

DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY


Guided Proof Search Using Large Language Models and Lemma Extraction in Coq

Citation

Prasad, Tarun. 2024. Guided Proof Search Using Large Language Models and Lemma Extraction in Coq. Bachelor's thesis, Harvard University Engineering and Applied Sciences.

Link

https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37379274

Terms of use

This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material (LAA), as set forth at

https://harvardwiki.atlassian.net/wiki/external/NGY5NDE4ZjgzNTc5NDQzMGIzZWZhMGFlOWI2M2EwYTg

Accessibility

https://accessibility.huit.harvard.edu/digital-accessibility-policy

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story

Guided Proof Search Using Large Language Models and Lemma Extraction in Coq

A DISSERTATION PRESENTED
BY
TARUN PRASAD
TO
THE DEPARTMENT OF COMPUTER SCIENCE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
BACHELOR OF ARTS WITH HONORS
IN THE SUBJECT OF
COMPUTER SCIENCE AND MATHEMATICS

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
MAY 2024

Thesis advisor: Prof. Nada Amin Tarun Prasad

Guided Proof Search Using Large Language Models and Lemma Extraction in Coq

Abstract

Interactive theorem provers are powerful tools for formalizing mathematics and verifying the correctness of software. However, they require significant background and effort to use due to the tedious nature of writing formal proofs and have not seen widespread adoption among either mathematicians or software engineers. Automated theorem provers aim to address this problem by automating the search for proofs, reducing the amount of human effort required. Recent advances in machine learning have shown that large language models can generate high-quality outputs in a variety of domains, including that of formal proofs. Most previous approaches that use language models, however, have focused on generating individual proof steps and using them in conjunction with an expensive search algorithm to find proofs. In 2023, First et al. introduced Baldur, a system that uses language models to generate entire proofs at once, instead of step-by-step, for theorems in the Isabelle proof assistant.

This thesis studies the feasibility of a similar whole-proof generation procedure for the Coq proof assistant and introduces a novel approach to automated theorem proving that recursively extracts lemmas at failure points in the proof generation process, allowing the system to break complex theorems down into simpler subproblems. We evaluate these approaches on a dataset of 724 theorems from the Software Foundations textbook and show that GPT-4 can generate whole-proofs for 66.44% of the theorems. Additionally, when augmented with our lemma extraction method, GPT-4 sees a 19.54% improvement to achieve a success rate of 79.42%, thus marginally outperforming CoqHammer—a state-of-the-art automated reasoning tool—which proves 78.73% of the theorems. We also evaluate the much smaller open-source model Phind CodeLlama, which depicts a 103.23% improvement over its baseline when utilizing lemma extraction. We release our Coq playground that contains an implementation of this procedure along with the dataset and evaluation results through an open-source repository to encourage further research in this area.

Contents

1 Introduction 1
- 1.1 Motivation . . . 3
- 1.2 Our Contributions . . . 4

2 Background 6
- 2.1 Formal Proofs . . . 7
- 2.2 The Coq Proof Assistant . . . 9
- 2.3 Large Language Models . . . 15

3 Literature Review 20
- 3.1 Hammers and SMT Solvers . . . 21
- 3.2 Neural Theorem Proving . . . 22
- 3.3 Theorem Proving Using LLMs . . . 25
- 3.4 Autoformalization . . . 31
- 3.5 Other Work . . . 36

4 Lemma Extraction 38
- 4.1 Introduction . . . 39
- 4.2 Method . . . 42
- 4.3 Implementation . . . 47
- 4.4 Evaluation . . . 49

5 Conclusion 57
- 5.1 Future Work . . . 59

Appendix A Prompts Used 61

Appendix B Examples of Proofs Found 64

References 94

Acknowledgments

This thesis would not have been possible, first and foremost, without the guidance and enthusiasm of my advisor, Nada Amin. Nada has been an incredible mentor, always giving me the freedom to explore my own ideas while being there to provide feedback and push me in the right directions when needed. Nada's speedy responses to my email queries at all hours of the day (and sometimes night!) were invaluable in keeping my research on track. I am also grateful to Stephen Chong for being my second thesis reader.

Thank you to Mark Pekala and Justin Ye for their collaboration on a class project that inspired some components of the work in this thesis. I would like to acknowledge the SEAS Computing group as well as Rakesh Nori for help with access to compute resources that were essential for running experiments. My research was also supported in part by the Office of Undergraduate Research and Fellowships through the Harvard College Research Program (HCRP) in Fall 2023, for which I am very thankful.

I'm fortunate to have had a strong support system of professors, advisors, collaborators, and friends during the course of my undergraduate education. I would specifically like to thank Professors Madhu Sudan, Anurag Anshu, and Dusty Grundmeier—who introduced me to the beautiful world of proofs early on in my freshman year—for their guidance and advice over the years, and whose classes I've had the pleasure of teaching as a course assistant or teaching fellow.

I'm immensely thankful to Ashley Zhuang for our collaboration across several courses since the beginning of college and for being a constant source of inspiration; to Michael Hwang for our frequent conversations about all things math, CS, and life more generally; and to Antara Bhattacharya for her support and encouragement during the final stages of my thesis. Thank you also to all my other friends and blockmates, in particular Harkirat Bhullar, Michelle Qin, Anurag Mitra, Shifa Hossain, Isheka Agarwal, and Prabidhik KC for all of the memories and for their unwavering support and friendship.

The only way to rectify our reasonings is to make them as tangible as those of the Mathematicians, so that we can find our error at a glance, and when there are disputes among persons, we can simply say: Let us calculate [calculemus], without further ado, to see who is right.

Gottfried Leibniz, *The Art of Discovery* (1685) [33].

1

Introduction

Calculemus. The year was 1685 when German polymath Gottfried Leibniz harbored a visionary dream: to make all reasoning as objective and unambiguous as the calculations performed by a mathematician. In his writings, Leibniz imagines a *characteristica universalis*, a hypothetical formal language that would allow expressing mathematical and sci-

entific statements, and a *calculus ratiocinator*, an algorithm that, given the statement of a theorem in this formal language along with a set of axioms, generates a sequence of arguments or proof steps that proves the theorem, where each step follows logically from the previous ones [44].

Leibniz's dream has been realized to some extent through the development of the field of formal logic in the late 19th and early 20th centuries. Formal logic studies the rules of correct reasoning and provides a symbolic framework for expressing and proving theorems. Traditionally, proofs have been written by mathematicians and computer scientists in natural language using high-level abstractions, with the emphasis being on communicating the major ideas as opposed to fleshing out every minor detail. *Formal* proofs differ from these in that the emphasis is on precision, rigor, and verifiability. As a consequence, formal proofs can be checked for correctness using a computer, usually through an *interactive theorem prover* or *proof assistant* such as Coq [51], Lean [11], Isabelle [38], or Metamath [37].

However, the process of finding these proofs continues to be an unsolved problem in artificial intelligence. While it is straightforward to verify a formal proof once it has been written in a formal language, *automated theorem proving*, the process of searching for or generating these proofs automatically, is computationally intractable. Nevertheless, there have been various attempts to automate the theorem proving process. One major class of automated theorem provers are based on satisfiability modulo theories (SMT) solvers, which use search algorithms in conjunction with heuristics and time-outs to attempt to find a proof in a reasonable amount of time. While these solvers are effective for many simple problems and are often used to automate the verification of small steps within a larger

proof, they are not well-suited for the task of finding complex proofs from scratch due to the vast search space of possible proofs.

More recently, machine learning has been used to guide the search for formal proofs. Since the advent of large language models (LLMs), in particular, there has been a surge of interest in proving theorems using these methods. While LLMs have been shown to be effective at generating some simple proofs, these approaches are still in their infancy and require further research before they can be used in practice. Language models are, after all, merely powerful next-word predictors and do not inherently have a sense of logic or reasoning. Indeed, there tend to be frequent logical errors and hallucinations even in natural language responses to prompts, so these errors are likely to be even more common in the context of formal logic.

The goal of this thesis is to first survey previous attempts and the current state of the art in automated theorem proving and then to develop an automated theorem prover based on large language models for a particular formal logical system, namely the Coq proof assistant.

1.1 Motivation

Writing formal proofs by hand is an extremely labor-intensive task. Unlike the process of writing mathematical proofs in natural language where the tedious details are assumed to be true and can be omitted, formal proofs require that every step be explicitly stated and justified. This can lead to proofs that are hundreds or even thousands of lines long, and this process is not only time-consuming but also error-prone.

