change could take weeks.

In traditional supervised learning, training dominates development time, as evaluating a model means running it just once on the test set. However, in the context of proof search, evaluation on a large benchmark set often takes as many or more computational resources as training, though it is usually more parallelizable across machines.

In the machine-learning literature, techniques have been proposed to make training faster [163, 120, 167, 124], which could be directly applied in proof search. And more tooling like data trackers [21], data validation, and static types can help catch bugs sooner, resulting in fewer training runs needed during development. Finally, some work in combining multiple models [59] has shown an ability to speed up proof search, and other search optimizations could also shorten that part of the feedback cycle.

Reproducibility As discussed and measured in our evaluation (Section 5.4.6), many current learning frameworks and APIs behave nondeterministically, resulting in nondeterministic variance in our end-to-end proof results. Much of the nondeterminism we encountered is difficult but possible to control, when it stems from hardware differences, random seeds, or OS-level file ordering. However, even when controlling for those factors and all documented nondeterminism, we found our model training was still nondeterministic. During the course of our development, we discovered some PyTorch APIs that were documented as deterministic behaved nondeterministically; we reported that bug, and the developers marked it as high-priority. [3]

A recent paper found this variance in performance across identical training runs to be pervasive in an evaluation of six popular neural networks on three datasets [159]. This paper found that very few of the researchers or practitioners surveyed in were aware of possible nondeterminism in these systems. We recommend that future researchers using machine-learning for proof search document the hardware and software used to train, and report some measure of the variance in their models' results.

Debugging The debugging of systems that mix machine learning and symbolic manipulation, such as Passport, inherits the challenges of both. Instead of failing to compile or throwing a runtime error, bugs in Passport often manifested solely as drops in evaluation numbers. It was challenging to identify whether these drops were caused by bugs to begin with, let alone in which part of the system the bug occurred when there was one.

We are unable to find any work on debugging machine learning systems outside of (potentially very useful) folk knowledge encoded in blog posts[4] and other informal

[3]https://github.com/pytorch/pytorch/issues/75240

[4]`http://karpathy.github.io/2019/04/25/recipe/`

sources. Perhaps a more formal exploration of debugging machine learning systems is warranted. Both better practices [163] and techniques for improved stability [128] may improve the debugging experience. We suspect that improvements to the challenges surrounding the feedback cycle and reproducibility will be not just helpful for but in fact *essential to* improving debugging, as many debugging difficulties are consequences of these challenges.

Other Difficulties These were only a few of the difficulties we faced as researchers applying machine learning to proof search. These systems are also known to have poor modularity [182] (modifying one component can significantly affect the performance of others); poor explainability [76, 14, 71, 118] (trained models do not lend themselves to high-level interpretation); and large hardware costs [85] (expensive hardware is required to train these models, limiting who can develop them, and often requiring the use of shared clusters which can slow development).

None of these weaknesses are shared by purely symbolic approaches to proof tasks such as proof repair [172], or first-order theorem proving [49]. However, current work indicates that tools using these machine learning models can sometimes overcome limitations that current existing purely symbolic tools cannot [62], especially when the solution space is large.

5.6 Contributions

Our Passport approach enriches a model used for proof synthesis with three different identifier encoding mechanisms: category vocabulary indexing, subword sequence modeling, and path elaboration. We empirically demonstrate that each encoding mechanism improves proof-synthesis performance on the CoqGym benchmark suite. Furthermore, we measured the impact of adding information for each individual category of identifier: global definitions, local variables, and type constructors. Again, empirically, each category improved performance.

These results are consistent with our intuition that identifiers matter for proofs, that the category of an identifier is useful information, and that drawing connections between identifiers is useful for proof synthesis. Passport automatically proves 12.7% of the theorems in CoqGym, an improvement of 38% over Tok (an example proof-synthesis tool), without changing the core architecture beyond the encoding of identifiers. Combining the new tools developed using the Passport approach with three baseline tools automatically proves 17.2% of the theorems in CoqGym, an improvement of 45% over the baseline tools combined. This intuition and these results will help developers of other tools for program and proof synthesis in other languages beyond Coq, and is a fruitful step toward better tools for engineering robust and reliable formally verified software systems.

CHAPTER 6

BALDUR: WHOLE-PROOF GENERATION AND REPAIR USING LARGE LANGUAGE MODELS

The work in this chapter was done in collaboration with Markus N. Rabe, Talia Ringer, and Yuriy Brun during my time as a student researcher at Google and as a research assistant at UMass. The following is adapted directly from work that is currently under review [63].

6.1 Introduction

There are two prior promising approaches for automating proof synthesis. The first is to use *hammers*, such as Sledgehammer [156] for the Isabelle proof assistant. Hammers iteratively apply known mathematical facts using heuristics. The second is to use search-based *neural theorem provers*, such as DeepHOL [13], GPT-f [161], TacticZero [218], Lisa [100], Evariste [112], Diva [59], TacTok [62], and ASTactic [223]. Given a partial proof and the current *proof state* (which consists of the current goal to prove and the list of known assumptions), these tools use neural networks to predict the next individual *proof step*. They use the *proof assistant* to evaluate the proposed next proof steps, which returns a new set of proof states. Neural theorem provers rely on diverse neural architectures, such as Wavenet [13, 204], graph neural networks [154], short long-term memory models [59], and language models with the transformer architecture [161, 82].

In this work, we propose Baldur, a different, simpler approach to proof synthesis. We show that using large language models (LLMs), fine-tuned on proofs, can produce entire proofs for theorems. LLMs are scaled-up transformer models trained on a large amount of text data, including natural language and code, that have proven to be remarkably effective across a wide variety of applications, including question answering, and text and code generation [28, 43]. Here, we show their remarkable effectiveness for whole proof generation.

The main contributions of our work are:

- We develop Baldur, a novel method that generates whole formal proofs using LLMs, without using hammers or computationally expensive search.
- We define a proof repair task and demonstrate that repairing incorrectly generated proofs with LLMs further improves Baldur's proving power when the LLM is given access to the proof assistant's error messages.
- We demonstrate empirically on a large benchmark that Baldur, when combined with prior techniques, significantly improves the state of the art for theorem proving.

We design Baldur to be able to work with any LLM internally, but we evaluate our implementation using two versions of Minerva [123], one with 8 billion parameters and another with 62 billion parameters. By contrast, existing tools that use (L)LMs for theorem proving, either predict individual proof steps [82, 100, 98], or rely on few-shot prompting and require the existence of natural language proofs as hints [99].

We evaluate Baldur on the PISA dataset [100] of Isabelle/HOL theorems and their proofs used in recent state-of-the-art Isabelle/HOL proof synthesis evaluations [100, 98]. The dataset consists of 183K theorems, of which we use 6,336 for measuring effectiveness. Our evaluation answers the following research questions:

RQ1: How effective are LLMs at generating whole proofs?

LLMs outperform small-model-driven search-based methods. Baldur (without repair) is able to generate whole proofs for 47.9% of the theorems completely automatically, whereas search-based approaches prove 39.0% [98].

RQ2: Can LLMs be used to repair proofs?

LLMs can repair proofs, including their own erroneous proof attempts. Baldur proves an additional 1.5% of the theorems when given access to a previous erroneous proof attempt and the error messages produced by the proof assistant, even when controlling for the computational cost of the additional inference. The error message is crucial for this improvement.

RQ3: Can LLMs benefit from using the context of the theorem?

In-context learning is remarkably effective for LLM-based theorem proving. With context, Baldur proves 47.5% of the theorems, but only 40.7% without context for the same model size.

RQ4: Does the size of the LLM affect proof synthesis effectiveness?

Larger LLMs do perform better, suggesting that our approach will continue to improve with further developments in LLM research.

RQ5: How do LLMs compare to other state-of-the-art proof generation methods?

Baldur complements state-of-the-art approaches by proving theorems they do not. Together with Thor [98], a tool that combines a learned model, search, and a hammer, Baldur can prove 65.7% of the theorems, whereas Thor alone proves 57.0%. These findings suggest that LLM- and search-based methods' ideas complement each other and can work together to further improve the automation of formal verification. An ensemble of 10 different fine-tuned Baldur models proves 58.0%.

By leveraging LLMs, Baldur simplifies the proof synthesis pipeline, greatly reducing the complexity and cost of the fine-grained interaction between the prediction model and the proof assistant that search-based methods require. This reduction enables us to leverage the power of LLMs, which would be prohibitively computationally expensive if synthesis required as many LLM queries as search-based methods. Further, those calls would require re-encoding with each step the additional information the LLM might need, whereas our approach allows us to make a single call and process the context only once, sampling multiple proofs of multiple proof steps, at once.[1] Overall, our study strongly suggest that LLMs are a very promising direction of research for automating formal verification, and identifies several new avenues for future explorations.

6.2 The Baldur Approach

Prior approaches to proof synthesis employ a neural model to predict the next *proof step* given the current *proof state*. The proof step predictions then guide a search strategy, such as best-first search or depth-first search. Throughout the search, the proof assistant needs to check each proof step prediction to determine whether it is valid. This means that existing proof synthesis tools require a tight interaction between the neural network and the proof assistant. As we move to using LLMs, this results in complex systems, as LLMs need to run on specialized hardware (GPUs or TPUs), while proof assistants run on CPUs.

We explore a simpler, yet effective method: fine-tuning LLMs to generate complete proofs. This simplification avoids the fine-grained interaction between neural model and the proof assistant, allowing us to run the jobs of generating proofs and checking completely separately. Besides reducing complexity, this can also improve efficiency, because (1) it enables us to use large batch sizes, which can significantly improve hardware utilization during inference (cf. [162]), and (2) when providing additional

[1]Alternatively path advanced caching strategies in the prediction servers of large language models could address this problem. This is beyond the scope of our work.

context to the model, the context now does not have to be reprocessed for each proof step, but only once per proof.

We fine-tune LLMs on proof data to generate entire proofs and explore the impact of giving the LLMs additional information. Our approach and implementation include the following:

- We fine-tune an LLM to generate an entire proof given only the theorem statement. We call this model the *proof generation model* (Section 6.2.1).
- We provide a model a proof attempt that did not check along with the corresponding *error message* from the proof assistant so that the model may attempt to find a better proof. We call this model the *proof repair model* (Section 6.2.2).
- We provide text from the same *theory file* that the problem was taken from. We add only the lines from the theory file that immediately precede the theorem we want to prove. We call this added information the *theory file context* and we add it to the proof generation model (Section 6.2.3).
- The LLM that we fine-tune at the core of all of this is Minerva [123], which is pre-trained on a mathematics corpus. We describe our Baldur-specific implementation details for how we use this model (Section 6.2.4).

These fine-tuned LLMs and their interaction with the Isabelle proof assistant make up our tool Baldur. This section details the Baldur approach, which includes creating training datasets and leveraging LLMs to generate and repair proofs.

6.2.1 Proof Generation

Existing proof generation methods using neural models generate the proof one step at a time. In contrast, our approach generates the entire proof, as illustrated with a single example in Figure 6.1. We use only the theorem statement as input to our *proof generation model*. We then sample a proof attempt from this model and perform proof checking using Isabelle. If Isabelle accepts the proof attempt without an error, then we have proven the theorem. Otherwise, we can try sampling another

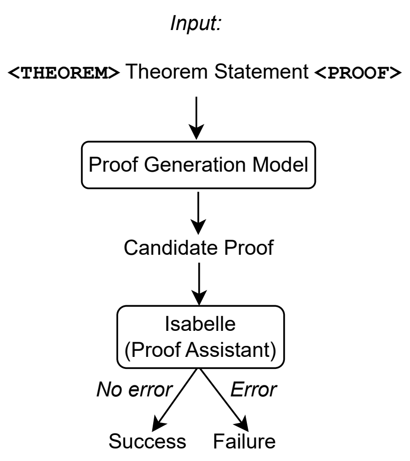


Figure 6.1: An example of using the proof generation model to generate a proof.

proof attempt from the proof generation model. Explicitly, the input and output of our proof generation model is as follows:

- **Input:** theorem statement.
- **Output:** candidate proof.

Example. To illustrate the power of the proof generation approach in our tool Baldur, we first consider, as an example, the theorem `fun_sum_commute`.

```
lemma fun_sum_commute:
  assumes "f 0 = 0" and "⋀x y. f (x + y) = f x + f y"
  shows "f (sum g A) = (Σa∈A. f (g a))"
```

The theorem states that for an additive function f where $f(0) = 0$, and an arbitrary function g, applying f on the sum of the set resulting from applying g on each element in a given set is equal to the sum of applying g followed by f to each element in that set. In this context, it is also assumed that the sum over an infinite set is zero. This theorem is from a project in the Archive of Formal Proofs called Polynomials, specifically in the file `Utils.thy`.

The human-written proof distinguishes between two cases: when the set is finite and when it is not. Induction is used for the finite set case.

```
proof (cases "finite A")
  case True
```

```
    thus ?thesis
    proof (induct A)
      case empty
      thus ?case by (simp add: assms(1))
    next
      case step: (insert a A)
      show ?case by (simp add:
        sum.insert[OF step(1) step(2)]
        assms(2)
        step(3))
    qed
  next
    case False
      thus ?thesis by (simp add: assms(1))
qed
```

If we were to derive a training example from this example, the input would be theorem statement and the target would be this human-written proof.

Our tool Baldur, using the proof generation model, is able to generate the following correct proof for this statement.

```
  by (induct A rule: infinite_finite_induct)
  (simp_all add: assms)
```

Baldur recognizes that induction is necessary and applies a special induction rule called `infinite_finite_induct`, following the same overarching approach as the human-written proof, but much more succinctly. It is interesting to note that Sledgehammer, the hammer for Isabelle, cannot prove this theorem by default, as it requires induction.

Training Data Creation. To train the proof generation model, we construct a new proof generation dataset. Existing datasets for training models in neural theorem provers contain examples of individual proof steps. Each training example includes, at minimum, the proof state (the input) and the next proof step to apply (the target). Given a dataset that contains individual proof steps, we want to create a new dataset so that we can train models to predict entire proofs at once. So we extract the proof steps of each theorem from the dataset and concatenate them to reconstruct the original proofs. We use this data to generate training examples for the proof generation model, where the input consists of the theorem statement and the target consists of the proof.

In particular, this means that we drop the *proof states* from the dataset, which make up most of the text in the dataset. We argue that for Isabelle proofs this is not necessarily a problem, as Isabelle uses a declarative proof language that is designed to be human-readable. This is in contrast to other proof assistants, such as Coq, where the proofs are typically written in a procedural style that is not easy to interpret for humans without using the proof assistant to generate the intermediate proof states.

Inference. We fine-tune an LLM on our data to predict the entire proof given only a theorem statement. To synthesize a proof using the fine-tuned LLM, we provide a potentially unseen theorem statement and sample a fixed number of sequences (typically 16 or 64) from the language model. We tune the sampling temperature from a small set (between 0.0 and 1.4 in increments of 0.2), which is a multiplicative factor on the log probabilities of the distribution of tokens sampled in each step.

Proof checking. After sampling proofs from the model, we check all of them with the proof assistant. This means that we first load the context in which the theorem was originally proven and then replace the original proof of the theorem with the one we sampled from the model. If Isabelle accepts any of the sampled proofs, we report the theorem as proven.

6.2.2 Proof Repair

If a proof is not accepted, Isabelle returns an error message that is intended to help humans with debugging their proof script. Existing proof generation methods, however, have no way to leverage error messages.