There are two groups of people for whom automating parts of the formal proof writ-

ing process will be highly significant. The first is those who currently use proof assistants in their work—mathematicians trying to formalize and verify their theorems and proofs, and engineers writing highly precise and critical code for processors, compilers, or operating systems, who use rigorous mathematical proofs to verify the correctness of their implementations. For such users, an intelligent LLM-based proof generator could potentially far outperform the naive heuristic-based SMT solvers and automation tools that they use to automatically fill in portions of their proofs today.

The second group is those who *don't* currently use proof assistants due to their steep learning curve and tediousness but would benefit by their use. Software developers, for example, commonly evaluate their code at runtime against a unit test suite that is often not fully comprehensive and lets bugs go unnoticed. Advancements in automated formal verification as well as autoformalization (translating from natural language to formal proofs) could make this unit testing process unnecessary by simply proving through static analysis that their code matches the required specifications, without them having to write the technically challenging proofs themselves.

In addition, there are also broader implications of studying techniques that enable language models to write mathematical proofs. Using LLMs in conjunction with proof assistants provides them with a "logic plugin," thereby perhaps making progress towards reducing logical errors in normal textual responses, or even providing them with tools to automate ethical decisions.

1.2 Our Contributions

The contributions of this thesis are as follows:

1. A comprehensive survey of the recent history of automated theorem proving, beginning with traditional methods such as hammers and SMT solvers and moving on to more recent attempts including deep learning, guided search techniques, and various applications of large language models to the field.

2. A novel approach to automated theorem proving that makes use of lemma extraction at failure points to guide the search for proofs using a large language model.

3. An open-source Coq playground[1] written in Python that includes an implementation of this technique for the Coq proof assistant. This repository also contains tools for parsing Coq files and performing model fine-tuning and inference for Coq proofs along with evaluation data and results.

4. An evaluation of the performance of the lemma extraction approach along with comparisons to baseline methods.

The remainder of the thesis is organized as follows. Chapter 2 introduces formal logic, interactive theorem provers and the Coq proof assistant, and Transformer-based large language models, providing the necessary background for the rest of the thesis. Chapter 3 contains the literature review and contextualizes this work in the broader field of automated theorem proving. Chapter 4 details the design of our lemma extraction technique, the implementation of the Coq playground, and an evaluation of the prover's performance. Finally, Chapter 5 concludes the thesis and discusses future work.

[1] Available at `https://github.com/mtarunpr/coq-prover`.

2

Background

This chapter provides an overview of the concepts and tools that are relevant to the work presented in this thesis. We begin by discussing formal proofs and the Coq proof assistant, which our automated theorem proving approach will focus on. We then introduce large language models (LLMs) and the architecture of the transformer, which we will make

use of in our algorithm and experiments.

2.1 Formal Proofs

This thesis will primarily deal with formal proofs as opposed to informal natural language proofs. In order to study and work with formal proofs, we must first define several terms from mathematical logic.

Given an *alphabet*—a set of symbols—a *formal language* is a set of finite sequences of symbols from the alphabet. Each such sequence of symbols in the language is known as a *well-formed formula* or simply a *formula*. We will use formal languages to represent mathematical statements as well as their proofs. A *formal grammar* is a description of the formulas of a formal language through a set of production rules.

A *formal system* or a *logical system* is a formal language along with a set of axioms and rules of inference. The axioms of the system, written in the formal language, are formulas that are taken to be true. These act as the starting points for proofs, from which new formulas can be derived using the rules of inference. There are several different examples of formal systems that can be used as a foundation of logic or of mathematics. Propositional logic, for instance, is a formal system that deals with the logical connectives $\{\neg, \rightarrow, \wedge, \vee\}$ and is used to reason about the truth values of propositions. First-order logic is a more expressive collection of formal systems that allow for quantification over non-logical variables, while higher-order logic extends this to allow for quantification over functions and predicates as well. Zermelo-Fraenkel set theory (ZFC) is an axiomatic system for set theory that is most commonly used as the foundation of mathematics.

A *formal proof* is a sequence of formulas, each of which is either an axiom or follows

from previous formulas by a rule of inference. A *theorem* is the last formula in a proof.

As an example, consider a formal system that supports primitive symbols that we will denote using uppercase letters, the operator symbols $\{\neg, \to, \wedge\}$, and consists of the following set of axioms from propositional logic.

1. $P \to (Q \to P)$
2. $(P \to (Q \to R)) \to ((P \to Q) \to (P \to R))$
3. $(\neg P \to \neg Q) \to ((\neg P \to Q) \to P)$
4. $(P \wedge Q) \to P$
5. $(P \wedge Q) \to Q$
6. $(P \to Q) \to ((P \to R) \to (P \to Q \wedge R))$

Also suppose the system has only one rule of inference, *modus ponens*, which states that from P and $P \to Q$, we can infer Q. We will attempt to prove the following theorem in this formal system.

Theorem. $P \wedge (P \to Q) \to Q$

Proof.

1. $(P \wedge (P \to Q)) \to P$ (Axiom 4)
2. $(P \wedge (P \to Q)) \to (P \to Q)$ (Axiom 5)
3. $((P \wedge (P \to Q)) \to (P \to Q)) \to (((P \wedge (P \to Q)) \to P) \to ((P \wedge (P \to Q)) \to Q))$ (Axiom 2)

4. $((P \wedge (P \to Q)) \to P) \to ((P \wedge (P \to Q)) \to Q)$ (Modus Ponens on 2 and 3)

5. $(P \wedge (P \to Q)) \to Q$ (Modus Ponens on 1 and 4)

□

Informally, the proof proceeds as follows. Axioms 4 and 5 state that a conjunction implies each of its components. We can use these axioms to derive the implications 1 and 2 respectively from the hypothesis of the theorem. Axiom 2 states the distributive property of implications, and we can use this on implication 2 to derive 3. Finally, applying modus ponens twice lets us go from the implication in 3 to the desired result.

2.2 The Coq Proof Assistant

Interactive theorem provers, also known as proof assistants, are software tools that provide an implementation of a formal system and allow users to construct and then verify formal proofs by expressing them as computer programs. Coq [51] is an interactive theorem prover that was first released in 1989 and is used in both academia and industry for formal verification of software and hardware systems, as well as for formalizing mathematics and proving theorems. Notable uses of Coq include a verified proof of the four color theorem in graph theory [19] and of the Feit-Thompson theorem in group theory [18], both of which are prohibitively difficult to verify by hand due to the length and complexity of the arguments. In the software space, CompCert [34] is an example of a verified C compiler that was developed and proven correct using Coq.

Such implementations of formal proofs in the form of machine-checkable programs are made possible because of the *Curry-Howard correspondence* [22], which states that there

is an isomorphism between proofs in a formal system and programs in a model of computation. This correspondence allows us to view a proof as a program, where the formula that is proven is the *type* for the program. Coq itself can also therefore be viewed as a dependently typed functional programming language, where `M:A` can be interpreted in either of two equivalent ways: as "`M` is a proof term for proposition `A`" or as "`M` is a program of type `A`." Checking that a proof is correct then reduces to simply type-checking a program.

Proofs in Coq are commonly written using a sequence of tactics, which are commands that are used to manipulate the proof state of a proof until it is complete. The proof state is simply a list of *goals* that, if proven in their entirety, will complete the proof. Each goal consists of a *conclusion* (the proposition that needs to be proven) and a *local context* containing local definitions, variables, and hypotheses that can be used in the proof. Table 2.1 lists some common Coq tactics that are used to write proofs; depending on the tactic, it may be necessary to pass one or more arguments to it. These arguments may include terms containing constants, definitions from the local context, as well as declarations in the *global environment* that may include definitions and previously-proven theorems and lemmas in the file or in an imported file. In natural language, it is usually common to start with the hypotheses (if any) and work towards the conclusion, but in Coq, the proof is typically written backwards, starting with the conclusion and working towards what is already known to be true.

Coq uses a type theory named the *calculus of inductive constructions* [8] as its formal system, as opposed to Zermelo-Fraenkel set theory (ZFC) which is the axiomatic system most commonly used as a foundation of mathematics. The calculus of inductive constructions differs from ZFC in a few different ways. Most notably, it does not assume the law of

excluded middle as an axiom, which states that for any proposition P, either P is true or $\neg P$ is true. A consequence of this is that Coq is constructive, in the sense that it only accepts proofs of existence that can be used to construct a witness of the theorem being proved. This is in contrast to classical logic, which allows for proofs by contradiction and other non-constructive methods that demonstrate the existence of a mathematical object without actually providing an example. Having said that, Coq does allow the addition of axioms beyond those of the calculus of inductive constructions, which can be used to implement classical logic and write non-constructive proofs.

Tactic	**Description**
`intro`	Introduces a variable or hypothesis
`induction`	Begins a proof by induction
`reflexivity`	Solves a goal that is a trivial equality
`assumption`	Solves a goal that is true by assumption
`contradiction`	Solves any goal if the context contains a contradictory hypothesis
`simpl`	Simplifies the goal or a hypothesis
`destruct`	Replaces a conjunctive hypothesis with the two component hypotheses, or generates two subgoals if the hypothesis is disjunctive, each assuming that one component is true; alternatively, performs case analysis on an inductive type
`split`	Replaces a conjunctive goal with two subgoals corresponding to the components
`left` / `right`	Replaces a disjunctive goal with a goal corresponding to the left / right component
`apply`	Applies a hypothesis or lemma
`rewrite`	Substitutes one equal term for another
`unfold`	Unfolds a definition
`ring`	Solves identities that follow from commutative ring or semiring axioms

Table 2.1: Commonly used tactics in Coq.

Consider the following example of a proof in Coq. The theorem we will prove is identical to the one from § 2.1, but we will use the formal system of Coq along with Coq tactics to write the proof instead of the formal system defined earlier.

```
Variables P Q R : Prop.

Theorem and_implies : P ∧ (P → Q) → Q.
Proof.
  intro H0.
  destruct H0 as [H1 H2].
  apply H2.
  apply H1.
Qed.
```

After each tactic is applied, the proof state is updated to reflect the changes made by the tactic, as shown below. In each proof state, the local context of the goal is shown above the horizontal line, while the conclusion that needs to be proven is shown below the line. The proof is complete when there are no more goals to prove.

```
Proof.
```

```
1 goal

  ============================
  P ∧ (P → Q) → Q
```

```
intro H0.
```

```
1 goal

  H0 : P ∧ (P → Q)
  ============================
  Q
```

```
destruct H0 as [H1 H2].
```

```
1 goal

  H1 : P
  H2 : P → Q
  ============================
  Q
```

```
apply H2.
```

```
1 goal

  H1 : P
  H2 : P → Q
  ============================
  P
```

```
apply H1.
```

```
No more goals.
```

```
Qed.
```

A natural language version of this proof that mirrors the formal Coq proof is as follows. Suppose $P \wedge (P \rightarrow Q)$ is true, and call this hypothesis H_0. Since H_0, by assumption, is a true conjunction of two propositions, each of the two propositions P and $P \rightarrow Q$ must also be true. We will name these two hypotheses H_1 and H_2 respectively. Our goal is to prove that Q is true. But by H_2, Q is implied by P, so it suffices to show that P is true. But P is true by H_1; thus, we are done.

Notice that, besides at the `destruct` step where `H0` was explicitly passed as an argument, each application of a tactic transformed the goal and not the hypotheses, until the

goal was known to be trivially true through one of the assumed hypotheses. As another example, consider the following proof of the closed-form expression for the sum of the first n natural numbers, a prototypical example of a proof by induction.

```
Require Import ArithRing.

Fixpoint sum_n (n : nat) : nat :=
  match n with
  | 0 ⇒ 0
  | S n' ⇒ n + sum_n n'
  end.

Lemma sum_n_formula_helper : forall n : nat,
  2 * sum_n (S n) = 2 * (S n) + 2 * sum_n n.
Proof.
  intro n.
  simpl.
  ring.
Qed.

Theorem sum_n_formula : forall n : nat, 2 * sum_n n = n * (n + 1).
Proof.
  intro n.
  induction n as [|n' IH].
  - simpl.
    reflexivity.
  - rewrite → sum_n_formula_helper.
    rewrite → IH.
    ring.
Qed.
```

The `ArithRing` import provides a set of useful declarations related to rings and semi-rings; the `ring` tactic in particular can be used to solve goals that involve, say, only additions and multiplications of natural numbers since the set of natural numbers forms a commutative semi-ring. The `Fixpoint` keyword is used to define recursive functions and is used

here to define `sum_n`, which recursively computes the sum of the first n natural numbers. Before stating the main theorem, we first prove the lemma `sum_n_formula_helper`, which states that twice the sum of the first $n + 1$ (denoted `S n` for the successor of n) natural numbers equals $2(n + 1)$ plus twice the sum of the first n natural numbers. The proof of this lemma is straightforward and involves simplifying the goal and then using the `ring` tactic to solve it. Now our main result `sum_n_formula` states that twice the sum of the first n natural numbers equals $n(n + 1)$, and the proof proceeds by induction on n, resulting in two subgoals that can be focused using a bullet. The base case subgoal involves a trivial simplification to `0 = 0`, while the inductive step subgoal uses the `sum_n_formula_helper` lemma along with the inductive hypothesis to transform the goal into a form that is solvable using `ring`.

2.3 Large Language Models

Large language models (LLMs) are a class of generative machine learning models that are trained on large amounts of textual data and are capable of generating human-like text. These models have been shown to perform well on a variety of natural language processing tasks including text generation, question answering, summarization, translation, text classification, and code generation, and have been used in a wide range of applications. Some examples of large language models include OpenAI's Generative Pre-trained Transformer (GPT) models, Anthropic's Claude, and Google's Gemini. Some LLMs have also been release as source-available models whose weights are publicly released or available on request, including BLOOM, Meta AI's LLaMA, and some of Mistral AI's models.

2.3.1 Architecture

The transformer [55] is a neural network architecture first presented in 2017 that has seen considerable success in many large language models. The transformer architecture (see Figure 2.1) consists of an encoder and a decoder, each of which is composed of a stack of layers. Each encoder layer consists of two sublayers: a *multi-head self-attention* unit, which allows the model to weigh the importance of different tokens in a sequence when making predictions, and a feedforward neural network. The decoder layers are similar to the encoder layers, but also include an additional multi-head attention sublayer that allows the decoder to take into account the encoder's output when making predictions. The input string is first embedded into a sequence of vectors which are then passed through the encoder layers. The output of the encoder is then passed through the decoder layers, which generate the output tokens. In practice, many language models use a decoder-only variant of the transformer that does not include the encoder layers.

Before transformers, recurrent neural networks (RNNs) such as long short-term memory (LSTM) networks and gated recurrent units (GRUs) were the state of the art approaches used for processing sequences of data and natural language. RNNs process sequences sequentially by generating a hidden state h_t at each time step t as a function of the previous hidden state h_{t-1}. However, RNNs suffer from several limitations, including difficulty in capturing long-range dependencies in data, difficulty in processing long sequences of data, and difficulty in parallelizing computations. Transformers address these limitations by using a positional encoding to encode the position of tokens in a sequence instead of using recurrence, allowing them to process sequences of data in parallel instead of token-by-token, and by using an attention mechanism that allows them to capture long-range

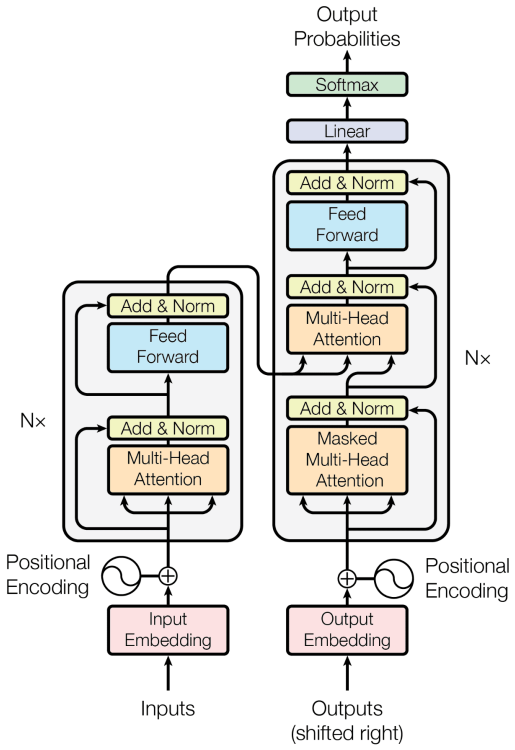


Figure 2.1: The transformer architecture. Source: [55].

dependencies in data more effectively than RNNs.