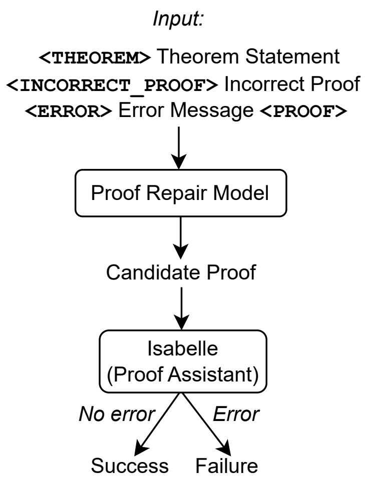


Figure 6.2: An example of using the proof repair model to repair an incorrect proof.

Building off our proof generation approach, we explore the use of error messages to improve neural theorem provers by developing a proof repair approach. Starting with just the problem statement, we apply the proof generation model from Section 6.2.1 to sample a proof attempt. If Isabelle accepts the proof attempt, we can stop. Otherwise, we use the error message returned by the proof checker and the incorrect proof attempt to construct an example to serve as input to the *proof repair model*. As depicted in Figure 6.2, we use the theorem statement, the incorrect proof, and the error message as input to our proof repair model. We then sample the proof attempt from this model, and perform proof checking in the same way as the proof generation approach. Explicitly, the input and output of our proof repair approach pipeline are as follows:

- **Input:** theorem statement, incorrect proof, error message.
- **Output:** candidate proof.

Example Starting from the theorem `fun_sum_commute`, we illustrate an example of the proof repair approach in our tool Baldur. We apply the proof generation model to obtain more proof attempts. The following is a proof attempt generated by Baldur, which fails in the proof checker.

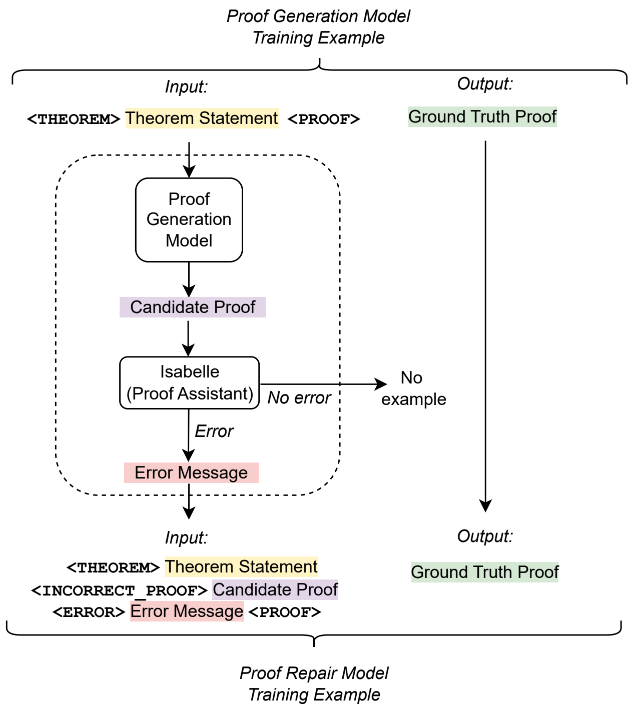


Figure 6.3: Training data creation for the proof repair model.

```
proof (induct A)
case (insert x A)
thus ?case
  by (simp add: assms(2))
qed simp
```

Baldur attempts to apply an induction, but fails to first break down the proof into two cases (finite vs. infinite set). Isabelle returns the following error message:

```
Step error: Unable to figure out induct rule
At command "proof" (line 1)
```

The error message details where the error occurs (line 1) and that the issue is regarding the induct rule. With these strings as input, using the proof repair model, Baldur can attempt to generate a correct proof for this statement. If we want to instead derive a proof repair training example from these strings, we concatenate the theorem statement, the failed proof attempt, and the error message to serve as the input, and we use the correct human-written proof (recall from previous section) as the target.

Training Data Creation. To train the proof repair model, we need to generate a proof repair training set. Figure 6.3 details the training data creation process. Using the proof generation model, we sample one proof with temperature 0 for each problem in the original training set used to train the proof generation model. Using the proof assistant, we record all failed proofs and their error messages. We then proceed to construct the new proof repair training set. For each original training example, we concatenate the theorem statement, the (incorrect) candidate proof generated by the proof generation model, and the corresponding error message to obtain the input sequence of the new training example. For the target sequence, we reuse the ground truth proof from the original training example. We fine-tune the pretrained LLM on the proof repair training set to obtain the proof repair model.

6.2.3 Adding Context

LLMs possess impressive in-context learning abilities (cf. [28, 43]) that allow them to flexibly use information that is provided as part of the input sequence (and, in fact, as part of their own output [151, 213]). In order to explore to what extent in-context learning can help in the theorem proving domain, we extend their inputs with potentially helpful context. Adding to our proof generation approach, we use the theory file contexts (the lines preceding the theorem statement) as input to our *proof generation model with context.* Explicitly, the input and output of our proof generation model with context is as follows:

- **Input:** theory file context and theorem statement.
- **Output:** candidate proof.

Example. Continuing the example, the theory file context directly preceding `fun_sum_commute` is the following theorem statement and its associated proof.

```
lemma additive_implies_homogenous:
  assumes "⋀x y. f (x + y) = f x +
  ((f (y::'a::monoid_add))::'b::cancel_comm_monoid_add)"
  shows "f 0 = 0"
proof -
  have "f (0 + 0) = f 0 + f 0" by (rule assms)
  hence "f 0 = f 0 + f 0" by simp
  thus "f 0 = 0" by simp
qed
```

The proof generation model with context in Baldur can leverage this additional information. Strings that appear in the theorem statement for `fun_sum_commute`, such as `"f 0 = 0"`, appear again in this context, and so the additional information surrounding them could help the model make better predictions.

Training Data Creation. We add the lines of the theory file that precede the theorem statement to serve as additional context. This means that context can include statements, such as the previous theorems, definitions, proofs, and even natural language comments. To make use of the available input length of LLMs, we first add up to 50 preceding statements from the same theory file. During training, we first tokenize all these statements, and then we truncate the left of the sequence to fit the input length.

Premise Selection Many proofs make frequent use of definitions and previously proven statements, also known as *premises*. Some neural theorem provers, such as HOList [13], focus entirely on the problem of selecting the right set of premises, which has been shown to be quite successful in theorem proving.

Premise selection is clearly similar to the addition of context in some aspects, but we want to emphasize some key differences: (1) Adding context is an extremely simple technique that only requires rudimentary text processing, (2) by adding the preceding lines of the theory file, the model can only observe a small fraction of the available premises, (3) most of the added context consists of proofs.

6.2.4 Large Language Model

We use Minerva [123], a large language model pretrained on a mathematics corpus based on the PaLM [43] large language model. Specifically, we use the 8 billion parameter model and the 62 billion parameter model. The Minerva architecture follows the original Transformer architecture [206], but has some noteworthy differences. It is a decoder-only transformer with maximum sequence length of 2,048 tokens. The model uses

- rotary position encodings [193] instead of sinusoidal absolute position embeddings,
- parallel layers [23], which compute the feed forward layer and the attention layer in parallel and add up their results instead of computing them in sequence, and
- multi-query attention, which uses a single key-value pair per token per layer for faster decoding [187].

As this model is not a contribution of this work, we refer the reader to prior work for lower-level details on the Minerva architecture [43].

Baldur-specific implementation details The proof generation task naturally consists of an input, which is the theorem statement (potentially augmented with additional information), and the output (target), which is the proof for the theorem. To work with the decoder-only model, we concatenate the inputs and targets, but the loss is only computed over the target during fine-tuning. The inputs use bidirectional attention while the targets use causal attention as in PrefixLM [166].

As the transformer has a maximum context length of 2048, we pad the sequences with zeros if they are too short, and we need to truncate them if they are too long. Inputs to the model are truncated to the maximum input length by dropping tokens on the left. The rationale for dropping tokens on the left is that the additional context is given before the theorem statement, and can be truncated more safely than the theorem statement itself. Similarly, targets (i.e. the proof to generate) are truncated on the right to the maximum target length.

We used a maximum input length of 1536 and a maximum target length of 512 in all experiments but the repair study, which used 1024 and 1024 instead. We use a drop-out rate of 0.1 for both generation and repair models to address overfitting.

During sampling from the language model we restrict the choice of the next token to the 40 tokens with the highest score, also called top-K sampling [58]. We sample sequences with a maximal length of 256 tokens. The model was trained to generate up to 512 tokens, but since most successful proofs are relatively short, this limitation has little impact on the proof rate while saving some compute.

We use a batch size of 32, and fine-tune for up to 100,000 steps, but we observed that the model begins to overfit to the training set after 50,000 to 70,000 steps. For inference, we selected checkpoints from just before the model started to overfit.

6.3 Evaluation

In this section we present several experiments and discuss the following research questions:

RQ1: How effective are LLMs at generating whole proofs?

RQ2: Can LLMs be used to repair proofs?

RQ3: Can LLMs benefit from using the context of the theorem?

RQ4: Does the size of the LLM affect proof synthesis effectiveness?

RQ5: How do LLMs compare to other SOTA proof generation methods?

Model	16 samples	64 samples
Baldur 8b generate	34.8%	40.7%
Baldur 8b generate + repair	36.3%*	—
Baldur 8b w/ context	40.9%	47.5%
Baldur 62b w/ context	42.2%	47.9%
Baldur 8b w/ context ∪ Thor	—	65.7%

Table 6.1: Proof rate of different models.
*The repair approach uses half the number of samples, and then one repair attempt for each sample.

To answer these questions, we trained several language models using the approach from Section 6.2, and evaluated them on the PISA benchmark (see Section 6.3.2). Our main results can be found in Table 6.1 and in Figure 6.4.

6.3.1 Experimental Setup

Machine specification For most of the training runs of the 8b model, we used 64 TPUv3 cores distributed across 8 hosts. For training the 62b model, we used 256 TPUv3 cores distributed across 32 hosts. For most inference jobs, we used 32 inference servers using 8 TPUv3 cores each.

Proof Checker We use the PISA codebase [100] under a BSD 3-clause license, which allows us to interact with the Isabelle proof assistant to check proofs. To run large jobs of the proof checker, we package it in a Docker container and run it on GCP. We extended the proof checker to discard any proofs that contain "sorry" or "oops", which are keywords that skip proofs, but otherwise pass the proof checker. We apply a timeout of 10 seconds to each proof step in the proof checker.

6.3.2 PISA Benchmark

We derive our datasets from the PISA dataset [100], which includes the Isabelle/HOL repository under a BSD-style license and the Archive of Formal Proofs (AFP) from October 2021. The AFP is a large collection of Isabelle/HOL proof developments. PISA includes the core higher-order logic library of Isabelle, as well as a diverse library of proofs formalised with Isabelle. This includes mathematics proofs and verification of software and hardware systems. The PISA dataset comes with a 95%/1%/4% split of theorems for the training/validation/test sets, which we follow in this work as well.

For the test set, prior work randomly chose 3,000 theorems from the test set to report their results on. We report our results on the complete test set. Some entries in the dataset are not proper theorems (starting with the keyword "lemmas" instead of "lemma"), which we filter out, as did prior work. This leaves us with a total of 6,336 theorems in our test set (originally 6,633 theorems).

It is worth noting that, as with any LLM-based work, there is the potential for proofs from the test set to have leaked into the LLM pretraining data. While the pretraining data for the Minerva LLM at the base of our models does not include the PISA dataset, it does contain code that may include some Isabelle/HOL proofs found in PISA. This should be kept in mind when interpreting the results.

6.3.3 RQ1: How effective are LLMs at generating whole proofs?

We aligned our methodology with the methodology described in the Thor paper [98] to enable a comparison between various methods. The Thor paper includes informative baselines for the PISA benchmark, including Sledgehammer, a method relying on heuristic search, and a language model approach using search.

Sledgehammer and the search-based language model approach achieve 25.6% and 39.0%, respectively. In comparison, our naive proof generation approach with an 8b language model achieves a proof rate of 34.8% with 16 samples and of 40.7% with 64 samples. The comparison is even more favorable, if we consider the other variants of Baldur, which achieve a proof rate of up to 47.9%.

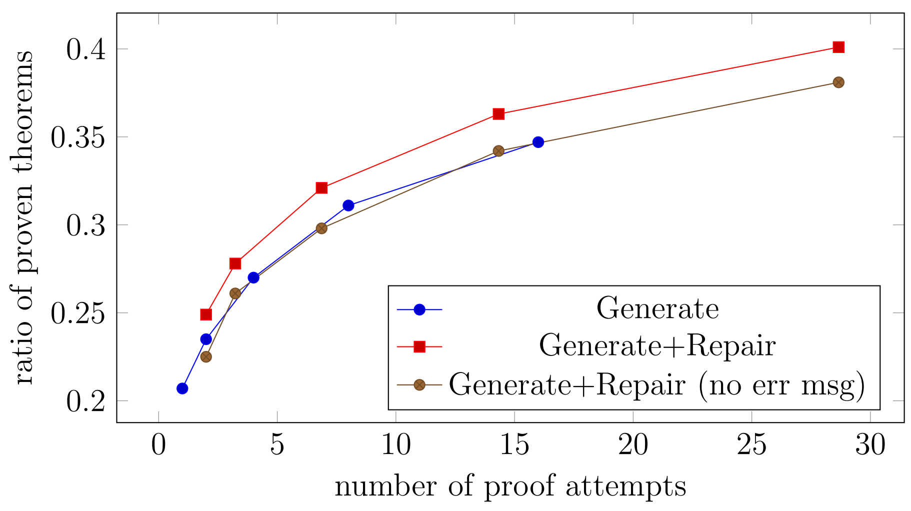


Figure 6.4: Ratio of theorems proven vs inference cost.

We observe that the comparison depends on the computational cost that we spend during inference. While comparing the cost required for the two methods is involved, one measure we can use is the amount of computational resources reserved during proof generation. For a single proof, the language model approach using search [98] requires a TPUv3 with 8 cores for 216 seconds,[2] while our methodology also requires a TPUv3 with 8 cores for around 35 seconds to sample 64 proofs – a difference of factor 6. This argument disregards the time spent on proof checking, which is intentional: proof checking is done on CPUs, which is cheap compared to time spent on TPUs. So, disentangling these two workloads can lead to significant reductions in computational cost.

RA1: These results demonstrate that LLMs can generate full proofs just as well as smaller language models augmented with a search strategy.

[2]Section 4.1 in [98] states that 1000 problems take around 60 TPU hours.

6.3.4 RQ2: Can LLMs be used to repair proofs?

We trained models for proof generation and repair as detailed in Section 6.2. If we sample from the proof generation model once with temperature 0, collect the failed proofs, and then repair once with temperature 0, we generate an additional 266 or 4.2% correct proofs. However, in this comparison, the generate + repair approach uses two samples, while the generate approach has only one sample. For a fair comparison, we have to compare the repair approach to the generate approach with additional inference attempts.

In Figure 6.4, we plot the proof success rate of the generate approach and the repair approach against the number of proof attempts. Note that the number of samples for the repair approach does not perfectly align with the number of samples for the generate approach. This is because the generate approach tends to produce multiple copies of the same proofs, which we deduplicate before repair, and only generate one repair attempt per failed proof attempt. For each of the number of samples of the generate approach, we tune the temperature in the range of 0.0 to 1.4 in increments of 0.2, and we always use temperature 0 for the repair approach.

We observe that the repair approach consistently outperforms the plain proof generation model, which only uses the theorem statement as input. However, this does not yet answer the question of where those gains from. To shed some light on this question, we trained another repair model that is given the same information, except that it does not see the error message. Plotting the proof success rate of this model in Figure 6.4 shows us that while it is able to prove additional theorems, it does not surpass the performance of the generate model when normalized for inference cost. This suggests that the information in the error message is crucial for the observed gains of the repair approach.

RA2: LLMs can be used to repair proofs, including their own failed proof attempts, and this can boost overall proving power.

6.3.5 RQ3: Can LLMs benefit from using the context of the theorem?

In Table 6.1, we report the impact of adding theory file context to our plain generation approach. At 64 samples, the proof rate increases from 40.7% to 47.5% for the same model size. In Figure 6.5, we plot the proof success rate of the generation model with and without context against the number of proof attempts. We observe that the proof generation models with context consistently outperform the plain generation model.

To get a better understanding of where these gains are coming from, we inspected 5 randomly sampled examples that the model using context was able to solve, but the plain generation model could not. Appendix B displays these examples and further details the process we used to select them.

While the sample size is not large enough to make quantitative judgements, it appears that the model frequently makes use of similar proofs in the context. We observe that for 3 of the 5 examples (see Appendices B.1, B.3, B.5) the model readily **copies and adapts** proofs that exist in its context. For another example (see Appendix B.2), the model made use of a premise that did not occur in its context, which happened to also be used in the ground truth proof, but with a different tactic. In the final example (see Appendix B.4), the model found a simpler proof that did not occur like this in the context. This suggests that the addition of context does not play the same role as premise selection.

RA3: LLMs can benefit from the context in which the theorem occurred in the theory file, both quantitatively by increasing proving power, and qualitatively by copying and adapting nearby proofs.

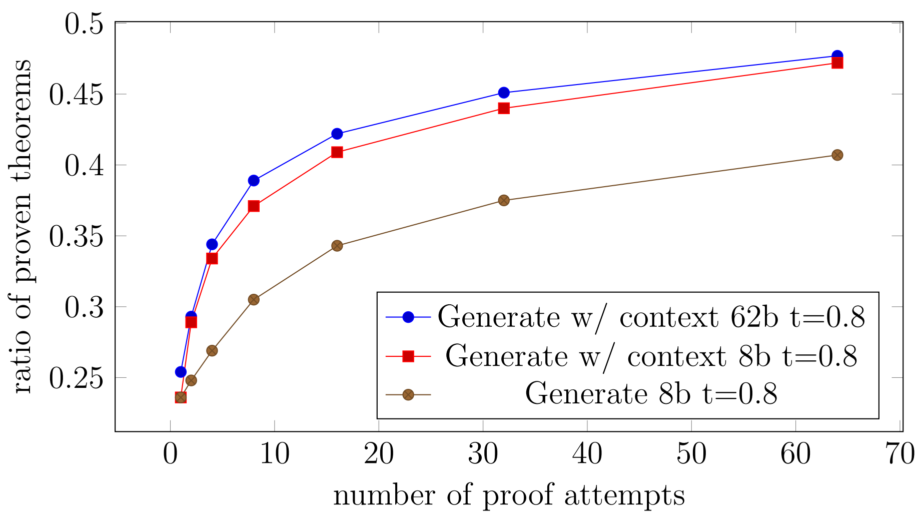


Figure 6.5: Ratio of theorems proven vs inference cost for models with different sizes and temperatures.

6.3.6 RQ4: Does the size of the LLM affect proof synthesis effectiveness?

We fine-tuned and evaluated the 62b version of Minerva on the proof generation task with context. In Table 6.1, we report that for 16 samples, the large model can prove an additional 1.3% over the 8b model, resulting in a total proof rate of 42.2%. For 64 samples, the large model can prove an additional 0.4% over the 8b model, resulting in a total proof rate of 47.9%.

In Figure 6.5, we plot the proof success rate of the generation model with context for the 8b model and the 62b model against the number of proof attempts. We observe that the 62b proof generation model with context outperforms the 8b proof generation model with context. One caveat here is that we were not able to tune hyperparameters as well due to the higher cost of these experiments, so an optimally tuned 62b model may perform even better.

RA4: Theorem proving performance improves somewhat with the scale of the language model.

AFP Topic	Test set	Baldur	Thor
Computer Science	4,019	50.0%	57.5%
Logic	966	51.6%	53.6%
Mathematics	2,200	41.9%	50.5%
Tools	102	53.9%	51.8%

Table 6.2: Proof rate by AFP topic classification, and the number of theorems in each category. While there are only 6336 theorems in total in the test set, the projects these theorems appear in can fall into multiple topics.

6.3.7 RQ5: How do LLMs compare to other SOTA proof generation methods?

While comparisons across different neural theorem provers are hard in general, we can compare to Thor [98], one of the strongest approaches available. Thor also relies on language models, but uses smaller models (700m parameters) and uses a different kind of proof step as its prediction target. Instead of using the human ground truth proofs, Thor generates a new training set and aims to solve each proof step by generating a declarative statement, which is then solved using Sledgehammer. That is, Thor disentangles the planning stage of the next proof step, which is the specification of the target state (using a "have" statement) and premise selection, which is done by Sledgehammer. This enables Thor to solve a total of 57% of the problems.

In contrast, our approach solves up to 47.9% of the problems. While there is a significant gap, we argue that the means by which the two techniques improve over plain language modeling are largely orthogonal. In Table 6.1, we report a large gain from 57% to 65.7% when we consider the union of Baldur and Thor, which supports this hypothesis.

We compare the proof rate of Baldur and Thor on different types of problems. The AFP is indexed by topic and there are four overarching topics: computer science, logic, mathematics, and tools. The authors of individual proof developments self-identity which topics their projects fall into. We use these provided topic labels to determine the categories of problems from our test that Baldur and Thor can most effectively solve. Table 6.2 shows the breakdown of which theorems in the test set fall into which topics and Baldur's and Thor's proof success rates on these theorems. In terms of relative performance, Baldur performs better than Thor on problems related to tools and similarly on problems related to logic. We observe that Baldur's performance on mathematics and computer science is less than that of Thor's performance. For mathematics proofs, we hypothesize that premise selection may be particularly useful, and Thor's use of Sledgehammer is likely what gives it a leg up on solving these mathematics problems.

RA5: Our findings suggest that LLM-based methods and search-based methods are complementary, and together can lead to large gains in proving power.

6.4 Contributions

This work is the first to fine-tune large language models to generate entire proofs of theorems without the need for proof search or hammers. We demonstrate that this approach is more effective and more efficient than prior methods that use one-step-at-a-time search-based generation, and that it is complementary to existing search-based and hammer-based approaches. Together, Baldur and prior tools can fully automatically synthesize proofs for 65.7% of the theorems in a large Isabelle/HOL benchmark, establishing a new state of the art. We further demonstrate that generate-and-repair improves proof synthesis when the language model is given access to the error messages produced by erroneous proofs.

This work opens new avenues of research into (1) using LLMs to automate theorem proving and simplify formal verification of software properties, (2) repair approaches, both for proofs and, potentially, more traditional automated program repair tasks, and (3) the use of context (e.g., failed synthesis attempts and error messages) in proof generation. Our very encouraging results suggest a bright future for automated proof generation and repair using LLMs.

CHAPTER 7
RELATED WORK

In this chapter, I place my research in the context of related work. The following is adapted directly from the related work sections within the publications and preprints corresponding to the prior chapters (refer to the start of those chapters to see the names of my collaborators).

7.1 Interactive Theorem Provers (ITPs)

ITPs, such as Coq [199], Agda [209], Lean [117], Dafny [119], F* [197], Liquid Haskel [207], Mizar [202], Isabelle [148], HOL4 [188], and HOL Light [84] are semi-automated systems for formally proving theorems. I focus on Coq and Isabelle, but my techniques applicable to other ITPs. Coq has been used to build and verify a C compiler [121], an operating system kernel [74], an x86 model [137], a file system [93], distributed protocols [185] and systems [217], a browser [97], and network controllers [75].

7.2 Automated Theorem Provers (ATPs)

ATPs, such as Z3 [52], Vampire [110], CVC4 [16], and E Prover [181], use automated methods to validate conjectures. They are used in practice to efficiently solve problems [152]. However, ATPs are based on first-order logic, which limits their ability to express more complex theorems in a higher-order logic.

7.3 Automation for Proof Systems

Heuristic-based search can partially automate ITPs [29, 30, 24, 5]. *Hammers* use external ATPs to automatically find proofs for ITPs [49]. Hammers, such as CoqHammer [49] and Sledgehammer [156], iteratively use a set of precomputed mathematical facts to attempt to "hammer" out a proof. While hammers are powerful, they lack the ability to employ certain tactics, such as induction, preventing them from proving certain large classes of theorems.

Classical search algorithms, such as A*, can also be used to search for proofs, which as been done in HOL4 [68]. TacticZero [218] learns not just tactics but also proof search strategies for end-to-end proof synthesis, rather than relying on a single fixed proof search strategy. The approach works by way of deep reinforcement learning, and improves over the previous state of the art on a benchmark for the HOL4 theorem prover. By contrast, my tools model existing proof scripts, use native tactics, and prove theorems within the ITP framework.

7.4 Neural Proof Synthesis

Neural proof synthesis tools use a prediction model that, given some information about a partially written proof, the target theorem being proven, and the current proof state, predicts a set of next likely proof steps. The methods then use metaheuristic search [83] to attempt to synthesize a proof. They iterate querying the prediction model for the likely next steps and, using the proof assistant to get feedback on those steps and prune non-promising paths, generate a search tree of possible proofs. The proof assistant also determines when the proof is complete. The tools mostly differ in the prediction model they use, which are typically learned automatically.

For Coq, ASTactic uses only the proof state [223], TacTok uses the proof state and the partially written proof script [62], Diva combines the use of many models and also uses the proof term [59], and Passport also uses identifier information [180]. These tools are all evaluated on the CoqGym benchmark suite [223].

Other neural proof synthesis tools for Coq include Proverbot9001 [179], Tactician [22], Gamepad [92], and ML4PG [109]. The models in these tools do not explicitly encode the category a particular identifier belongs to, do not encode the path that an identifier comes from, nor do they apply sub-word tokenization. Passport-style enhancements may help further improve performance of these tools.

Other neural proof synthesis tools for other proof assistants include TacticToe [69] for HOL4 and DeepHOL [13, 154] for HOL Light. Prior work has found that hammers and search-based methods are complementary, each often proving theorems the other cannot [223, 62, 59]. Thor [98] combines a search-based method with a hammer, using both a prediction model and Sledgehammer in its search. In contrast, my tool Baldur uses an LLM to generate an entire proof at once, and then repairs it in a single attempt.

The most closely related work to my tool Baldur is LISA [100], which fine-tunes a pretrained language model on a large Isabelle/HOL proof corpus, and uses it inside of a search procedure to predict proof steps (this work also introduces the PISA benchmark). GPT-f [161] also combines a generative language model with proof search to target the MetaMath proof language. A Monte-Carlo tree search approach outperforms GPT-f in the Lean proof assistant [112].

My tools TacTok, Diva, and Passport model the proof state, and are generalizations of ASTactic [223], which uses a Tree-LSTM architecture for the proof state. Other models in this space use sequences [179, 13, 22], other tree architectures [92], and graph architectures [154, 126] (which demonstrate improvements over tree architectures).

Recent work shows that the choice to encode variable names or not has a significant impact on the performance of a graph neural network for proof synthesis in HOL on the HOList benchmark suite [154]. My Passport tool explores this tradeoff at a higher level of granularity, looking at the impacts of including different kinds of variables and other syntactic information (such as paths), and exploring different tokenization choices and vocabulary sizes.

7.5 Automated Proof Repair

As demonstrated in the REPLica study, proof engineers continuously perform proof repair during formal proof development [173]. The effort surrounding the automation of this task started with symbolic tools for automatic proof repair in the Coq proof assistant [170]. These tools include Pumpkin Patch, which generates proof patches when software evolves [174] by learning from a template of a human-written fix to a similar evolution, and Pumpkin Pi, which repairs the proof term of a broken proof and then uses a decompiler to generate a proof script [172]. These techniques have since been integrated into tools for other proof systems [131].

My tool Baldur introduces the proof repair task with error messages. This is a new machine learning task for formal proofs. Baldur shows that solving this task improves neural theorem proving performance. My work is among the first to explore proof repair in a machine learning context, and the first to use error messages for a proof repair task, and to use repair to improve performance of proof synthesis.

7.6 Software Engineering for Interactive Proof Assistants

My tools can help minimize human effort in formal verification by automatically synthesizing proofs for some theorems. Other tools that assist humans writing formal verification proofs can similarly save time, and can be complementary to our work for theorems they cannot prove fully automatically. iCoq [31, 32], and its parallelized version PiCoq [155], find failing proof scripts in evolving projects by prioritizing proof scripts affected by a revision. iCoq tracks fine-grained dependencies between Coq definitions, propositions, and proof scripts to narrow down the potentially affected proof scripts. QuickChick [113], a property-based testing tool for Coq, searches for counterexamples to executable theorems, helping a programmer gain confidence that a theorem is correct. Roosterize [147, 145] suggests names for lemmas through using both syntactic and semantic information found when combining data from multiple phases of the Coq compiler, including tokens, parse trees, and fully elaborated terms.

Language models can also help automatically format proofs [146] by encoding spacing information in proof scripts, improving both readability and maintainability. Mutation analysis is useful for identifying weak specifications when mutating definitions does not break proofs [33, 96].

There are numerous other tasks that machine learning tools for proofs consider that may either help users with proof engineering directly, or improve neural theorem proving performance themselves. For example, PaMpeR [143] predicts proof methods alongside explanations in Isabelle/HOL. ACL2(ml) [88] generates helper lemmas and suggests similar theorems in ACL2. Other popular tasks leveraging machine learning include premise selection and datatype alignment (covered in more detail in the survey paper, QED at Large [171]).

7.7 Metaheuristic Search

Metaheuristic-search-based software engineering [83] has been used for developing test suites [135, 210], finding safety violations [4], refactoring [183], project management and effort estimation [15], and automated program repair [215, 105, 2]. In search, low-quality fitness functions can lead to low-quality results, such as, for example, incorrect bug patches [189, 139]. With my tools, the interactive theorem prover provides a strong assurance that the final produced proof script leads to a correct proof, and thus, proof script synthesis is particularly well suited for metaheuristic-search-based methods.

7.8 Ensemble Learning

Ensemble learning is the generation and combination of multiple models to make a decision. This is typically used in supervised machine learning tasks [176]. The idea is that weighing and combining several opinions is better than simply choosing a single one. When generating a model to be used in an ensemble learning method, the model should be sufficiently diverse for the ensemble to achieve a desired predictive performance [53], and the individual model's predictive performance should be as high as possible. There are several approaches to generating diverse models, including input manipulation [35], manipulation of the learning algorithm [127, 27, 133], and combinations of strategies.

Ensemble learning methods either have dependent models, where the output of each model affects the generation of the next, or independent models, where each model is constructed independently from the others [176]. Another way to combine classifiers is through stacking [55], which uses a classifier to decide which model to apply to each input. My tool Diva differs from these methods by using independent models in separate searches of the proof script space since the proof assistant serves as an oracle for whether the resulting proof scripts are valid.

7.9 Autoformalization

A related problem to neural proof synthesis is *autoformalization*: the automatic translation of natural language specifications and proofs into formal, machine-checkable specifications and proofs. LLMs have shown promise for autoformalization of specifications, and automatically generated proofs of the resulting autoformalized specifications have been used to improve a neural theorem prover on a widely used benchmark suite in Isabelle/HOL [219]. ProofNet [11] introduces a dataset and benchmark suite for autoformalization in Lean, based on undergraduate mathematics, and shows preliminary promising results autoformalizing proofs on that benchmark using Codex [38] with few-shot learning. Autoformalization of both theorems and proofs in Coq shows promise on a small preliminary benchmark suite [48]. Autoformalization for specification logics in verification is also promising [81].