This attention mechanism works by computing an output vector as a function of a set of key-value pairs and a query vector. The output vector is a weighted sum of the values, where the weights are computed as a scaled dot product of the query vector and the key vectors followed by the application of a softmax function to the result. In other words,

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where Q, K, and V are matrices containing the query, key, and value vectors, respectively, and d_k is the dimension of the key and query vectors. These three matrices are computed

from the input sequence using learned linear transformations. If I is the matrix of input vectors, then $Q = IW_Q$, $K = IW_K$, and $V = IW_V$, where W_Q, W_K, and W_V are parameter matrices that are learned during training. In *multi-headed* attention, the query, key, and value vectors are split into h groups, and the attention mechanism is applied to each group separately. The results are then concatenated and passed through a linear transformation to produce the final attention output.

2.3.2 Training

Large language models are typically trained on massive corpuses of textual data using self-supervised or semi-supervised learning techniques. In self-supervised learning, the data is entirely unlabeled and the model is simply trained to predict the next token in a sequence given the previous tokens. In semi-supervised learning, the model is trained on a combination of labeled and unlabeled data. For instance, OpenAI used a semi-supervised approach consisting of two steps to perform training [43]. The first is a "pre-training" stage on a diverse dataset of unlabeled text, and the second is a supervised discriminative "fine-tuning" stage that adapts the weights learned to a specific task. GPT-3 was trained on a filtered version of a web archive dataset known as Common Crawl, which consists of petabytes of data crawled from billions of pages on the web.

Once trained, a model can be further fine-tuned on a smaller dataset for a specific task, such as text generation in a specific format, question answering on a particular topic, or code completion. Codex [6] and CodeLlama [45] are models that have been fine-tuned for code generation tasks. Another method that has been used successfully to fine-tune LLMs is a technique known as reinforcement learning with human feedback (RLHF) [63],

using a reinforcement learning algorithm called Promixal Policy Optimization (PPO) [47]. This has been used to great success in conversational models like ChatGPT, Claude, and Gemini.

Besides fine-tuning—a method that requires large-scale datasets—other approaches to priming a model for a specific task and improving performance involve prompt engineering, i.e., designing prompts that guide the model towards the desired output. This is possible due to the ability of LLMs to perform in-context learning and only requires a minimal amount of task-specific data. One such approach that has been successful at several tasks is few-shot learning [5], in which the model is given a small number of examples of the task it is to perform as part of the prompt during inference, and then asked to perform the task on new examples.

The reasoning capabilities of language models is a challenging area of research; LLMs have shown to have dismal performance on reasoning benchmarks [54]. Chain-of-thought prompting is another prompting technique that aims to improve the reasoning capabilities of LLMs by eliciting a chain of reasoning steps from the model, each of which is conditioned on the previous step. It remains to be seen to what extent language models can be made to reason effectively [24], whether in the context of mathematical proofs or in other domains.

3

Literature Review

This chapter provides a survey of the literature on automated theorem proving and traces the evolution of the field, starting with hammers and traditional automated theorem provers but focusing on the recent developments in neural theorem proving, both with and without the use of large language models. We also discuss related work on autoformaliza-

tion and other aspects of mathematical reasoning.

3.1 Hammers and SMT Solvers

A satisfiability modulo theories (SMT) instance is a generalization of a boolean formula where boolean variables are replaced by predicates from a set of "theories," which define the predicate symbols. SMT solvers are tools that attempt to determine the satisfiability of SMT instances. In the context of automated theorem proving, SMT solvers and related tools work by converting the problem of finding a proof of a theorem (say in first-order logic) to that of finding a satisfying assignment to a formula. This formula is constructed in conjunctive normal form from the premises of the theorem and the negation of the goal, and heuristics are used to search for a satisfying assignment. Some examples of SMT solvers include Z3 [10] and CVC4 [3], while tools like Vampire [31] and Eprover [48] are closely related first-order automated theorem provers.

Hammers are powerful tools built for specific proof assistants that perform automated reasoning using a combination of external automated theorem provers and SMT solvers along with facts from large libraries of formal proofs. However, hammers are not always successful in proving theorems; CoqHammer [9]—a state-of-the-art hammer plugin for Coq released in 2018—for instance, is unable to prove any theorem that requires induction. Additionally, the proofs generated by hammers are often long and contrived, making them difficult to understand and debug. That said, hammers are still widely used in practice due to their ability to prove many theorems that are tedious to prove by hand: Sledgehammer [39] is a popular hammer that is built into the Isabelle proof assistant.

In addition to hammers, proof assistants also come with a few simpler automation

tactics built into the system. In Coq, the `auto` tactic solves goals that are solvable by any combination of `intros`, `reflexivity`, `assumption`, and `apply`, where `apply` is used on the hypotheses in the local context. The `intuition` tactic is a more powerful version of `auto` that uses a search tree built by a decision procedure for constructive propositional calculus to generate a set of subgoals equivalent to the original goal. It then applies `auto` to these subgoals.

3.2 Neural Theorem Proving

Before we delve into the use of large language models for theorem proving, we first discuss the early attempts at using neural networks for theorem proving that predate the advent of LLMs.

3.2.1 GamePad

In 2018, Huang et al. introduced GamePad [23], a learning environment for theorem proving in Coq. While there are works predating GamePad that use search techniques like the A* search algorithm to search for formal proofs [16] or even deep learning techniques for tasks like premise selection [25] or proof-step usefulness classification [29], GamePad was the first to use neural networks to synthesize proofs in Coq.

The proposed method involves two components:

1. Position evaluation, which involves estimating the ease of proving a given proof state, measured in terms of the number of proof steps required to complete a proof.

2. Tactic prediction, which predicts the next tactic to apply to a proof state. This may also require the synthesis of one or more arguments.

GamePad uses recurrent neural networks to embed proof states and implements proof synthesis by combining the tactic prediction model with verifier interaction. The authors explore two applications: to a simple algebraic rewrite problem and to a real-world dataset containing 1602 lemmas from a Coq formalization of the Feit-Thompson theorem [19]. For the latter task, the authors compare the performance of a GRU and a TreeLSTM [50] with that of an SVM baseline trained on a set of heuristic features. Using kernel-level proof states, both RNNs outperformed the SVM in terms of accuracy on both position evaluation and tactic prediction.

Notably, the methods used in GamePad are somewhat rudimentary: the system does not, for instance, attempt to synthesize arguments for the `have` tactic which introduces local lemmas, despite the tactic being highly prevalent in the Feit-Thompson dataset. GamePad also does not allow for any backtracking. Regardless, the results showcase a very promising first step towards the use of machine learning in interactive theorem proving.

3.2.2 CoqGym/ASTactic

In 2019, Yang and Deng [60] published a paper that made two major contributions to neural theorem proving. The first was CoqGym, a dataset and learning environment for theorem proving in Coq consisting of 71K human-written proofs sourced from 123 Coq projects spanning various fields of mathematics such as number theory, group theory, and linear algebra, the correctness of algorithms such as RSA and Huffman coding, as well as verified software projects such as CompCert. This was much larger than the dataset provided by GamePad and became a standard benchmark used by Coq provers.

The second contribution was ASTactic, a neural theorem prover that proves theorems

by interacting with the Coq proof assistant. ASTactic generates tactics one step at a time in the form of abstract syntax trees (ASTs) and synthesizes arguments to these tactics using the available premises (i.e. previously-defined definitions and theorems). Its architecture consists of an encoder-decoder model; the encoder is a TreeLSTM [50] that embeds all input Coq terms including the goal and the premises as ASTs into vectors, while the decoder is a GRU with an attention module that generates a program-structured tactic by sequentially growing an AST. During testing, multiple possible tactics are sampled from the model and DFS is used to search for a valid proof.

ASTactic achieves a success rate of 12.2% on the test subset of the CoqGym dataset, which is a significant improvement over the 4.9% success rate of the built-in automation tactics in Coq like `auto` and `easy`. However, the `hammer` tactic provided by CoqHammer outperforms ASTactic with a success rate of 17.8% with a time limit of just 20 seconds and 24.8% with an extended time limit of 10 minutes. Nevertheless, this was an encouraging result given that ASTactic was trained from scratch and did not use any additional domain knowledge or heuristics. ASTactic when combined with `hammer` was able to prove 30.0% of the theorems, indicating that the model was learning to prove theorems that automation could not do on its own.

3.2.3 Data-driven Lemma Synthesis

While somewhat orthogonal to other papers discussed in this section, the work by Sivaraman et al. in 2022 [49] is nevertheless very relevant to theorem proving. Instead of focusing on the proof generation problem, the authors propose a method for synthesizing lemmas that may be useful in proving a given theorem, and then evaluate their approach with the

help of the existing Proverbot9001 [46] that uses neural methods for theorem proving. The tool is released as a Coq plugin that enables the `lfind` tactic.

Specifically, this work treats lemma synthesis as a data-driven program synthesis problem, where examples generated from the current proof state serve as the data. Given a goal, `lfind` first generalizes it, then performs synthesis, filtering, and ranking of candidate lemmas. The filtering is done using a counterexample search, while the ranking is done through the automated prover Proverbot9001. A lemma is considered successful if the automated prover is able to prove the theorem using that lemma and the lemma itself is also automatically proven. In their evaluation, the authors find that useful lemmas are produced in 68% of cases where a human prover used a lemma to make progress.

3.3 Theorem Proving Using LLMs

Next, we trace the development of LLM-based provers. One common theme across several works in neural theorem proving predating the use of large language models is the difficulty of generating original mathematical terms to use as arguments to tactics. Attempts at circumventing this problem typically involved severely limiting the infinitely large space of possible terms to some finite subset of literals, theorems, and/or definitions. Large language models pose a potential solution to this problem through the generation of new terms as a sequence of tokens, much like the generation of natural language text that was not seen during training and is not merely a subset of the input. This includes complex terms that are functions of declarations in the global environment as well as declarations from libraries and other imported files, and literals.

3.3.1 GPT-f

GPT-f [42] was an early investigation into the use of large language models for theorem proving by OpenAI researchers Polu and Sutskever in 2020 for the Metamath proof assistant. Unlike other formal systems like Coq and Lean, Metamath has only a single type of tactic based on substitution, and the goal can be inferred from the tactic itself. This makes it a simpler and more convenient choice of proof assistant for use with neural networks.

The authors use a decoder-only Transformer model with architecture similar to GPT-2 and GPT-3; the largest trained model has 36 layers and 774 million parameters. This is first pre-trained on a general textual corpus and then fine-tuned on a Metamath dataset based on the `set.mm` library, which contains proofs of 37K theorems in ZFC. To study the effect of pre-training, the authors compare results on different pre-training datasets: (1) CommonCrawl, (2) CommonCrawl + GitHub, and (3) CommonCrawl + WebMath, where WebMath includes GitHub, mathematics articles from arXiv, as well as Math StackExchange. Proofs are found by running a proof search, which maintains a proof tree and a queue of open goals sorted by the model's estimated probability of success, measured as the sum of the log probabilities of the tactics used to reach the goal. In the "learned value function" variation, a value function is iteratively trained to guide the proof search instead of log probabilities.

Evaluated on a validation set of about 1k theorems, the model pre-trained on CommonCrawl + WebMath achieved the best performance in the regular variation, proving 42.56% of the theorems. In the learned value function variation, the best set of hyperparameters led to a success rate of 56.5%. Perhaps even more impressively, the authors were able to use GPT-f to find 23 proofs of theorems in `set.mm` that were shorter than

the human-written ones and were accepted into the Metamath library. This work demonstrated the potential of large language models for theorem proving and was followed quickly by several other works that built on this foundation.

3.3.2 HTPS: HyperTree Proof Search

Lample et al. in 2022 introduced HyperTree Proof Search (HTPS) [32], a proof search algorithm inspired by the AlphaZero learning algorithm, which uses deep learning in conjunction with Monte Carlo Tree Search to learn to play the two-player games of Go, chess, and shogi. Theorem proving differs from these games in three significant ways: (1) each move in a game is chosen from among a small, finite set of possibilities, whereas proof search has an infinitely large action space; (2) it suffices to only consider the most likely moves of the opponent in games, but every subgoal needs to be correctly proven in theorem proving; and (3) it is often useful to consider suboptimal moves in two-player games since they do not necessarily lose the game and can provide information, whereas if a subgoal generated deep in a search tree is unprovable, the entire branch adds no useful information.

The authors detail the architecture of HTPS which aims to overcome these challenges through the use of a policy model (from which tactics are sampled) and a critic model (which estimates the provability of a goal), both of which are encoder-decoder transformer models that guide the search algorithm by reducing the search to the most promising branches. This search happens on a *hypergraph*, where the theorem to be be proven is the root node, tactics are the edges, and subgoals are the child nodes. A successful proof corresponds to a hypertree from the root to leaves that are empty sets.

HTPS was implemented for Metamath, Lean, as well as a new proof environment

named "Equations" introduced in this work. The models were trained on 37K proofs from the `set.mm` library for Metamath, 24K theorems from `mathlib` for Lean, and on supervised data generated on the fly for Equations. Their results show that the resulting model, named Evariste, proves 65.4% of the tested Metamath theorems, outperforming the previous state-of-the-art of 56.5% by GPT-f (see § 3.3.1). Similarly, the pass rate for Lean using 64 trials is 42%.

3.3.3 Thor

Thor [27] is a framework released in 2022 that serves to integrate language models with hammers. This is done by using the hammers for premise selection—a challenging task due to the vast space of possible premises—while designating all other tasks to language models. The authors pre-train a 700M-parameter decoder-only transformer on GitHub and arXiv and then fine-tune on the PISA dataset [26]. The model is trained to predict a special `<hammer>` token to indicate when to use a hammer.

The language model on its own is able to prove 39.0% of the theorems in the test subset of PISA, while Sledgehammer on its own proves 25.7%, and the union of the theorems proven by both tools is 48.8%. Thor outperforms both tools individually as well as the union, achieving a success rate of 57.0%.

3.3.4 Diva

Diva is a diversity-driven automated theorem proving approach for Coq introduced by First et al. in 2022 [13]. Similar to an approach used by Thor (see § 3.3.3), the authors propose a method that combines the strengths of multiple hammer-based and neural meth-

ods, thus increasing diversity in the models used. Specifically, Diva combines TacTok [14], a semantics-aware proof synthesis tool, with ASTactic [60] and is able to prove 21.7% of the theorems of the CoqGym benchmark, which is significantly more than either of those tools alone. Although this is outperformed by CoqHammer which proves 26.6%, Diva is able to prove 33.8% of the theorems when additionally combined with CoqHammer, a notable improvement over CoqHammer alone.

3.3.5 Baldur

In a significant departure from previous attempts at using search techniques that incorporate language models as only one component of the larger search algorithm, Baldur [15] attempts to directly generate whole proofs at once instead of one step at a time for the Isabelle/HOL theorem prover. In this work published in 2023, First et al. study the feasibility of whole-proof generation using a transformer-based model, comparing it to previous more expensive search-based methods. While Baldur is designed to work with any language model, the authors use Minerva [35] to run their experiments, which is pretrained on a mathematics corpus.

Baldur consists of two models:

1. A *proof generation* model that generates an entire proof given only the theorem statement. This model is fine-tuned on proofs sourced from the training subset of the PISA dataset. During inference, the generated proof is checked for correctness by Isabelle. If the proof is correct, it is accepted; otherwise, another proof attempt is sampled.

2. A *proof repair* model that takes as input the theorem statement, an incorrect proof

sampled from the proof generation model, and the error message returned by Isabelle when checking the proof. This model is fine-tuned on a proof repair training set created by sampling proofs for all theorems in the training set and retaining all failed proofs and corresonding error messages. The human-written proof of each such theorem is used as the target sequence. During inference, the model then generates a new proof attempt that aims to correct the error in the original sample.

Evaluated on a 6,336 theorems from the test subset of PISA, the authors find that Baldur even without the repair model is able to generate correct proofs for 47.9% of the theorems, compared to Thor's 39.0% (see § 3.3.3). With the repair model, Baldur proves an additional 1.5% of the theorems. Most notably, the union of the theorems correctly proven by Baldur and those proven by a vartion of Thor that combines a learned model, search, and a hammer, covers 65.7% of the tested theorems, which is a significant improvement over the 57.0% success rate of this Thor variant alone. This demonstrates that whole-proof generation and search techniques could complement each other to improve the overall success of theorem proving.

3.3.6 LeanDojo/ReProver

LeanDojo [61] is a Lean playground that contains toolkits, data, and benchmarks for theorem proving in Lean. Unlike previous LLM-based provers, LeanDojo is completely open-source. In addition to the playground, the authors also present ReProver, a retrieval-augmented prover that fetches relevant premises from Lean's library to assist the model in proving theorems. This is similar to Thor (see § 3.3.3) in that premise selection is used, but differs in the use of retrieval instead of hammers. This retrieval mechanism is based on Dense Passage

Retrieval (DPR) [30]: it uses vector embeddings for both the current proof state and the list of premises to identify premises most similar to the state. The theorem proving itself happens step-by-step: given a proof state, the model generates one tactic to apply that may use one or more of the selected premises.