The Draft, Sketch, and Prove method (DSP) [99] presents a hybrid between theorem proving and autoformalization, which, similar to the Baldur approach, makes use of LLMs for theorem proving. It provides informal proofs as drafts for the LLM to translate into a formal proof sketch, which is then proven via Sledgehammer. In contrast, we use fine-tuning for LLMs, do not make use of Sledgehammer, and do not rely on the availability of natural language proofs.

Pretrained language models can be used to answer natural-language mathematics questions [150]. Large language models, such as Minerva [123] and PaLM [43], have been evaluated on natural language mathematics benchmarks, such as GSM8k [45] and MATH [87]. The ProofNet [11] benchmark suite includes informal proofs alongside formal proofs as a benchmark.

7.10 Automated Program Repair

The automated program repair field studies the task of taking a program with a bug, evidenced by one or more failing tests, and automatically producing a modified version of the program that passes all the tests [116]. Generate-and-validate repair techniques use search-based techniques or predefined templates to generate many syntactic candidate patches, validating them against the tests (e.g., GenProg [115], Prophet [129], AE [214], HDRepair [114], ErrDoc [201], JAID [37], Qlose [51], and Par [106], ssFix [220], CapGen [216], SimFix [102], Hercules [178], Recoder [229], among others). Techniques such as DeepFix [79] and ELIXIR [177] use learned models to predict erroneous program locations, as well as the patches. It is possible to learn how to repair errors together by learning how to create errors, which can increase the amount of available training data, but poses an additional challenge of learning to approximate making human-like errors [224]. Unfortunately, these automated program repair techniques often overfit to the available tests and produce patches that, while passing all the tests, fail to encode the developers' intent [189, 140, 149, 164]. Improving the quality of the resulting repairs can be done via improving fault localization strategies [138, 8, 221, 194, 111, 101, 130], patch generation algorithms (e.g., heuristic-based [115, 129, 201, 216, 102, 158], constraint-based [2, 105, 211, 78, 134], and learning-based [40, 79, 177]), and patch validation methodologies [212, 222, 227, 200, 225]. By contrast, in the theorem proving domain, it is impossible to produce a proof that appears to prove the theorems, but actually fails to do so, because the theorem prover acts as an absolute oracle for the correctness of the proof. As a result, it may be more difficult to produce a proof in the first place, but if techniques in this domain do produce proofs, they are guaranteed to be correct.

7.11 Language Modeling for Code and Program Synthesis

Language modeling of source code can detect bugs and generate tests [168, 57, 1]. Modeling code with n-grams can help code completion [91, 90]. Modified n-grams can be used as a cache to capture local dependencies in code [203]. Recent work [103] demonstrates the benefits of BPE tokenization for code completion, especially in combination with cache-based models. Other researchers [196] have introduced a framework for evaluating different design decisions for integrating the structure within identifiers within a code completion model, and show similar benefits for BPE. VarCLR explores using contrastive learning to learn which identifiers have similar meanings, in contrast to simply being related [39]. It does this by mining variable renamings from GitHub edits, and enables effective use of general purpose language models.

The model beneath Github's Copilot code auto-complete tool, Codex, is trained on a large corpus of Github projects, and treats all programs and proofs as text, regardless of the language [38]. AlphaCode (from DeepMind) and PaLM-Coder (from

Google) solve a similar tasks [125, 43] Other work from Google [9] showed that large language models of this kind are promising, but struggle to understand the semantics of programs.

Applying language models to Coq and HOL4 proof scripts showed that n-gram models outperform recurrent neural networks [86]. Unlike my tools, this approach did not consider the proof state or proof term and does not synthesize complete proof scripts.

Several different models have also been proposed for modeling code, such as AST-like trees [141], long-term language models [50], and probabilistic grammars [20]. Program synthesis is also widely studied using non-learning based methods [77, 36], both from types alone [80] and examples and types [153, 64].

CHAPTER 8
CONCLUSION

8.1 Summary

In this dissertation, I developed various approaches to improve proof synthesis in Coq and Isabelle. In Chapter 3, I presented TacTok, which is the first proof synthesis tool to explore modeling proof state and the partially written proof script together. In Chapter 4, I presented Diva, which is a proof synthesis tool that explores the creation of diverse models and uses the theorem prover as an oracle for optimally aggregating model results. In Chapter 5, I presented Passport, which is a proof synthesis tool that investigates the use of different identifier encoding mechanisms for different categories of identifiers. In Chapter 6, I presented Baldur, which is the first proof synthesis tool to fine-tune large language models to generate and repair entire proofs of theorems without using proof search or hammers.

My tools TacTok, Diva, and Passport improve on CoqHammer and ASTactic to prove an additional 4.9% of the CoqGym test set. CoqHammer, ASTactic and my tools for Coq together prove 34.3% of the CoqGym test set. My tool Baldur improves on Thor to prove an additional 8.7% of the PISA test set. Together, Thor and Baldur prove 65.7% of the PISA test set.

This dissertation contributes new ideas for improving models of proof synthesis, and the results support that the improvement is significant. This dissertation also opens up new avenues of research that warrant continued exploration. By continuing to improve proof synthesis techniques, we can enable more verification and work towards a world with more bug-free software.

8.2 Future Work

This future work section was adapted directly from the discussion section of the Baldur preprint.

Going forward, there are a few directions that I believe are particularly promising and important to explore:

1. integrating proof generation and proof repair models into a new **learnable proof search** strategy,
2. exploring the **collaboration of diverse models** at search time,
3. investigating **alternative data splits** corresponding to different goals, and
4. evaluating techniques across **different proof assistants**.

8.2.1 Learnable Proof Search

While Baldur's generate + repair approach to proof synthesis avoids costly proof search procedures, it also lends itself to a new proof search strategy. The search strategy would work as follows:

1. use the generation model to sample candidate proofs,

2. use the repair model to attempt to repair those proofs, and

3. continue to use the repair model to repair the repair-model-generated attempts from (2).

This paves the way for a learnable proof search strategy.

During the development of Baldur, I demonstrated a proof-of-concept of this new proof search strategy using Baldur's generation and repair models. I sampled once using the generation model, repaired the generated sample using the repair model, and repaired the repair model's attempt using the repair model. When using both models, I sampled with temperature 0. So the inference cost in this setup is 3 (1 for the first generation, 1 for the first repair, and 1 for the second repair).

The generate + repair approach with inference cost of 2 proves 24.9% of the test set theorems. With a second repair attempt, it proves an additional 1.3%, for a total of 26.2%. The generation approach with inference cost of 3 proves 25.4%, which is 0.8% less than the second repair attempt for the same inference cost.

To make this a more viable proof search strategy, future work needs to focus on generating proof repair training data that better mirrors the required changes for the subsequent repair attempts. When proof checking, the resulting error message is for the first occurring error, typically from the first couple of lines of the predicted proof. So the proof repair model will only learn to address these types of errors. An alternative approach could be, for example, to take the training examples from the proof generation model and use the first few lines of the human-written ground truth proof as a *proof prefix*. One could then concatenate this proof prefix to the end of the input. Since it is a decoder-only model, one can simply sample the model's attempt at the rest of the proof. If the proof prefix concatenated with the rest of the proof does not check, then that can serve as a new training example for the proof repair model.

8.2.2 Collaborative Search

With Diva, each diverse model was part of its own tree-search. Even when efficiently combining the searches such that they interrupted one another, each search and the predictions from each model still remained separate and independent. A potentially fruitful avenue of research to explore is having the models collaborate at search time. One example of a collaboration method could be to have the models vote on their predictions at each step of the search.

8.2.3 Alternative Data Splits

The benchmarks that I use to evaluate my tools commit to particular data splits between training data and testing data. It is interesting to note, however, that different data splits may themselves correspond to different goals, even fixing the same evaluation task and metric. Moving forward, it may be useful to consider different kinds of data splits corresponding to different goals, even fixing the same dataset and benchmark suite. Here, I consider two different splits: *theorem-wise* and *project-wise*.

PISA [100] uses a random theorem-wise split of the theorems appearing the AFP. This means that for any theorem in the test set, the theorems and (the corresponding proofs) that appear before or after that theorem may be in the training set. This split is useful to evaluate since a forward-looking goal of neural theorem prover researchers is to integrate these tools directly into proof assistants, where they could make use of the full project context. That project context may include human-written proofs of nearby theorems that look similar (or even identical) to one another — automatically repurposing and adapting those proofs can be quite fruitful.

By contrast with PISA, CoqGym [223] uses a project-wise split, where training and testing data come from entirely different projects. This is useful when the goal is to help proof engineers who start completely new projects and want an automated proof synthesis tool to prove as much as it can. A tool that is trained and evaluated in a setting where it expects that it has seen proofs in a given proof development, as may happen with a theorem-wise split, may not perform as well in this new setting. Explicit consideration for the data split and the goals it achieves may help drive neural theorem proving research even further.

8.2.4 Different Proof Assistants

To make better sense of new strides in neural theorem proving, it makes sense to evaluate the same techniques across many different proof assistants. But this remains challenging. Consider once again the problem of data splits: since prover developments that evaluate on CoqGym, such as TacTok, Diva, and Passport, follow the same project-wise split as CoqGym, it can be hard to make sense of how those developments compare to those trained and evaluated using theorem-wise data splits, like Baldur.

With Baldur, for example, I used an established benchmark of Isabelle/HOL proofs to fairly compare Baldur to prior work and to increase the chances that the results generalize. However, we observed that search-based proof-synthesis tools for other proof assistants tend to prove a smaller fraction of theorems than we have found in for Baldur. For example, Diva and CoqHammer together prove 33.8% of the CoqGym benchmark automatically. This could be a reflection of size and quality of the available training data or the complexity of the available evaluation data (which, by necessity, is different from what we use because it involves theorems and proofs in different languages), or a more fundamental difference in the complexity of synthesizing proofs in these respective languages.

Future work should allow for direct comparisons by porting the developed techniques across proof assistants. Cross-proof-assistant benchmark suites may help substantially with this, but still have their limitations. For example, MiniF2F [228] implements the same benchmark suite for Math Olympiad problems across many different proof assistants. However, the benchmark does not implement the problems for Coq. This is because math problems are not evenly represented across proof assistants, which draw different user communities with different emphases. Thus, fair comparisons between proof assistants are hard, but we do believe they are necessary.

APPENDIX A
PASSPORT: CATEGORIES OF IDENTIFIERS

Before we implemented Passport, we manually inspected the proof corpora in our training dataset, walking through proofs and analyzing the kinds of information needed to make decisions about which tactic to apply next in a proof. The choice to include identifiers was a product of realizing how much proof engineers rely on naming information to reason about these decisions. But the choice of *which* identifiers to include was less clear. Consider, for example, local variables: many common local variable names are used in a variety of contexts which may have little relation with one another. A variable named `x` can carry a totally different meaning than the `x` from Figure 5.2 in Section 5.2. Without empirical evidence, it was unclear whether an enriched model could potentially suffer performance degradation from drawing fallacious connections like this. As a result, experimental data was an important factor in our selection of which identifiers to include.

Our experiments in Section 5.4 show that all three categories of identifiers help. In particular, search using the Tok model Passport-enriched with *any one* of the three categories of identifiers alone outperforms search using that model with no identifier information. Furthermore, a search using the Tok model Passport-enriched with *all three* categories of identifiers at once outperforms a search using a Passport-enriched Tok model with just one category of identifiers, for all categories.

The remainder of this Appendix details each of these three categories — global definitions (Appendix A.1), local variables (Appendix A.1.1), and type constructors

(Appendix A.2) — and gives intuition for why each of them may be useful for a tactic prediction model. Finally, Appendix A.3 discusses Passport implementation details.

A.1 Global Definitions

The most straightforward of our categories to include was identifiers referencing global definitions. These identifiers refer to objects defined globally directly by the user, using the keywords `Definition`, `Theorem`, `Inductive`, or one of their variants. Global definitions are generally either an inductive type name, or a name given to some Gallina term (function, constant value, etc). Crucially, since proof objects themselves are terms, theorems are global definitions with their names bound to their proof objects.

In Coq, most code amounts to creating new global definitions, through a variety of means. The simplest is by writing the term which corresponds to the name explicitly, and using a vernacular command to bind it to the name, as in `Definition n := 5.`. This is commonly how the `Definition` keyword is used, both in defining constant values and in defining functions. When a definition needs to refer to its own name within its body, that is done either using a `fix` in the term, or using the special vernacular keyword `Fixpoint`, which is essentially syntactic sugar for the former.

Global definitions can also be defined interactively, using Coq's tactic system. For example, the proof script in Figure 5.2 specifies a sequence of tactics which produce a Gallina term referred to by its identifier `posnatMult_comm`. In Gallina, this is indistinguishable from a plain definition — in fact, any term in Coq can be defined using tactics, though this is most common for proofs of lemmas and theorems.

Finally, inductive types can be created using Coq's `Inductive` command. This command creates a new inductive type or type family, given a set of "type constructors," or ways to build objects of the type. When complete, this command defines several objects, including the type itself, its type constructors, and recursion and induction principles for the type. Type constructors are explored in more detail in Appendix A.2.

Encoding the usage of global definitions in terms is extremely useful for predicting tactics. Often, a particular common identifier will signify that certain lemmas will be useful. For instance, in the proof context:

```
n : nat
============================
le (div2 n) n
```

the presence of the `div2` and `le` identifiers indicates that lemmas involving those operators will be useful; in fact, the correct next step is to apply a lemma named `div2_decr`, which applies to goals of the form `le (div2 _)_`. Both `div2` and `le` identifiers correspond to global definitions.

A.1.1 Local Variables

Besides global definitions, local variables are the most common type of identifier in Coq terms. Local variables can be bound to an explicit term, as in a `let` definition, but in many cases (function parameters, forall bindings, and existential pairs) are given only a type binding. This is in contrast to global definitions, which are always bound directly to terms.

Encoding local variables is often critical to determining the correct next step in a proof, or even understanding its basic structure. Even when the local variable's name is not particularly informative, knowing when local variables repeat is often critical. For example, consider the following proof context (from VST [6]):

```
n : nat
============================
n >= div2 n + div2 n
```

If the `n` variable were not the same in all three occurrences, this goal would be impossible to prove without more information. However, because the `n` variable is repeated, this goal holds by the definition of `div2`, which is round-down division by two.

While local variable names often provide useful information, as mentioned above, common names are often overloaded in their usage. We learned early on that the possibility of performance regression due to uninformative local variables like `x` had concerned the ASTactic authors, and contributed to their decision not to encode identifiers.[1] However, upon closer inspection of the data we determined that even single-letter identifier names often carry consistent semantic meaning across proofs. The identifier names `hd` and `tl`, for instance, seemed to uniformly refer to the head and tail of a list; because they carried consistent semantic meaning, these identifiers were treated similarly within proofs.

Because of these consistencies in naming, we decided to include local variables.

A.2 Type Constructors

Unlike global definitions and local variables, type constructors are not bound on their own, but are instead defined as part of inductive type definitions. As an example of how type constructors are defined, Figure A.1 shows the definition of the option type.

```
(* Library Coq, directory Init, file Datatypes.v *)
Inductive option¹ (A² : Type) : Type :=
| Some³ : A → option A
| None³ : option A
```

Figure A.1: The polymorphic `option` datatype in Coq, found in the fully-qualified path `Coq.Init.Datatypes`. Given a type parameter `A`, an `option A` in Coq is one of two things: either it is `Some a` given an element `a` of type `A`, or it is `None`. For consistency, identifiers are highlighted using the same conventions from Figure 5.1.