The authors use the `google/byt5-small` model checkpoint as their base model. ReProver is then trained for a week and then evaluated on subsets of the LeanDojo benchmark consisting of 98,734 theorems from Lean's math library. Notably, it outperforms both the `tidy` automation tactic as well as GPT-4, achieving a success rate of 51.2% with retrieval and 47.6% without.

3.4 Autoformalization

A distinct but related problem to automated theorem proving is that of autoformalization, which involves translating natural language (or "informal") proofs of theorems into formal proofs. This is a challenging problem due to the ambiguity and imprecision of natural language, as well as the difficulty of mapping the high-level assertions that are often seen in a natural language proof to the highly detailed and precise proof steps necessary in a formal proof. Additionally, since the formal system used by a proof assistant varies from one to another, some proofs in ZFC may not directly translate to equivalent proofs in Coq, for instance. Attempts at autoformalization typically do not involve training a model for this specific task due to the lack of availability and difficulty of construction of datasets that contain both informal proofs and a corresponding formal proof that maintains the same argument structure.

3.4.1 First Experiments with Neural Translation

The first attempt at autoformalization using neural networks was made by Wang et al. in 2018 [56], who attempted to translate informal LaTeXversions of Mizar texts into the formal language of the Mizar proof assistant. The informalized dataset was constructed using a Mizar-to-LaTeXtranslation tool developed by Bancerek [2] for publishing Mizar articles in the journal Formal Mathematics and contains one million pairs of Mizar formulas and their corresponding LaTeXsentences. This dataset was then used to train a neural machine translation (NMT) implementation of the sequence-to-sequence (seq2seq) model consisting of an encoder and a decoder, both having multiple layers of recurrent neural networks along with the attention mechanism.

The authors found that their best NMT model was able to correctly translate as much as 65.73% of their test set correctly to Mizar, and the union of all 39 models that they tested produced correct translations of 79.17% of the same test set. However, it is important to note that the work used a synthetic dataset where the natural language sentences were generated from Mizar formulas and hence likely mirrored the structure of the formal statements very closely, unlike human-written mathematical texts. Additionally, the authors did not attempt to generate entire formal proofs from the their informal versions, which is the ultimate goal of autoformalization. That said, this work was the first to demonstrate the feasibility of using neural networks for autoformalization and laid the groundwork for future research in this area even before the advent of Transformer-based language models.

3.4.2 Autoformalization with Large Language Models

An early study of the applicability of Transformer-based large language models to the task of autoformalization was conducted by Wu et al. in 2022 [59]. Unlike Wang et al. who relied on synthetic data to train their models, the authors of this paper aimed to test the performance of models trained via self-supervised language modeling on a vast corpus of data, but not specifically for the task of autoformalization. In particular, they tested the PaLM [7] and Codex models [6] without any additional fine-tuning on the task of translating informal problem statements from high school math competitions into formal theorem statements in Isabelle. The primary, perhaps surprising, result was that the base Codex model was able to achieve a success rate of 25.3% on the 150 problems in the MATH dataset [21].

The authors also study the use of *expert iteration* in theorem proving to demonstrate the usefulness of the autoformalization of theorem statements. Expert iteration is a process that, given a base version of a neural theorem prover, iteratively generates a better dataset and uses this new dataset to fine-tune and improve the theorem prover further. However, additional formalized theorem statements beyond the original dataset used to train the base prover are needed to begin this process. Traditionally, this additional formalization is done by a human; this work shows that the theorem statements autoformalized by a language model are sufficient to successfully perform expert iteration, thus bypassing the tedious process of writing formal theorem statements by hand. After two iterations of expert iteration on the Thor theorem prover, the authors were able to improve its success rate on the test subset of the MiniF2F dataset [62], consisting of 488 problems from high-school mathematical competitions, from 29.9% to 35.2%.

3.4.3 ProofNet

ProofNet is a benchmark for the autoformalization of undergraduate-level mathematical proofs for the Lean proof assistant introduced by Azerbayev et al. in 2023 [1]. While existing benchmarks such as MiniF2F focused on formal theorem statements, ProofNet contains not only theorem statements in Lean, but also corresponding informal theorem statements and proofs. As a result, the dataset supports not only the evaluation of formal theorem proving and autoformalization/informalization of theorem statements, but also informal theorem proving and the autoformalization of entire proofs. The dataset consists of 371 problems sourced from the exercises of undergraduate-level mathematics textbooks.

The authors also report baseline results on statement autoformalization using OpenAI's code-davinci-002 Codex model as well as ProofGPT, a much smaller Pythia [4] model fine-tuned on mathematical texts available online. Using in-context learning, ProofGPT was unable to correctly formalize any of the 371 problems of ProofNet, likely due to its smaller parameter count, although the Codex model achieved a success rate of 13.4%.

Additionally, the authors introduce two new statement autoformalization methods:

1. Prompt retrieval. Since the performance of in-context learning is highly sensitive to the exact prompt used, the authors argue that it would be beneficial to selectively choose the k most-relevant few-shot examples sourced from a knowledge-base to include in the prompt, instead of using a fixed set of examples. The authors do this by first generating an autoformalized theorem statement using the standard in-context learning procedure, then producing vector embeddings for the 90K statements in Lean's `mathlib` as well as the formalized statement, and finally retrieving its k-nearest

neighbors. This is similar to the retrieval-augmented generation used by ReProver (see § 3.3.6), although the retrieval there is used for premise selection rather than few-shot example selection.

2. Distilled backtranslation. This is a method inspired by previous work on unsupervised translation from one natural language to another [20] and serves to overcome the limitation of the lack of parallel informal-formal data for autoformalization. Here, informal mathematics serves as the source language and Lean serves as the target language. The authors fine-tune their ProofGPT model (the "student" model) with the help of Codex (the "teacher" model) by first manually constructing a few-shot prompt with a few informal-formal pairs from `mathlib`, then sampling synthetic backtranslations from Codex on all of `mathlib`, and finally training ProofGPT on the synthetic pairs.

Prompt retrieval improves the 13.4% success rate of Codex to 16.1%, while distilled backtranslation improves the 0% accuracy of ProofGPT to 3.2%.

3.4.4 DSP: Draft, Sketch, and Prove

One very recent approach to autoformalization is the Draft, Sketch, and Prove (DSP) system introduced in 2023 by Jiang et al. [28]. Unlike previous evaluations of large language models at formalizing theorem statements or individual sentences, DSP proposes a method to formalize entire proofs of theorems. Given a natural language statement of a theorem, DSP performs three steps:

1. Draft: Generate a draft of the proof in natural language. This can be done either by

a large language model (if the task is proof generation) or by a human (if the task is autoformalization).

2. Sketch: Convert the natural language proof draft into a formal proof *sketch* using a large language model. This sketch is a high-level skeleton of the proof that follows the structure of the informal draft but is written in the formal language of a proof assistant, with the assertions or conjectures left unproven.

3. Prove: Fill in the gaps in the proof sketch using an off-the-shelf automated theorem prover like Z3 or other automation built into proof assistants to obtain a complete formal proof.

The authors provide an implementation of DSP for the Isabelle proof assistant and evaluate it on the MiniF2F dataset augmented with informal proofs either retrieved from an official source or written manually by hand. They show that their approach is able to correctly autoformalize human-written informal proof drafts for 39.3% of the problems in the test subset of MiniF2F, compared to the 20.9% success rate of Sledgehammer with heuristics on the formal theorem statements in the same dataset.

3.5 Other Work

Besides the major works discussed above, there have been several other attempts at using neural networks for theorem proving. TacticToe [17] uses an MCTS algorithm for HOL4. TacticZero [58] frames the HOL4 proof search problem as a Markov decision process and uses deep reinforcement learning. Proverbot9001 [46] uses a combination of feed forward neural networks and recurrent neural networks for theorem proving in Coq. StepCoder

[12] deals with code generation instead of theorem proving, but uses reinforcement learning from compiler feedback, a method that can be adapted for proof assistants.

Additionally, there has also been much simultaneous interest in the use of large language models for reasoning outside of formal proofs. For instance, some recent work has focused on natural language proof attempts and solving word problems in mathematics [57, 36], while others have focused on Olympiad geometry problems [53].

4

Lemma Extraction

This chapter presents a novel approach to automated theorem proving that aims to break theorems down into smaller lemmas to guide proof generation using a large language model. We first describe the motivation behind our approach and then detail the algorithm, an implementation for Coq, and an evaluation of its performance on a dataset

of 724 theorems.

4.1 Introduction

Large language models have had much success in recent years at understanding and generating code in various programming languages [6]. The formal languages used by interactive theorem provers, however, differ from these in a few different ways, not least of which is the limited availability of human-written programs in these languages due to the significantly smaller communities and the tediousness of writing formal proofs by hand. The Archive of Formal Proofs[1], for instance, is only 300MB in size as of March 2024, and the CoqGym dataset [60] is even smaller, with the raw source files measuring 34MB. In comparison, OpenAI's Codex was trained on 159GB of Python code from 54 million GitHub repositories [6]. This lack of data makes it difficult to achieve significantly improved performance on formal logic tasks through only fine-tuning.

Nevertheless, LLMs have shown promise in theorem proving and there has been much research in the last few years on other ways of leveraging them for this purpose, beyond training and fine-tuning. As discussed in Chapter 3, recent attemps at using LLMs for automated theorem proving have used one of two approaches: either a step-by-step proof generation process that generates one proof step at a time to grow a search tree that is used in conjunction with an expensive search algorithm to find a complete proof, or a more direct approach that generates the entire proof at once. We will focus on the latter approach, which was used in Baldur [15] (see § 3.3.5), a recent work that will serve as the basis for our own investigations.

[1] https://www.isa-afp.org

The authors of Baldur focus on the Isabelle proof assistant, where proofs can be written using a declarative proof language called Isar that includes the intermediate proof states within the proof itself, hence allowing it to be human-readable. This makes it more amenable for language models to generate proofs in a manner akin to chain-of-thought (see § 2.3.2) as the output of the model includes the proof states at each step, allowing the transformer to consider these when generating the next step. In contrast, Coq uses a procedural proof language that is not human-readable and requires the user to step through the series of tactics in the proof one by one using the proof assistant's verifier to view the intermediate proof states and decide what tactic to use next. This makes whole-proof generation in Coq a possibly more challenging task for language models and motivates our study of their effectiveness in this setting.

Given the likely difficulty of generating whole proofs in Coq, we also propose a method of breaking theorems down into smaller lemmas with the aim of improving the chances of finding a correct proof. This is motivated by our hypothesis that large language models will be able to reason about smaller, simpler theorems with fewer proof steps more accurately than complex theorems with highly involved proofs. Now how do we decide what lemmas to extract for a given theorem? Previous works adjacent to ours have used data-driven lemma synthesis or large language models to generate proof sketches as a series of assertions or "local lemmas" based on an informal proof draft. See §§ 3.2.3 and 3.4.4 for a discussion of these methods.

In this work, we propose simply generating a whole-proof, attempting to verify its correctness, and then extracting a lemma that would complete the goal at the point of the first error encountered during the verification. This technique is motivated by our

```
theorem distribute:
  "A ∧ (B ∨ C) ⟹ (A ∧ B) ∨ (A ∧ C)"
proof -
  assume "A ∧ (B ∨ C)"
  then have A: "A" and BC: "B ∨ C"
    by auto
  {
    assume "B"
    with A have "A ∧ B" by simp
    then have "(A ∧ B) ∨ (A ∧ C)"
      by simp
  }
  moreover
  {
    assume "C"
    with A have "A ∧ C" by simp
    then have "(A ∧ B) ∨ (A ∧ C)"
      by simp
  }
  ultimately show "(A ∧ B) ∨ (A ∧ C)"
    using BC by blast
qed
```

```
Variables A B C : Prop.

Theorem distribute : A ∧ (B ∨ C)
  → (A ∧ B) ∨ (A ∧ C).
Proof.
  intro H.
  destruct H as [H1 H2].
  destruct H2 as [H2 | H3].
  - left. split.
    + exact H1.
    + exact H2.
  - right. split.
    + exact H1.
    + exact H3.
Qed.
```

Figure 4.1: Comparison of proof languages in Isabelle (left) and Coq (right) for a simple theorem from propositional logic. Proofs written using Isar in Isabelle include the intermediate proof states within the proof itself, for instance in the `have` and the `assume` steps, making them more human-readable albeit more verbose. In contrast, Coq proofs only specify *how* to transform the proof state, not what the goal or the hypotheses themselves are, requiring the assistance of the verifier to determine proof states line by line.

hypothesis that LLMs are likely to make mistakes in the fine-grained details of a proof—hallucinations of identifiers or inapplicable tactics, for instance—when generating whole proofs, but are more likely to get the general proof structure right. To the best of our knowledge, this is the first work to explore this method.

To summarize, we ask the following research questions:

1. How effective are large language models at generating whole proofs of theorems in

Coq?

2. Can their performance on the whole-proof generation task be improved by extracting lemmas at failure points during proof generation and recursively generating whole proofs for these lemmas?

4.2 Method

In this section, we will describe the algorithm we use to extract lemmas at failure points to guide the proof generation process. While this approach is general and can be used with other interactive theorem provers too, we will focus on Coq in this work.

4.2.1 Lemma Extraction Algorithm

Given the statement of a theorem and a global environment, our proposed method of proof search proceeds as follows.

1. Generate a whole-proof attempt of the theorem using a large language model. In addition to the theorem statement itself, the model is also given access to the statements of the definitions and theorems in the global environment as part of the prompt to allow it to use these in its proof. We do not include proofs of these definitions and theorems to avoid exceeding the context window size of the model.

2. Attempt to verify the proof using the Coq verifier. If the proof is correct, we are done. If an error is encountered, proceed to the next step.

3. Using an annotated version of the sentences in the proof attempt, identify the index i_{err} of the first sentence that caused the error.

4. Extract the first goal of the proof state at the sentence with index $i_{err} - 1$.

5. Construct a lemma that uses the variables and hypotheses from the extracted goal as premises and the goal's conclusion as its conclusion.

6. Recursively generate a whole-proof attempt for the lemma with the same global environment as the original theorem, recursing until either a successful proof of the lemma is found or a maximum depth is reached.

7. Replace the portion of the original theorem's proof that attempted to prove the extracted goal with an application of the lemma (i.e. using the `apply` tactic). Also modify the global environment of the theorem to include the lemma.

8. Repeat the process for the next error encountered, if any.

4.2.2 An Example

To illustrate the lemma extraction algorithm, consider the following example based on the induction proof in Coq discussed in § 2.2. As a reminder, this is a proof of the theorem `sum_n_formula` that states that twice the sum of the first n natural numbers is equal to $n(n + 1)$. Suppose we have previous proven the lemma `sum_n_formula_helper`, so the global environment for the theorem includes this lemma in addition to the definition `sum_n` that recursively computes the sum of the first n natural numbers.

Using the statement of `sum_n_formula` and the global environment, suppose a large language model generates the following proof attempt.

```
1 Require Import ArithRing.
2
```

```
Fixpoint sum_n (n : nat) : nat :=
  match n with
  | 0 ⇒ 0
  | S n' ⇒ n + sum_n n'
  end.

Lemma sum_n_formula_helper : forall n : nat,
      2 * sum_n (S n) = 2 * (S n) + 2 * sum_n n.
Proof.
  intro n.
  simpl.
  ring.
Qed.

Theorem sum_n_formula : forall n : nat, 2 * sum_n n = n * (n + 1).
Proof.
  induction n as [| n' IHn'].
  - simpl.
    reflexivity.
  - ring.
Qed.
```

This proof is valid up to the base case. At the beginning of the inductive step, which is focused using the bullet, the proof state looks as follows.

```
  -
1 goal
  n' : nat
  IHn' : 2 * sum_n n' = n' * (n' + 1)

  ============================
  2 * sum_n (S n') = S n' * (S n' + 1)
```

The next proof step as output by the model is the `ring` tactic, which produces an error.

```
Tactic failure: not a valid ring equation.
```

Indeed, the `ring` tactic is not applicable here as the goal's conclusion depends on the `sum_n` function and is not simply derivable from the axioms of a ring or semi-ring. The inductive hypothesis is needed to complete the proof, and the `sum_n_formula_helper` lemma is also useful. We extract the hypotheses and conclusion of the goal at the point just before the error (shown above), and use them construct the following lemma.

```
Lemma extracted_lemma : forall n' : nat,
  forall IHn' : 2 * sum_n n' = n' * (n' + 1),
  2 * sum_n (S n') = S n' * (S n' + 1).
```

Note that this lemma can also be written—as is more commonly done in human-written propositions—without the use of an explicit quantified variable for the inductive hypothesis:

```
Lemma extracted_lemma' : forall n' : nat,
  2 * sum_n n' = n' * (n' + 1) →
  2 * sum_n (S n') = S n' * (S n' + 1).
```