The type definition for `option` has two type constructors: `Some`, which creates an `option A` for any object of type `A`, and `None`, which is a constant value of type `option A`

[1] https://github.com/princeton-vl/CoqGym/discussions/60

```
1 subgoal
m, n : nat
E1 : ev n
E2 : ev m
IH1 : ev (n + m)
============================
ev (S (S (n + m)))
```

Figure A.2: A mid-proof context from the first volume of the logical foundations series [160]

for any `A`. There are many examples of such type constructors in common inductive types: `S` and `O` for natural numbers, `cons` and `nil` for lists, and others. Logically, just as type definitions correspond to theorems, type constructors are analogous to introduction rules for types. In the option type in Figure A.1, `Some` and `None` encode all possible ways of introducing terms of type `option`. Because of this, type constructors play a special role in deconstructing types — in particular, they appear inside match statements, which *act* on the structure of a type by having one branch per type constructor. Similarly, proofs by induction in Coq *prove* propositions about inductive types by having one case per type constructor.

Knowledge of type constructors can be incredibly useful in determining the next proof step in a proof. In the example from Figure A.2, the goal states that `S (S ( n + m))` is even, where `m` and `n` are natural numbers. The context shows `(n + m)` is even, but does not include information about `S`. The knowledge that `S` is a successor type constructor of `nat`, and that there exists an `ev` type constructor `ev_SS` of type `ev n -> ev (S (S n))`, is necessary to solve the goal. Here, running the `constructor` tactic results in the goal `ev (n + m)`, which matches one of the hypotheses (IH1).

A.3 Passport Enrichment Implementation

Enriching the data with these three categories of identifiers amounted to modifying inherited data processing code from TacTok and ASTactic that had erased all

```
(constructor
  (inductive
    (file_path
      (directory_path [Datatypes; Init; Coq])
      (label option¹)))
  (int 1³))
```

Figure A.3: An unprocessed AST representing a use of the `Some` type constructor for the `option` inductive type from Figure A.1, simplified for the sake of presentation. For consistency, identifiers are highlighted using the same conventions from Figure 5.1, and the index `1` of the `Some` type constructor is highlighted in `yellow`[3]. Note that the identifier of the `Some` type constructor itself is not present.

information about those identifiers from the data. The inherited code had used the SerAPI [7] library to serialize Coq proof objects (terms) as well as proof states and theorems (types), then processed the serialized ASTs returned by SerAPI to erase all identifier information. Enriching the data with two of the three categories of identifiers — definition and local variable names — was a straightforward modification of the post-processing code.

By contrast, adding type constructor names was a more involved process, as Gallina ASTs do not directly store type constructor names. Instead, like its parent type theory, the calculus of inductive constructions [47, 46], Coq represents each type constructor in the AST as a tuple consisting of the name of its inductive type together with the index of the particular type constructor.

Figure A.3 shows the AST for `Some`, which is the first (type constructors are 1-indexed) type constructor of the `option` datatype. Notably, the AST by default stores the fully-qualified path and name of the inductive type that the type constructor constructs. Thus, the only remaining step is to look up the type constructor from the global environment by passing the fully-qualified name of the inductive type and the index of the type constructor — here, `Coq.Init.Datatypes.option` and `1` — then place it back into the AST where the index is.

To do this, between parsing and encoding, the Passport implementation *unparses* subterms that correspond to type constructor nodes into string representations of the ASTs of the subterms. It then feeds those string representations back through SerAPI, which performs an environment lookup to recover the type constructor name. As with the other identifiers, Passport then inserts a child node containing the identifier into the AST before encoding.

Overall, the Passport approach implementation is 1.5K lines of code and took four developers about a year to build. While the conceptual and design aspects of the Passport approach can extend to all prediction-model-driven, search-based, proof-synthesis tools, the current implementation is straightforwardly applicable to all such tools built within the CoqGym environment [223].

APPENDIX B

BALDUR: EXAMPLES OF PROOF GENERATION WITH CONTEXT

We provide a number of examples that the model using context could solve but the plain proof generation model could not. We determined the lists of problems each model could solve, computed their difference, and then sampled 5 examples uniformly at random. For examples that had multiple correct proofs generated by the model, we selected one at random. We modified whitespace in the examples to make them more readable with the reduced line length. Further, we truncated the examples on the left to help with readability, but we inspected also the full context to ensure that our conclusions below are not affected. Each example consists of the "context and problem statement", the "ground truth proof", and the "generated proof".

We can observe in examples 1, 3, and 5 that the model readily **copies and adapts** proofs that exist in its context. In example 2, the model made use of a premise that did not occur in its context, which happened to also be used by the ground truth proof, but with a different tactic. In example 4, the model found a simpler proof that did not occur like this in the context.

B.1 Example 1

Context and problem statement:

```
lemma (in Interpretation) InterpExprWellDefined:
  "L\<lbrakk>Vx : A \<turnstile> e :
    B\<rbrakk> \<rightarrow> i \<Longrightarrow>
      Sig iS \<triangleright> Vx :
        A  \<turnstile> e : B"
  apply (rule Interp.cases)
  by auto

lemma (in Interpretation) WellDefined:
  "L\<lbrakk>\<phi>\<rbrakk> \<rightarrow> i
    \<Longrightarrow> Sig iS \<triangleright> \<phi>"
  apply(rule Interp.cases)
  by (auto simp add: InterpExprWellDefined)

lemma (in Interpretation) Bool:
  "L\<lbrakk>\<phi>\<rbrakk> \<rightarrow>
    (IBool i) \<Longrightarrow> \<exists> A B e d .
      \<phi> = (Vx : A \<turnstile> e \<equiv> d : B)"
```

Ground truth proof:

```
apply(rule Interp.cases)
by auto
```

Generated proof:

```
apply (rule Interp.cases)
by (auto simp add: InterpExprWellDefined)
```

B.2 Example 2

Context and problem statement:

```
lemma Aeta_expand_0[simp]:
  "Aeta_expand 0 e = e"
  by transfer simp

lemma Aeta_expand_inc[simp]:
  "Aeta_expand (inc\<cdot>n) e =
    (Lam [fresh_var e].
      Aeta_expand n (App e (fresh_var e)))"
  apply (simp add: inc_def)
  by transfer simp

lemma subst_Aeta_expand:
  "(Aeta_expand n e)[x::=y] =
    Aeta_expand n e[x::=y]"
```

Ground truth proof:

```
by transfer (rule subst_eta_expand)
```

Generated proof:

```
by transfer (simp add: subst_eta_expand)
```

B.3 Example 3

Context and problem statement:

```
subsubsection \<open>Store and Assignment\<close>

type_synonym \'a store = "string \<Rightarrow> \'a"

definition gets ::
  "string \<Rightarrow> (\'a store \<Rightarrow> \'a)
    \<Rightarrow> \'a store rel" ("_ ::= _" [70, 65] 61)
  where "v ::= e = {(s, s(v := e s)) |s. True}"

lemma H_assign:
  "rel_kat.H \<lceil>\<lambda>s.
    P (s (v := e s))\<rceil> (v ::= e) \<lceil>P\<rceil>"
  by (auto simp:
    gets_def
    rel_kat.H_def
    rel_kat.t_op_def
    rel_at_def)

lemma H_assign_var:
  "(\<forall>s. P s \<longrightarrow> Q (s (v := e s)))
    \<Longrightarrow>
    rel_kat.H
      \<lceil>P\<rceil> (v ::= e) \<lceil>Q\<rceil>"
```

Ground truth proof:

```
by (auto simp:
  gets_def
  rel_kat.H_def
  rel_kat.t_op_def
  rel_at_def)
```

Generated proof:

```
by (auto simp:
  gets_def
  rel_kat.H_def
  rel_kat.t_op_def
  rel_at_def)
```

B.4 Example 4

Context and problem statement:

```
section \<open>Primitive words\<close>

text\<open>If a word w is not a non-trivial power of
some other word, we say it is primitive.\<close>

definition primitive :: "\'a list \<Rightarrow> bool"
  where "primitive u = (\<forall> r k.
     r\<^sup>@k = u \<longrightarrow> k = 1)"

lemma primI[intro]:
  "(\<And> r k. r\<^sup>@k = u \<Longrightarrow> k = 1)
  \<Longrightarrow> primitive u"
  by (simp add: primitive_def)

lemma prim_nemp:
  "primitive u \<Longrightarrow> u \<noteq> \<epsilon>"
```

Ground truth proof:

```
proof-
have "u = \<epsilon> \<Longrightarrow>
  \<epsilon>\<^sup>@0 = u"
by simp
thus "primitive u \<Longrightarrow>
  u \<noteq> \<epsilon>"
using primitive_def zero_neq_one
by blast
qed
```

Generated proof:

```
using primitive_def
by force
```

B.5 Example 5

Context and problem statement:

```
lemma drop_take_drop:
  "drop n (take m xs) @ drop m xs = drop (min n m) xs"
proof cases
  assume A: "\<not>m < n"
  then show ?thesis
```

```
    using drop_append[of n "take m xs" "drop m xs"]
  by (cases "length xs < n")
       (auto simp: not_less min_def)
qed (auto simp: min_def)

lemma not_distinct_decomp_min_prefix:
  assumes "\<not> distinct ws"
  shows "\<exists> xs ys zs y.
    ws = xs @ y # ys @ y # zs \<and>
    distinct xs \<and>
    y \<notin> set xs \<and>
    y \<notin> set ys "
proof -
  obtain xs y ys where
    "y \<in> set xs" "distinct xs" "ws = xs @ y # ys"
    using assms by (auto simp: not_distinct_conv_prefix)
  moreover then obtain xs\' ys\'
    where "xs = xs\' @ y # ys\'"
    by (auto simp: in_set_conv_decomp)
  ultimately show ?thesis by auto
qed

lemma not_distinct_decomp_min_not_distinct:
  assumes "\<not> distinct ws"
  shows "\<exists>xs y ys zs.
    ws = xs @ y # ys @ y # zs \<and>
    distinct (ys @ [y])"
```

Ground truth proof:

```
using assms
proof (induct ws)
  case (Cons w ws)
  show ?case
  proof (cases "distinct ws")
    case True
    then obtain xs ys where
        "ws = xs @ w # ys" "w \<notin> set xs"
      using Cons.prems
        by (fastforce dest: split_list_first)
    then have "distinct
        (xs @ [w])" "w # ws = [] @ w # xs @ w # ys"
      using \<open>distinct ws\<close> by auto
    then show ?thesis by blast
  next
    case False
    then obtain xs y ys zs
        where "ws = xs @ y # ys @ y # zs \<and>
               distinct (ys @ [y])"
      using Cons by auto
    then have "w # ws = (w # xs) @ y # ys @ y # zs \<and>
               distinct (ys @ [y])"
      by simp
    then show ?thesis by blast
  qed
qed simp
```

Generated proof:

```
proof -
  obtain xs y ys where
    "y \<in> set xs" "distinct xs" "ws = xs @ y # ys"
  using assms by (auto simp: not_distinct_conv_prefix)
  moreover then obtain xs' ys'
    where "xs = xs' @ y # ys'"
    by (auto simp: in_set_conv_decomp)
  ultimately
  show ?thesis by auto
qed
```

BIBLIOGRAPHY

[1] Abou-Assaleh, Tony, Cercone, Nick, Keselj, Vlado, and Sweidan, Ray. N-gram-based detection of new malicious code. In *Annual International IEEE Computer Software and Applications Conference* (2004), vol. 2, pp. 41–42.

[2] Afzal, Afsoon, Motwani, Manish, Stolee, Kathryn T., Brun, Yuriy, and Le Goues, Claire. SOSRepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering (TSE) 47*, 10 (October 2021), 2162–2181.

[3] Agda Development Team. The Agda wiki, 2007-2021.

[4] Alba, Enrique, and Chicano, Francisco. Finding safety errors with ACO. In *Conference on Genetic and Evolutionary Computation (GECCO)* (London, England, UK, July 2007), pp. 1066–1073.

[5] Andrews, Peter B, and Brown, Chad E. TPS: A hybrid automatic-interactive system for developing proofs. *Journal of Applied Logic 4*, 4 (2006), 367–395.

[6] Appel, Andrew W. Verified software toolchain. In *Programming Languages and Systems* (Berlin, Heidelberg, 2011), Gilles Barthe, Ed., Springer Berlin Heidelberg, pp. 1–17.

[7] Arias, Emilio Jesús Gallego. Serapi: Machine-friendly, data-centric serialization for coq.

[8] Assiri, Fatmah Yousef, and Bieman, James M. Fault localization for automated program repair: Effectiveness, performance, repair correctness. *Software Quality Journal 25*, 1 (2017), 171–199.

[9] Austin, Jacob, Odena, Augustus, Nye, Maxwell, Bosma, Maarten, Michalewski, Henryk, Dohan, David, Jiang, Ellen, Cai, Carrie J., Terry, Michael, Le, Quoc V., and Sutton, Charles. Program synthesis with large language models. *CoRR abs/2108.07732* (2021).

[10] AWS Provable Security. `https://aws.amazon.com/security/provable-security`.

[11] Azerbayev, Zhangir, Piotrowski, Bartosz, and Avigad, Jeremy. ProofNet: A benchmark for autoformalizing and formally proving undergraduate-level mathematics problems. In *Workshop MATH-AI: Toward Human-Level Mathematical Reasoning* (New Orleans, Louisiana, USA, Nov. 2022).

[12] Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)* (San Diego, CA, USA, 2015).

[13] Bansal, Kshitij, Loos, Sarah M., Rabe, Markus N., Szegedy, Christian, and Wilcox, Stewart. Holist: An environment for machine learning of higher-order theorem proving. 454–463.

[14] Barredo Arrieta, Alejandro, Díaz-Rodríguez, Natalia, Del Ser, Javier, Bennetot, Adrien, Tabik, Siham, Barbado, Alberto, Garcia, Salvador, Gil-Lopez, Sergio, Molina, Daniel, Benjamins, Richard, Chatila, Raja, and Herrera, Francisco. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information Fusion 58* (2020), 82–115.

[15] Barreto, Ahilton, Barros, Márcio, and Werner, Cláudia. Staffing a software project: A constraint satisfaction approach. *Computers and Operations Research 35*, 10 (2008), 3073–3089.

[16] Barrett, Clark, Conway, Christopher L., Deters, Morgan, Hadarean, Liana, Jovanović, Dejan, King, Tim, Reynolds, Andrew, and Tinelli, Cesare. CVC4. In *International Conference on Computer Aided Verification (CAV)* (Snowbird, UT, USA, July 2011), vol. 6806, Springer, pp. 171–177.

[17] BedRock Systems Inc. `https://bedrocksystems.com`.

[18] Bengio, Yoshua, Ducharme, Réjean, Vincent, Pascal, and Jauvin, Christian. A neural probabilistic language model. *Journal of Machine Learning Research 3*, Feb. (2003), 1137–1155.

[19] Bessey, Al, Block, Ken, Chelf, Ben, Chou, Andy, Fulton, Bryan, Hallem, Seth, Henri-Gros, Charles, Kamsky, Asya, McPeak, Scott, and Engler, Dawson. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM 53*, 2 (Feb. 2010), 66–75.

[20] Bielik, Pavol, Raychev, Veselin, and Vechev, Martin. Phog: Probabilistic model for code. In *Proceedings of The 33rd International Conference on Machine Learning* (New York, New York, USA, 20–22 Jun 2016), Maria Florina Balcan and Kilian Q. Weinberger, Eds., vol. 48 of *Proceedings of Machine Learning Research*, PMLR, pp. 2933–2942.

[21] Biewald, Lukas. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[22] Blaauwbroek, Lasse, Urban, Josef, and Geuvers, Herman. Tactic learning and proving for the coq proof assistant. In *LPAR23. LPAR-23: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning* (2020), Elvira Albert and Laura Kovacs, Eds., vol. 73 of *EPiC Series in Computing*, EasyChair, pp. 138–150.

[23] Black, Sidney, Biderman, Stella, Hallahan, Eric, Anthony, Quentin, Gao, Leo, Golding, Laurence, He, Horace, Leahy, Connor, McDonell, Kyle, Phang, Jason, Pieler, Michael, Prashanth, Usvsn Sai, Purohit, Shivanshu, Reynolds, Laria, Tow, Jonathan, Wang, Ben, and Weinbach, Samuel. GPT-NeoX-20B: An open-source autoregressive language model. In *Proceedings of BigScience Episode #5 – Workshop on Challenges & Perspectives in Creating Large Language Models* (virtual+Dublin, May 2022), Association for Computational Linguistics, pp. 95–136.

[24] Blanchette, Jasmin Christian, Bulwahn, Lukas, and Nipkow, Tobias. Automatic proof and disproof in Isabelle/HOL. In *International Symposium on Frontiers of Combining Systems* (2011), Springer, pp. 12–27.

[25] Blanchette, Jasmin Christian, Kaliszyk, Cezary, Paulson, Lawrence C, and Urban, Josef. Hammering towards QED. *Journal of Formalized Reasoning 9*, 1 (2016), 101–148.

[26] Brill, Eric, and Moore, Robert C. An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics* (Hong Kong, 2000), pp. 286–293.

[27] Brown, Gavin, Wyatt, Jeremy L, Tino, Peter, and Bengio, Yoshua. Managing diversity in regression ensembles. *Journal of machine learning research (JMLR) 6*, 9 (2005).

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

[29] Bundy, Alan. A science of reasoning. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods* (1998), Springer, pp. 10–17.

[30] Bundy, Alan, Harmelen, Frank Van, Horn, Christian, and Smaill, Alan. The oser-clm system. In *International Conference on Automated Deduction (CADE)* (1990), Springer, pp. 647–648.

[31] Celik, Ahmet, Palmskog, Karl, and Gligoric, Milos. ICoq: Regression proof selection for large-scale verification projects. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Urbana-Champaign, IL, USA, 2017), pp. 171–182.

[32] Celik, Ahmet, Palmskog, Karl, and Gligoric, Milos. A regression proof selection tool for Coq. In *International Conference on Software Engineering Demonstrations Track (ICSE DEMO)* (Gothenburg, Sweden, 2018), pp. 117–120.

[33] Celik, Ahmet, Palmskog, Karl, Parovic, Marinela, Arias, Emilio Jesús Gallego, and Gligoric, Milos. Mutation analysis for Coq. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (San Diego, California, 2019), pp. 539–551.

[34] Certora. `https://www.certora.com`.

[35] Chan, Philip K, and Stolfo, Salvatore J. A comparative evaluation of voting and meta-learning on partitioned data. In *Machine Learning Proceedings*. Elsevier, 1995, pp. 90–98.

[36] Chaudhuri, Swarat, Ellis, Kevin, Polozov, Oleksandr, Singh, Rishabh, Solar-Lezama, Armando, Yue, Yisong, et al. Neurosymbolic programming. *Foundations and Trends® in Programming Languages 7*, 3 (2021), 158–243.

[37] Chen, Liushan, Pei, Yu, and Furia, Carlo A. Contract-based program repair without the contracts. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Urbana, IL, USA, Nov. 2017), pp. 637–647.

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

[39] Chen, Qibin, Lacomis, Jeremy, Schwartz, Edward J., Neubig, Graham, Vasilescu, Bogdan, and Goues, Claire Le. Varclr: Variable semantic representation pre-training via contrastive learning. In *Proceedings of the 44th International Conference on Software Engineering* (New York, NY, USA, 2022), ICSE '22, Association for Computing Machinery, p. 2327–2339.

[40] Chen, Zimin, Kommrusch, Steve James, Tufano, Michele, Pouchet, Louis-Noël, Poshyvanyk, Denys, and Monperrus, Martin. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *TSE 47*, 9 (2019), 1943–1959.

[41] Chlipala, Adam. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press, 2013.

[42] Cho, Kyunghyun, van Merriënboer, Bart, Gulcehre, Caglar, Bahdanau, Dzmitry, Bougares, Fethi, Schwenk, Holger, and Bengio, Yoshua. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)* (Doha, Qatar, Oct. 2014), pp. 1724–1734.

[43] Chowdhery, Aakanksha, Narang, Sharan, Devlin, Jacob, Bosma, Maarten, Mishra, Gaurav, Roberts, Adam, Barham, Paul, Chung, Hyung Won, Sutton, Charles, Gehrmann, Sebastian, Schuh, Parker, Shi, Kensen, Tsvyashchenko, Sasha, Maynez, Joshua, Rao, Abhishek, Barnes, Parker, Tay, Yi, Shazeer, Noam, Prabhakaran, Vinodkumar, Reif, Emily, Du, Nan, Hutchinson, Ben, Pope, Reiner, Bradbury, James, Austin, Jacob, Isard, Michael, Gur-Ari, Guy, Yin, Pengcheng, Duke, Toju, Levskaya, Anselm, Ghemawat, Sanjay, Dev, Sunipa, Michalewski, Henryk, Garcia, Xavier, Misra, Vedant, Robinson, Kevin, Fedus, Liam, Zhou, Denny, Ippolito, Daphne, Luan, David, Lim, Hyeontaek, Zoph, Barret, Spiridonov, Alexander, Sepassi, Ryan, Dohan, David, Agrawal, Shivani, Omernick, Mark, Dai, Andrew M., Pillai, Thanumalayan Sankaranarayana, Pellat, Marie, Lewkowycz, Aitor, Moreira, Erica, Child, Rewon, Polozov, Oleksandr, Lee, Katherine, Zhou, Zongwei, Wang, Xuezhi, Saeta, Brennan, Diaz, Mark, Firat, Orhan, Catasta, Michele, Wei, Jason, Meier-Hellstern, Kathy, Eck, Douglas, Dean, Jeff, Petrov, Slav, and Fiedel, Noah. Palm: Scaling language modeling with pathways. *CoRR abs/2204.02311* (2022).

[44] Chung, Junyoung, Gulcehre, Caglar, Cho, KyungHyun, and Bengio, Yoshua. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *Deep Learning and Representation Learning Workshop (DL&RL)* (2014).

[45] Cobbe, Karl, Kosaraju, Vineet, Bavarian, Mohammad, Hilton, Jacob, Nakano, Reiichiro, Hesse, Christopher, and Schulman, John. Training verifiers to solve math word problems. *CoRR abs/2110.14168* (2021).

[46] Coquand, Thierry, and Huet, Gérard. The calculus of constructions. Tech. Rep. RR-0530, INRIA, May 1986.

[47] Coquand, Thierry, and Paulin, Christine. Inductively defined types. In *COLOG-88* (Berlin, Heidelberg, 1990), Per Martin-Löf and Grigori Mints, Eds., Springer Berlin Heidelberg, pp. 50–66.

[48] Cunningham, Garett, Bunescu, Razvan C., and Juedes, David. Towards autoformalization of mathematics and code correctness: Experiments with elementary proofs, 2023.

[49] Czajka, Łukasz, and Kaliszyk, Cezary. Hammer for coq: Automation for dependent type theory. *Journal of Automated Reasoning 61*, 1 (Jun 2018), 423–453.

[50] Dam, Hoa Khanh, Tran, Truyen, and Pham, Trang. A deep language model for software code. *CoRR abs/1608.02715* (2016).

[51] D'Antoni, Loris, Samanta, Roopsha, and Singh, Rishabh. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification (CAV)* (Toronto, ON, Canada, July 2016), pp. 383–401.

[52] de Moura, Leonardo, and Bjørner, Nikolaj. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems 4963* (Apr. 2008), 337–340.

[53] Deng, Houtao, Runger, George, Tuv, Eugene, and Vladimir, Martyanov. A time series forest for classification and feature extraction. *Information Sciences 239* (2013), 142–153.

[54] Devlin, Jacob, Chang, Ming-Wei, Lee, Kenton, and Toutanova, Kristina. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)* (Minneapolis, MN, USA, 2019), pp. 4171–4186.

[55] Džeroski, Saso, and Ženko, Bernard. Is combining classifiers with stacking better than selecting the best one? *Machine learning 54*, 3 (2004), 255–273.

[56] Erbsen, Andres, Philipoom, Jade, Gross, Jason, Sloan, Robert, and Chlipala, Adam. Simple high-level code for cryptographic arithmetic — with proofs, without compromises. In *IEEE Symposium on Security and Privacy (S&P)* (2019), pp. 1202–1219.

[57] Ernst, Michael D. Natural language is a programming language: Applying natural language processing to software development. In *Summit on Advances in Programming Languages (SNAPL)* (Dagstuhl, Germany, 2017), vol. 71, pp. 4:1–4:14.

[58] Fan, Angela, Lewis, Mike, and Dauphin, Yann. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Melbourne, Australia, July 2018), Association for Computational Linguistics, pp. 889–898.

[59] First, Emily, and Brun, Yuriy. Diversity-driven automated formal verification. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)* (Pittsburgh, PA, USA, May 2022).

[60] First, Emily, and Brun, Yuriy. Replication package for "diversity-driven automated verification". `https://doi.org/10.5281/zenodo.5903318`, 2022.

[61] First, Emily, Brun, Yuriy, and Guha, Arjun. Replication package for "TacTok: Semantics-aware proof synthesis", 2020.

[62] First, Emily, Brun, Yuriy, and Guha, Arjun. TacTok: Semantics-aware proof synthesis. *Proceedings of the ACM on Programming Languages (PACMPL) Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) issue 4* (November 2020), 231:1–231:31.

[63] First, Emily, Rabe, Markus N, Ringer, Talia, and Brun, Yuriy. Baldur: Whole-proof generation and repair with large language models. *arXiv preprint arXiv:2303.04910* (2023).

[64] Frankle, Jonathan, Osera, Peter-Michael, Walker, David, and Zdancewic, S. Example-directed synthesis: a type-theoretic interpretation. *ACM SIGPLAN Notices 51* (01 2016), 802–815.

[65] Gage, Philip. A new algorithm for data compression. *C Users J. 12*, 2 (feb 1994), 23–38.

[66] Galois, Inc. `https://galois.com`.

[67] Gao, Xiang. cub device scan is not deterministic as described in the documentation #454. `https://github.com/NVIDIA/cub/issues/454`, 2022.

[68] Gauthier, Thibault, Kaliszyk, Cezary, and Urban, Josef. TacticToe: Learning to reason with HOL4 tactics. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)* (2017), vol. 46, pp. 125–143.

[69] Gauthier, Thibault, Kaliszyk, Cezary, Urban, Josef, Kumar, Ramana, and Norrish, Michael. Tactictoe: Learning to prove with tactics. *J. Autom. Reason. 65*, 2 (feb 2021), 257–286.

[70] Gers, Felix A, Schmidhuber, Jürgen, and Cummins, Fred. Learning to forget: Continual prediction with lstm. *Neural Computation 12*, 10 (1999), 2451–2471.

[71] Gilpin, Leilani H., Bau, David, Yuan, Ben Z., Bajwa, Ayesha, Specter, Michael A., and Kagal, Lalana. Explaining explanations: An approach to evaluating interpretability of machine learning. *CoRR abs/1806.00069* (2018).

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

[73] Greff, Klaus, Srivastava, Rupesh K, Koutník, Jan, Steunebrink, Bas R, and Schmidhuber, Jürgen. LSTM: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS) 28*, 10 (2017), 2222–2232.

[74] Gu, Ronghui, Shao, Zhong, Chen, Hao, Wu, Xiongnan, Kim, Jieung, Sjöberg, Vilhelm, and Costanzo, David. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (2016).

[75] Guha, Arjun, Reitblatt, Mark, and Foster, Nate. Machine verified network controllers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Seattle, WA, USA, 2013).

[76] Guidotti, Riccardo, Monreale, Anna, Ruggieri, Salvatore, Turini, Franco, Giannotti, Fosca, and Pedreschi, Dino. A survey of methods for explaining black box models. *ACM Comput. Surv. 51*, 5 (aug 2018).

[77] Gulwani, Sumit, Polozov, Oleksandr, Singh, Rishabh, et al. Program synthesis. *Foundations and Trends® in Programming Languages 4*, 1-2 (2017), 1–119.

[78] Gulwani, Sumit, Radiček, Ivan, and Zuleger, Florian. Automated clustering and program repair for introductory programming assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Philadelphia, PA, USA, June 2018), pp. 465–480.

[79] Gupta, Rahul, Pal, Soham, Kanade, Aditya, and Shevade, Shirish K. DeepFix: Fixing common C language errors by deep learning. In *National Conference on Artificial Intelligence (AAAI)* (San Francisco, CA, USA, Feb. 2017), pp. 1345–1351.

[80] Gvero, Tihomir, Kuncak, Viktor, Kuraj, Ivan, and Piskac, Ruzica. Complete completion using types and weights. *PLDI 2013* (2013), 12. 27–38.

[81] Hahn, Christopher, Schmitt, Frederik, Tillman, Julia J., Metzger, Niklas, Siber, Julian, and Finkbeiner, Bernd. Formal specifications from natural language. *CoRR abs/2206.01962* (2022).

[82] Han, Jesse Michael, Rute, Jason, Wu, Yuhuai, Ayers, Edward W, and Polu, Stanislas. Proof artifact co-training for theorem proving with language models.

[83] Harman, Mark. The current state and future of search based software engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2007), pp. 342–357.

[84] Harrison, John. HOL Light: A tutorial introduction. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)* (Palo Alto, CA, USA, 1996), pp. 265–269.

[85] Heim, Lennart. Estimating PaLM's training cost. `https://blog.heim.xyz/author/lennart/`, 2022.

[86] Hellendoorn, Vincent J., Devanbu, Premkumar T., and Alipour, Mohammad Amin. On the naturalness of proofs. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) New Ideas and Emerging Results track* (Orlando, FL, USA, Nov. 2018), pp. 724–728.

[87] Hendrycks, Dan, Burns, Collin, Kadavath, Saurav, Arora, Akul, Basart, Steven, Tang, Eric, Song, Dawn, and Steinhardt, Jacob. Measuring mathematical problem solving with the MATH dataset. *CoRR abs/2103.03874* (2021).

[88] Heras, Jónathan, and Komendantskaya, Ekaterina. Acl2(ml): Machine-learning for ACL2. In *Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications, Vienna, Austria, 12-13th July 2014.* (2014), pp. 61–75.

[89] Heras, Jónathan, and Komendantskaya, Ekaterina. Recycling proof patterns in coq: Case studies. *Mathematics in Computer Science 8*, 1 (2014), 99–116.

[90] Hindle, Abram, Barr, Earl T., Gabel, Mark, Su, Zhendong, and Devanbu, Premkumar. On the naturalness of software. *Communications of the ACM (CACM) 59*, 5 (Apr. 2016), 122–131.

[91] Hindle, Abram, Barr, Earl T, Su, Zhendong, Gabel, Mark, and Devanbu, Premkumar. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)* (2012), pp. 837–847.

[92] Huang, Daniel, Dhariwal, Prafulla, Song, Dawn, and Sutskever, Ilya. Gamepad: A learning environment for theorem proving. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019* (2019), OpenReview.net.

[93] İleri, Atalay, Chajed, Tej, Chlipala, Adam, Kaashoek, M. Frans, and Zeldovich, Nickolai. Proving confidentiality in a file system using DISKSEC. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Carlsbad, CA, 2018), pp. 323–338.

[94] Isabelle Development Team. Isabelle, 1994-2021.

[95] Jacobs, Kevin, and Beurdouche, Benjamin. Performance improvements via formally-verified cryptography in Firefox. `https://blog.mozilla.org/security/2020/07/06/performance-improvements-via-formally-verified-cryptography-in-firefox/`, 2020.

[96] Jain, Kush, Palmskog, Karl, Celik, Ahmet, Arias, Emilio Jesús Gallego, and Gligoric, Milos. MCoq: Mutation analysis for Coq verification projects. In *International Conference on Software Engineering Demonstrations Track (ICSE DEMO)* (Seoul, South Korea, 2020), pp. 89–92.

[97] Jang, Dongseok, Tatlock, Zachary, and Lerner, Sorin. Establishing browser security guarantees through formal shim verification. In *USENIX Security Symposium (USENIX Security)* (Bellevue, WA, USA, Aug. 2012), pp. 113–128.

[98] Jiang, Albert, Czechowski, Konrad, Jamnik, Mateja, Milos, Piotr, Tworkowski, Szymon, Li, Wenda, and Wu, Yuhuai Tony. Thor: Wielding hammers to integrate language models and automated theorem provers. In *Neural Information Processing Systems (NeurIPS)* (2022).

[99] Jiang, Albert Q., Welleck, Sean, Zhou, Jin Peng, Li, Wenda, Liu, Jiacheng, Jamnik, Mateja, Lacroix, Timothée, Wu, Yuhuai, and Lample, Guillaume. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *CoRR abs/2210.12283* (2022).

[100] Jiang, Albert Qiaochu, Li, Wenda, Han, Jesse Michael, and Wu, Yuhuai. LISA: Language models of ISAbelle proofs. In *Conference on Artificial Intelligence and Theorem Proving (AITP* (Aussois, France, September 2021), pp. 17.1–17.3.

[101] Jiang, Jiajun, Xiong, Yingfei, and Xia, Xin. A manual inspection of Defects4J bugs and its implications for automatic program repair. *Science China Information Sciences 62*, 10 (2019), 200102.

[102] Jiang, Jiajun, Xiong, Yingfei, Zhang, Hongyu, Gao, Qing, and Chen, Xiangqun. Shaping program repair space with existing patches and similar code. In *ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (Amsterdam, The Netherlands, July 2018), pp. 298–309.

[103] Karampatsis, Rafael Michael, Babii, Hlib, Robbes, Romain, Sutton, Charles, and Janes, Andrea. Big Code != Big Vocabulary: Open-Vocabulary Models for Source code. In *Proceedings of the 42nd International Conference on Software Engineering* (2020), ICSE '20, ACM.

[104] Katz, Slava. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing 35*, 3 (1987), 400–401.

[105] Ke, Yalin, Stolee, Kathryn T., Le Goues, Claire, and Brun, Yuriy. Repairing programs with semantic code search. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Lincoln, NE, USA, November 2015), pp. 295–306.

[106] Kim, Dongsun, Nam, Jaechang, Song, Jaewoo, and Kim, Sunghun. Automatic patch generation learned from human-written patches. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (San Francisco, CA, USA, 2013), pp. 802–811.

[107] Kneser, Reinhard, and Ney, Hermann. Improved backing-off for m-gram language modeling. In *International Conference on Acoustics, Speech, and Signal Processing* (Detroit, MI, USA, 1995), vol. 1, pp. 181–184.

[108] Koehn, Philipp, Hoang, Hieu, Birch, Alexandra, Callison-Burch, Chris, Federico, Marcello, Bertoldi, Nicola, Cowan, Brooke, Shen, Wade, Moran, Christine, Zens, Richard, et al. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the ACL; Interactive Poster and Demonstration Session* (2007), pp. 177–180.

[109] Komendantskaya, Ekaterina, Heras, Jónathan, and Grov, Gudmund. Machine learning in proof general: Interfacing interfaces. In *Proceedings 10th International Workshop On User Interfaces for Theorem Provers, UITP 2012, Bremen, Germany, July 11th, 2012* (2012), Cezary Kaliszyk and Christoph Lüth, Eds., vol. 118 of *EPTCS*, pp. 15–41.

[110] Kovács, Laura, and Voronkov, Andrei. First-order theorem proving and Vampire. In *International Conference on Computer Aided Verification (CAV)* (Saint Petersburg, Russia, 2013), vol. 8044, Springer-Verlag, pp. 1–35.

[111] Koyuncu, Anil, Liu, Kui, Bissyandé, Tegawendé F, Kim, Dongsun, Monperrus, Martin, Klein, Jacques, and Traon, Yves Le. iFixR: Bug report driven program repair. In *ESEC/FSE* (2019), pp. 314–325.

[112] Lample, Guillaume, Lachaux, Marie-Anne, Lavril, Thibaut, Martinet, Xavier, Hayat, Amaury, Ebner, Gabriel, Rodriguez, Aurélien, and Lacroix, Timothée. Hypertree proof search for neural theorem proving. *CoRR abs/2205.11491* (2022).

[113] Lampropoulos, Leonidas, Paraskevopoulou, Zoe, and Pierce, Benjamin C. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages (PACMPL) 2*, POPL (Dec. 2017), 45:1–45:30.

[114] Le, Xuan Bach D., Lo, David, and Le Goues, Claire. History driven program repair. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (March 2016), vol. 1, pp. 213–224.

[115] Le Goues, Claire, Nguyen, ThanhVu, Forrest, Stephanie, and Weimer, Westley. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE) 38* (2012), 54–72.

[116] Le Goues, Claire, Pradel, Michael, and Roychoudhury, Abhik. Automated program repair. *CACM 62*, 12 (Nov. 2019), 56–65.

[117] Lean Development Team. Theorem proving in lean, 2014-2021.

[118] Lebese, Thabang, Makondo, Ndivhuwo, Cornelio, Cristina, and Khan, Naweed. Proof extraction for logical neural networks. In *Advances in Programming Languages and Neurosymbolic Systems Workshop* (2021).

[119] Leino, K. Rustan M. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)* (Dakar, Senegal, 2010).

[120] Lepikhin, Dmitry, Lee, HyoukJoong, Xu, Yuanzhong, Chen, Dehao, Firat, Orhan, Huang, Yanping, Krikun, Maxim, Shazeer, Noam, and Chen, Zhifeng. Gshard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations* (2020).

[121] Leroy, Xavier. Formal verification of a realistic compiler. *Communications of the ACM (CACM) 52*, 7 (2009), 107–115.

[122] Leroy, Xavier. Formal verification of a realistic compiler. *Communications of the ACM 52*, 7 (2009), 107–115.

[123] Lewkowycz, Aitor, Andreassen, Anders, Dohan, David, Dyer, Ethan, Michalewski, Henryk, Ramasesh, Vinay V., Slone, Ambrose, Anil, Cem, Schlag, Imanol, Gutman-Solo, Theo, Wu, Yuhuai, Neyshabur, Behnam, Gur-Ari, Guy, and Misra, Vedant. Solving quantitative reasoning problems with language models. *CoRR abs/2206.14858* (2022).

[124] Li, Yangguang, Liang, Feng, Zhao, Lichen, Cui, Yufeng, Ouyang, Wanli, Shao, Jing, Yu, Fengwei, and Yan, Junjie. Supervision exists everywhere: A data efficient contrastive language-image pre-training paradigm. In *International Conference on Learning Representations* (2022).

[125] Li, Yujia, Choi, David, Chung, Junyoung, Kushman, Nate, Schrittwieser, Julian, Leblond, Rémi, Eccles, Tom, Keeling, James, Gimeno, Felix, Lago, Agustin Dal, Hubert, Thomas, Choy, Peter, d'Autume, Cyprien de Masson, Babuschkin, Igor, Chen, Xinyun, Huang, Po-Sen, Welbl, Johannes, Gowal, Sven, Cherepanov, Alexey, Molloy, James, Mankowitz, Daniel J., Robson, Esme Sutherland, Kohli, Pushmeet, de Freitas, Nando, Kavukcuoglu, Koray, and Vinyals, Oriol. Competition-level code generation with alphacode, 2022.

[126] Li, Zhaoyu, Chen, Binghong, and Si, Xujie. Graph contrastive pre-training for effective theorem reasoning. In *International Conference on Machine Learning (ICML)* (2021), vol. PLMR 139.

[127] Lin, Shih-Wei, and Chen, Shih-Chieh. Parameter determination and feature selection for C4.5 algorithm using scatter search approach. *Soft Computing 16*, 1 (2012), 63–75.

[128] Liu, Liyuan, Liu, Xiaodong, Gao, Jianfeng, Chen, Weizhu, and Han, Jiawei. Understanding the difficulty of training transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (Online, Nov. 2020), Association for Computational Linguistics, pp. 5747–5763.

[129] Long, Fan, and Rinard, Martin. Automatic patch generation by learning correct code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (St. Petersburg, FL, USA, 2016), pp. 298–312.

[130] Lou, Yiling, Ghanbari, Ali, Li, Xia, Zhang, Lingming, Zhang, Haotian, Hao, Dan, and Zhang, Lu. Can automated program repair refine fault localization? a unified debugging approach. In *ISSTA* (2020), pp. 75–87.

[131] Masci, Paolo, and Dutle, Aaron. Proof mate: An interactive proof helper for pvs (tool paper). In *NASA Formal Methods Symposium* (2022), Springer, pp. 809–815.

[132] Mauborgne, Laurent. Astrée: Verification of absence of runtime error. In *Building the Information Society* (2004), pp. 385–392.

[133] Maudes, Jesús, Rodríguez, Juan J, and García-Osorio, César. Disturbing neighbors diversity for decision forests. In *Applications of supervised and unsupervised ensemble methods*. Springer, 2009, pp. 113–133.

[134] Mechtaev, Sergey, Nguyen, Manh-Dung, Noller, Yannic, Grunske, Lars, and Roychoudhury, Abhik. Semantic program repair using a reference implementation. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Gothenburg, Sweden, 2018), pp. 129–139.

[135] Michael, Christoph C., McGraw, Gary, and Schatz, Michael A. Generating software test data by evolution. *IEEE Transactions on Software Engineering (TSE) 27*, 12 (Dec. 2001), 1085–1110.

[136] Mikolov, Tomáš, Karafiát, Martin, Burget, Lukáš, Černockỳ, Jan, and Khudanpur, Sanjeev. Recurrent neural network based language model. In *Annual Conference of the International Speech Communication Association (INTERSPEECH)* (Makuhari, Chiba, Japan, 2010).

[137] Morrisett, Greg, Tan, Gang, Tassarotti, Joseph, Tristan, Jean-Baptiste, and Gan, Edward. RockSalt: Better, faster, stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Beijing, China, 2012).

[138] Motwani, Manish, and Brun, Yuriy. Better automatic program repair by using bug reports and tests together. In *International Conference on Software Engineering (ICSE)* (Melbourne, Australia, May 2023).

[139] Motwani, Manish, Soto, Mauricio, Brun, Yuriy, Just, René, and Le Goues, Claire. Quality of automated program repair on real-world defects. *IEEE Transactions on Software Engineering (TSE)* (2021). 10.1109/TSE.2020.2998785.

[140] Motwani, Manish, Soto, Mauricio, Brun, Yuriy, Just, René, and Le Goues, Claire. Quality of automated program repair on real-world defects. *TSE 48*, 2 (February 2022), 637–661.

[141] Mou, Lili, Li, Ge, Jin, Zhi, Zhang, Lu, and Wang, Tao. TBCNN: A tree-based convolutional neural network for programming language processing. *CoRR abs/1409.5718* (2014).

[142] Murray, Toby, Matichuk, Daniel, Brassil, Matthew, Gammie, Peter, Bourke, Timothy, Seefried, Sean, Lewis, Corey, Gao, Xin, and Klein, Gerwin. seL4: From general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy (S&P)* (San Francisco, CA, USA, May 2013), pp. 415–429.

[143] Nagashima, Yutaka, and He, Yilun. PaMpeR: Proof method recommendation system for Isabelle/HOL. In *International Conference on Automated Software Engineering (ASE)* (Montpellier, France, 2018), p. 362–372.

[144] Nettleton, David F., Orriols-Puig, Albert, and Fornells, Albert. A study of the effect of different types of noise on the precision of supervised learning techniques. *Artificial Intelligence Review 33* (2010), 275–306.

[145] Nie, Pengyu, Palmskog, Karl, Li, Junyi Jessy, and Gligoric, Milos. Deep generation of Coq lemma names using elaborated terms. In *International Joint Conference on Automated Reasoning (IJCAR)* (Paris, France, June 2020), pp. 97–118.

[146] Nie, Pengyu, Palmskog, Karl, Li, Junyi Jessy, and Gligoric, Milos. Learning to format Coq code using language models. In *The Coq Workshop* (Aubervilliers, France, 2020).

[147] Nie, Pengyu, Palmskog, Karl, Li, Junyi Jessy, and Gligoric, Milos. Roosterize: Suggesting lemma names for Coq verification projects using deep learning. In *International Conference on Software Engineering Demonstrations Track (ICSE DEMO)* (Madrid, Spain, 2021), pp. 21–24.

[148] Nipkow, Tobias, Paulson, Lawrence C, and Wenzel, Markus. *Isabelle/HOL: A proof assistant for higher-order logic*, vol. 2283. Springer Science & Business Media, 2002.

[149] Noda, Kunihiro, Nemoto, Yusuke, Hotta, Keisuke, Tanida, Hideo, and Kikuchi, Shinji. Experience report: How effective is automated program repair for industrial software? In *SANER* (2020), pp. 612–616.

[150] Noorbakhsh, Kimia, Sulaiman, Modar, Sharifi, Mahdi, Roy, Kallol, and Jamshidi, Pooyan. Pretrained language models are symbolic mathematics solvers too! *CoRR abs/2110.03501* (2021).

[151] Nye, Maxwell I., Andreassen, Anders Johan, Gur-Ari, Guy, Michalewski, Henryk, Austin, Jacob, Bieber, David, Dohan, David, Lewkowycz, Aitor, Bosma, Maarten, Luan, David, Sutton, Charles, and Odena, Augustus. Show your work: Scratchpads for intermediate computation with language models. *CoRR abs/2112.00114* (2021).

[152] Ohrimenko, Olga, Stuckey, Peter J, and Codish, Michael. Propagation via lazy clause generation. *Constraints 14* (2009), 357–391.

[153] Osera, Peter-Michael, and Zdancewic, Steve. Type-and-example-directed program synthesis. *SIGPLAN Not. 50*, 6 (June 2015), 619–630.

[154] Paliwal, Aditya, Loos, Sarah, Rabe, Markus, Bansal, Kshitij, and Szegedy, Christian. Graph representations for higher-order logic and theorem proving. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2020), vol. 34, pp. 2967–2974.

[155] Palmskog, Karl, Celik, Ahmet, and Gligoric, Milos. PiCoq: Parallel regression proving for large-scale verification projects. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (Amsterdam, Netherlands, 2018), pp. 344–355.

[156] Paulson, Larry, and Nipkow, Tobias. The Sledgehammer: Let automatic theorem provers write your Isabelle scripts! `https://isabelle.in.tum.de/website-Isabelle2009-1/sledgehammer.html`, 2023.

[157] Peters, Matthew E., Neumann, Mark, Iyyer, Mohit, Gardner, Matt, Clark, Christopher, Lee, Kenton, and Zettlemoyer, Luke. Deep contextualized word representations. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)* (New Orleans, LA, USA, June 2018), vol. 1, Association for Computational Linguistics, pp. 2227–2237.

[158] Petke, Justyna, and Blot, Aymeric. Refining fitness functions in test-based program repair. In *APR* (2018), p. 13–14.

[159] Pham, Hung Viet, Qian, Shangshu, Wang, Jiannan, Lutellier, Thibaud, Rosenthal, Jonathan, Tan, Lin, Yu, Yaoliang, and Nagappan, Nachiappan. Problems and opportunities in training deep learning software systems: An analysis of variance. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2020), ASE '20, Association for Computing Machinery, p. 771–783.

[160] Pierce, Benjamin C., de Amorim, Arthur Azevedo, Casinghino, Chris, Gaboardi, Marco, Greenberg, Michael, Hriţcu, Cătălin, Sjöberg, Vilhelm, and Yorgey, Brent. *Software Foundations*, vol. 1: Logical Foundations. 2021.

[161] Polu, Stanislas, and Sutskever, Ilya. Generative language modeling for automated theorem proving. *CoRR abs/2009.03393* (2020).

[162] Pope, Reiner, Douglas, Sholto, Chowdhery, Aakanksha, Devlin, Jacob, Bradbury, James, Levskaya, Anselm, Heek, Jonathan, Xiao, Kefan, Agrawal, Shivani, and Dean, Jeff. Efficiently scaling transformer inference, 2022.

[163] Popel, Martin, and Bojar, Ondřej. Training tips for the transformer model. *The Prague Bulletin of Mathematical Linguistics 110*, 1 (2018), 43–70.

[164] Qi, Zichao, Long, Fan, Achour, Sara, and Rinard, Martin. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis (ISSTA)* (Baltimore, MD, USA, 2015), pp. 24–36.

[165] Qian, Shangshu, Pham, Viet Hung, Lutellier, Thibaud, Hu, Zeou, Kim, Jungwon, Tan, Lin, Yu, Yaoliang, Chen, Jiahao, and Shah, Sameena. Are my deep learning systems fair? an empirical study of fixed-seed training. In *Advances in Neural Information Processing Systems* (2021), M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, Eds., vol. 34, Curran Associates, Inc., pp. 30211–30227.

[166] Raffel, Colin, Shazeer, Noam, Roberts, Adam, Lee, Katherine, Narang, Sharan, Matena, Michael, Zhou, Yanqi, Li, Wei, and Liu, Peter J. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res. 21* (2020), 140:1–140:67.

[167] Rajbhandari, Samyam, Rasley, Jeff, Ruwase, Olatunji, and He, Yuxiong. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), SC '20, IEEE Press.

[168] Ray, Baishakhi, Hellendoorn, Vincent, Godhane, Saheel, Tu, Zhaopeng, Bacchelli, Alberto, and Devanbu, Premkumar. On the naturalness of buggy code. In *IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (Austin, TX, USA, 2016), pp. 428–439.

[169] Reichel, Tom P. Large cumulative sums appear to be nondeterministic. #75240. `https://github.com/pytorch/pytorch/issues/75240`, 2022.

[170] Ringer, Talia. *Proof Repair*. PhD thesis, University of Washington, 2021.

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

[172] Ringer, Talia, Porter, RanDair, Yazdani, Nathaniel, Leo, John, and Grossman, Dan. Proof repair across type equivalences. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)* (June 2021), pp. 112–127.

[173] Ringer, Talia, Sanchez-Stern, Alex, Grossman, Dan, and Lerner, Sorin. Replica: Repl instrumentation for coq analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New York, NY, USA, 2020), CPP 2020, Association for Computing Machinery, p. 99–113.

[174] Ringer, Talia, Yazdani, Nathaniel, Leo, John, and Grossman, Dan. Adapting proof automation to adapt proofs. In *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)* (Los Angeles, CA, USA, 2018), pp. 115–129.

[175] Rumelhart, David E., Hinton, Geoffrey E., and Williams, Ronald J. Learning representations by back-propagating errors. *Nature 323* (1986), 533–536.

[176] Sagi, Omer, and Rokach, Lior. Ensemble learning: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 8*, 4 (2018), e1249.

[177] Saha, Ripon K., Lyu, Yingjun, Yoshida, Hiroaki, and Prasad, Mukul R. ELIXIR: Effective object oriented program repair. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Urbana, IL, USA, Nov. 2017), pp. 648–659.

[178] Saha, Seemanta, Saha, Ripon K., and Prasad, Mukul R. Harnessing evolution for multi-hunk program repair. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Montreal, QC, Canada, May 2019), pp. 13–24.

[179] Sanchez-Stern, Alex, Alhessi, Yousef, Saul, Lawrence, and Lerner, Sorin. Generating correctness proofs with neural networks. In *ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)* (2020), pp. 1–10.

[180] Sanchez-Stern, Alex, First, Emily, Zhou, Timothy, Kaufman, Zhanna, Brun, Yuriy, and Ringer, Talia. Passport: Improving automated formal verification using identifiers. *ACM TOPLAS* (2023).

[181] Schulz, Stephan. System description: E 1.8. In *Logic for Programming, Artificial Intelligence, and Reasoning* (2013), Ken McMillan, Aart Middeldorp, and Andrei Voronkov, Eds., Springer Berlin Heidelberg, pp. 735–743.

[182] Sculley, D., Holt, Gary, Golovin, Daniel, Davydov, Eugene, Phillips, Todd, Ebner, Dietmar, Chaudhary, Vinay, and Young, Michael. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)* (2014).

[183] Seng, Olaf, Stammel, Johannes, and Burkhart, David. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Conference on Genetic and Evolutionary Computation (GECCO)* (Seattle, WA, USA, July 2006), pp. 1909–1916.

[184] Sennrich, Rico, Haddow, Barry, and Birch, Alexandra. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Berlin, Germany, Aug. 2016), Association for Computational Linguistics, pp. 1715–1725.

[185] Sergey, Ilya, Wilcox, James R., and Tatlock, Zachary. Programming and proving with distributed protocols. *Proceedings of the ACM on Programming Languages (PACMPL) 2*, POPL (Dec. 2017), 28:1–28:30.

[186] Shamir, Gil, and Lin, Dong. Reproducibility in deep learning and smooth activations. `https://ai.googleblog.com/2022/04/reproducibility-in-deep-learning-and.html?m=1`, 2022.

[187] Shazeer, Noam. Fast transformer decoding: One write-head is all you need. *CoRR abs/1911.02150* (2019).

[188] Slind, Konrad, and Norrish, Michael. A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)* (Montreal, QC, Canada, 2008), pp. 28–32.

[189] Smith, Edward K., Barr, Earl, Le Goues, Claire, and Brun, Yuriy. Is the cure worse than the disease? overfitting in automated program repair. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* (Bergamo, Italy, September 2015), pp. 532–543.

[190] Socher, Richard, Perelygin, Alex, Wu, Jean, Chuang, Jason, Manning, Christopher D, Ng, Andrew, and Potts, Christopher. Recursive deep models for semantic compositionality over a sentiment treebank. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2013), pp. 1631–1642.

[191] Souyris, Jean. Industrial use of compcert on a safety-critical software product. `http://projects.laas.fr/IFSE/FMF/J3/slides/P05_Jean_Souyiris.pdf`, 2014.

[192] Stolcke, Andreas. SRILM — An extensible language modeling toolkit. In *Proceedings of the International Conference on Spoken Language Processing* (2002).

[193] Su, Jianlin, Lu, Yu, Pan, Shengfeng, Wen, Bo, and Liu, Yunfeng. Roformer: Enhanced transformer with rotary position embedding. *CoRR abs/2104.09864* (2021).

[194] Sun, Shuyao, Guo, Junxia, Zhao, Ruilian, and Li, Zheng. Search-based efficient automated program repair using mutation and fault localization. In *COMPSAC* (2018), vol. 1, pp. 174–183.

[195] Sundermeyer, Martin, Schlüter, Ralf, and Ney, Hermann. LSTM neural networks for language modeling. In *Annual Conference of the International Speech Communication Association (INTERSPEECH)* (Portland, OR, USA, 2012).

[196] Svyatkovskiy, Alexey, Lee, Sebastian, Hadjitofi, Anna, Riechert, Maik, Franco, Juliana, and Allamanis, Miltiadis. Fast and memory-efficient neural code completion. *CoRR abs/2004.13651* (2020).

[197] Swamy, Nikhil, Hriţcu, Cătălin, Keller, Chantal, Rastogi, Aseem, Delignat-Lavaud, Antoine, Forest, Simon, Bhargavan, Karthikeyan, Fournet, Cédric, Strub, Pierre-Yves, Kohlweiss, Markulf, Zinzindohoue, Jean-Karim, and Zanella-Béguelin, Santiago. Dependent types and multi-monadic effects in f. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (St. Petersburg, FL, USA, 2016), vol. 51, pp. 256–270.

[198] Tai, Kai Sheng, Socher, Richard, and Manning, Christopher D. Improved semantic representations from tree-structured long short-term memory networks. In *Annual Meeting of the Association for Computational Linguistics (ACL)* (Beijing, China, July 2015), vol. 1, pp. 1556–1566.

[199] The Coq Development Team. Coq, v.8.7. `https://coq.inria.fr`, 2017.

[200] Tian, Haoye, Liu, Kui, Kaboré, Abdoul Kader, Koyuncu, Anil, Li, Li, Klein, Jacques, and Bissyandé, Tegawendé F. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *ASE* (2020).

[201] Tian, Yuchi, and Ray, Baishakhi. Automatically diagnosing and repairing error handling bugs in C. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany, Sept. 2017), pp. 752–762.

[202] Trybulec, Andrzej, and Blair, Howard A. Computer assisted reasoning with MIZAR. In *International Joint Conferences on Artificial Intelligence (IJCAI)* (Los Angeles, CA, USA, 1985), vol. 85, pp. 26–28.

[203] Tu, Zhaopeng, Su, Zhendong, and Devanbu, Premkumar. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, Association for Computing Machinery, p. 269–280.

[204] van den Oord, Aäron, Dieleman, Sander, Zen, Heiga, Simonyan, Karen, Vinyals, Oriol, Graves, Alex, Kalchbrenner, Nal, Senior, Andrew W., and Kavukcuoglu, Koray. Wavenet: A generative model for raw audio. *CoRR abs/1609.03499* (2016).

[205] van Rooyen, Brendan, Menon, Aditya, and Williamson, Robert C. Learning with symmetric label noise: The importance of being unhinged. In *Advances in Neural Information Processing Systems* (2015), vol. 28, Curran Associates, Inc.

[206] Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N, Kaiser, Łukasz, and Polosukhin, Illia. Attention is all you need. In *NeurIPS* (2017).

[207] Vazou, Niki. *Liquid Haskell: Haskell as a theorem prover.* PhD thesis, University of California, San Diego, 2016.

[208] Vechev, Martin, and Yahav, Eran. Programming with "big code". *Foundations and Trends® in Programming Languages 3*, 4 (2016), 231–284.

[209] Wadler, Philip, Kokke, Wen, and Siek, Jeremy G. *Programming Language Foundations in Agda.* July 2020.

[210] Walcott, Kristen R., Soffa, Mary Lou, Kapfhammer, Gregory M., and Roos, Robert S. Time-aware test suite prioritization. In *International Symposium on Software Testing and Analysis (ISSTA)* (Portland, ME, USA, July 2006), pp. 1–12.

[211] Wang, Ke, Singh, Rishabh, and Su, Zhendong. Search, align, and repair: Data-driven feedback generation for introductory programming exercises. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Philadelphia, PA, USA, June 2018), pp. 481–495.

[212] Wang, Shangwen, Wen, Ming, Lin, Bo, Wu, Hongjun, Qin, Yihao, Zou, Deqing, Mao, Xiaoguang, and Jin, Hai. Automated patch correctness assessment: How far are we? In *ASE* (2020), Association for Computing Machinery, p. 968–980.

[213] Wei, Jason, Wang, Xuezhi, Schuurmans, Dale, Bosma, Maarten, Chi, Ed H., Le, Quoc, and Zhou, Denny. Chain of thought prompting elicits reasoning in large language models. *CoRR abs/2201.11903* (2022).

[214] Weimer, Westley, Fry, Zachary P., and Forrest, Stephanie. Leveraging program equivalence for adaptive program repair: Models and first results. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Palo Alto, CA, USA, 2013), pp. 356–366.

[215] Weimer, Westley, Nguyen, ThanhVu, Le Goues, Claire, and Forrest, Stephanie. Automatically finding patches using genetic programming. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Vancouver, BC, Canada, 2009), pp. 364–374.

[216] Wen, Ming, Chen, Junjie, Wu, Rongxin, Hao, Dan, and Cheung, Shing-Chi. Context-aware patch generation for better automated program repair. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Gothenburg, Sweden, June 2018), pp. 1–11.

[217] Wilcox, James R., Woos, Doug, Panchekha, Pavel, Tatlock, Zachary, Wang, Xi, Ernst, Michael D., and Anderson, Thomas. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015), PLDI '15, ACM, pp. 357–368.

[218] Wu, Minchao, Norrish, Michael, Walder, Christian, and Dezfouli, Amir. TacticZero: Learning to prove theorems from scratch with deep reinforcement learning. *CoRR abs/2102.09756* (2021).

[219] Wu, Yuhuai, Jiang, Albert Q., Li, Wenda, Rabe, Markus N., Staats, Charles, Jamnik, Mateja, and Szegedy, Christian. Autoformalization with large language models. *CoRR abs/2205.12615* (2022).

[220] Xin, Qi, and Reiss, Steven P. Identifying test-suite-overfitted patches through test case generation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (Santa Barbara, CA, USA, 2017), pp. 226–236.

[221] Yang, Deheng, Qi, Yuhua, and Mao, Xiaoguang. Evaluating the strategies of statement selection in automated program repair. In *SATE* (2018), Springer.

[222] Yang, Jinqiu, Zhikhartsev, Alexey, Liu, Yuefei, and Tan, Lin. Better test cases for better automated program repair. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (2017), pp. 831–841.

[223] Yang, Kaiyu, and Deng, Jia. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning (ICML)* (Long Beach, CA, USA, 2019).

[224] Yasunaga, Michihiro, and Liang, Percy. Break-it-fix-it: Unsupervised learning for program repair. In *International Conference on Machine Learning (ICML)* (2021), PMLR, pp. 11941–11952.

[225] Ye, He, Martinez, Matias, and Monperrus, Martin. Automated patch assessment for program repair at scale. *EMSE 26*, 2 (2021).

[226] Yin, Pengcheng, and Neubig, Graham. A syntactic neural model for general-purpose code generation. In *Annual Meeting of the Association for Computational Linguistics (ACL)* (Vancouver, BC, Canada, July 2017), vol. 1, pp. 440–450.

[227] Yu, Zhongxing, Martinez, Matias, Danglot, Benjamin, Durieux, Thomas, and Monperrus, Martin. Alleviating patch overfitting with automatic test generation: A study of feasibility and effectiveness for the Nopol repair system. *Empirical Software Engineering (EMSE) 24*, 1 (Feb. 2019), 33–67.

[228] Zheng, Kunhao, Han, Jesse Michael, and Polu, Stanislas. miniF2F: A cross-system benchmark for formal Olympiad-level mathematics. In *ICLR* (2022).

[229] Zhu, Qihao, Sun, Zeyu, an Xiao, Yuan, Zhang, Wenjie, Yuan, Kang, Xiong, Yingfei, and Zhang, Lu. A syntax-guided edit decoder for neural program repair. In *ESEC/FSE* (2021), pp. 341–353.