However, we use the former form in our implementation for ease of construction—since there is no need to differentiate between variables and hypotheses—and in order to allow variables and hypotheses to be passed as arguments to the lemma when applying it in the proof of the theorem. Now we recursively generate a proof attempt for `extracted_lemma` using the same global environment as the original theorem. Suppose the model generates the following correct proof.

```
Lemma extracted_lemma : forall n' : nat,
  forall IHn' : 2 * sum_n n' = n' * (n' + 1),
  2 * sum_n (S n') = S n' * (S n' + 1).
```

```
Proof.
  intro n'.
  intro IHn'.
  rewrite sum_n_formula_helper.
  rewrite IHn'.
  ring.
Qed.
```

Then, it suffices to inject this new lemma along with its proof into the proof script before the definition of `sum_n_formula`, and then replace the portion of the original proof within the inductive step with a single `apply` tactic taking the lemma as an argument. This completes the proof of `sum_n_formula`, and the resulting proof script is as shown below.

```
Require Import ArithRing.

Fixpoint sum_n (n : nat) : nat :=
  match n with
  | 0 ⇒ 0
  | S n' ⇒ n + sum_n n'
  end.

Lemma sum_n_formula_helper : forall n : nat,
      2 * sum_n (S n) = 2 * (S n) + 2 * sum_n n.
Proof.
  intro n.
  simpl.
  ring.
Qed.

Lemma extracted_lemma : forall n' : nat,
  forall IHn' : 2 * sum_n n' = n' * (n' + 1),
  2 * sum_n (S n') = S n' * (S n' + 1).
Proof.
  intro n'.
  intro IHn'.
  rewrite sum_n_formula_helper.
```

```
  rewrite IHn'.
  ring.
Qed.

Theorem sum_n_formula : forall n : nat, 2 * sum_n n = n * (n + 1).
Proof.
  induction n as [| n' IHn'].
  - simpl.
    reflexivity.
  - apply (@extracted_lemma n' IHn').
Qed.
```

In this example, the model correctly identifies that the original theorem requires a proof by induction and also correctly proves the simple base case. The proof of the inductive step, however, results in an error in the form of a failed tactic. Extracting a lemma for this inductive step allows the proof state (i.e. the subgoal containing the hypotheses and conclusion) to be passed to the model as a "reminder" of what needs to be proven, allowing it to focus on the details of that subgoal while having access to the local context. This is in contrast to the original whole-proof attempt, where the model had to generate tactics with no guidance as to what remained to be proven after each step.

4.3 Implementation

Our Coq playground is available as an open-source repository on GitHub[2]. The implementation is written in Python and uses Alectryon [41], a Python interface for Coq, to interact with the verifier. The codebase is organized into several source files, including ones that handle parsing, model inference, and theorem proving using lemma extraction. It also

[2] https://github.com/mtarunpr/coq-prover

contains the Coq source files from which the dataset used for evaluation is constructed (see § 4.4.2) as well as the generated proof scripts and logs.

The main components of the playground are as follows.

- `coq.py`: Contains definitions of classes `Theorem` and `Definition` that represent theorems (including variants such as lemmas, examples, and propositions) and definitions in Coq respectively. Each theorem has a statement and a proof, as well as a list of other theorems and definitions that exist in its global environment.

- `coq_parser.py`: Contains a simplified parser for Coq files that extracts theorems and definitions from a specified Coq project directory and creates instances of the `Theorem` and `Definition` classes. Also contains functions to save the dataset to disk for fine-tuning purposes.

- `finetune.py` and `ppo.py`: Scripts to fine-tune a language model using the dataset generated by the parser. The former uses supervised learning to fine-tune the model on the dataset, while the latter uses reinforcement learning to fine-tune the model based on feedback from the Coq verifier, specifically using the proximal policy optimization (PPO) algorithm. Note that these scripts are not used in our experiments but are provided as a tool for future research.

- `infer.py`: Contains functions to generate proof attempts for theorems using a large language model and parse the proof from the response. The proof attempts are generated by providing the theorem statement and the global environment to the model as part of the prompt. See Appendix A for a list of the prompts used. Python's

`joblib` library is used to implement a cache for the prompts and responses to avoid making expensive duplicate requests to the model.

- `lemma-prover.py`: Contains the implementation of the theorem proving algorithm using lemma extraction as described in § 4.2.1. This file accepts settings to control, among other things, the theorem to prove or the project to parse and run a sweep on, the language model to use, and the maximum recursive depth of lemma extraction. When run on a project, the prover saves the results of the proof attempts generated during each theorem's run as well as the generated proof scripts to disk in the `data/raw/{project_name}/proofs` directory. Successful proofs are given the filename `thm{thm_idx}_{thm_name}.v` while failed proofs are named `thm{thm_idx}_err_{thm_name}.v`.

Although our experiments use specific models as discussed in § 4.4.1, the playground is designed to be model-agnostic and can be used with any large language model, including GPT models in OpenAI's suite, pre-trained models hosted inside a model repository on HuggingFace, as well as locally trained or fine-tuned models.

More detailed instructions on how to use the playground to prove a specific theorem or run experiments are available in the README of the GitHub repository.

4.4 Evaluation

To evaluate our lemma extraction approach, we compare the performance of a large language model used in conjunction with lemma extraction to that of a single whole-proof approach as used in Baldur. We further report baseline results of existing automation tools in

Coq, specifically the `auto` and `intuition` tactics as well as the `hammer` tactic from the state-of-the-art CoqHammer plugin [9]. See § 3.1 for a discussion of these tools. We use `hammer` with the CVC4 [3] automated theorem prover and the default time limit of 20 seconds.

We additionally analyze the kinds of errors that the models make when attempting to generate entire proofs of theorems, and the kinds of errors that lemma extraction is effective in addressing. We consider a proof to be valid if it is successfully verified by the Coq proof assistant (exposed to Python through Alectryon, as described in § 4.3) and does not contain sentences that allow it to exit a proof without completing it, such as `Admitted` or `Abort`.

4.4.1 Models

We run experiments using two large language models: GPT-4, specifically the `gpt-4-1106-preview` model, and the open-source model `Phind-CodeLlama-34B-v2`[3]. The former is a general purpose foundational model available through the OpenAI API, while the latter is a much smaller 34B parameter model that is a fine-tuned and instruction-following variant of CodeLlama [45], which itself is a variant of the foundational model Llama 2 [52] that has been specifically fine-tuned to perform well on code generation tasks. At the time of writing, Phind CodeLlama is the state-of-the-art among open-source models for code generation tasks and achieves a performance of 73.8% pass@1 on the HumanEval benchmark [6]. We focus our investigations on these base models and do not further fine-tune them on Coq-specific data.

It is worth noting that the pre-training data for these models may have been contaminated with some or all of the theorems in our dataset, and this should be kept in mind

[3] https://huggingface.co/Phind/Phind-CodeLlama-34B-v2

when interpreting results. We do not believe this to be a significant issue, however, since this work focuses on improvements in performance when using our lemma extraction approach, as opposed to merely the base performance of these models on this dataset.

4.4.2 Dataset

We use the theorems and exercises from the Software Foundations, Volume 1: Logical Foundations textbook [40], sourced from a public GitHub repository[4] available under the MIT License, as our evaluation dataset. This dataset consists of 724 theorems and exercises in Coq, which we use to evaluate the whole-proof generation success rate of the base models, as well as the performance of our theorem proving approach and the automation baselines. These theorems are split across 17 source files—each corresponding to a chapter in the textbook—spanning a variety of topics in logic and functional programming, including induction, lists, maps, polymorphism, and parsing.

A copy of this dataset is included in our repository under the `data/raw/software_foundations` directory. The generated proofs for the theorems are available in the `proofs_gpt4` and `proofs_phind` subdirectories for GPT-4 and Phind CodeLlama respectively.

4.4.3 Results

Table 4.1 shows a comparison of the performance of the lemma extraction approach using each of the two language models, measured as the percentage of theorems in the dataset successfully proven, with various baseline methods.

The results show, firstly, that whole-proof generation is a promising approach for

[4] https://github.com/marshall-lee/software_foundations

Method	Count	%	Improvement %
GPT-4 + Lemma Extraction	575	79.42%	19.54%
GPT-4 Baseline	481	66.44%	
Phind-CodeLlama + Lemma Extraction	189	26.10%	103.23%
Phind-CodeLlama Baseline	93	12.84%	
`auto`	260	35.91%	
`intuition`	331	45.72%	
`hammer`	570	78.73%	

Table 4.1: Performance of the lemma extraction approach using GPT-4 and Phind CodeLlama on the 724 theorems in the dataset, compared to baseline methods. The improvement percentage is calculated as the percentage increase in the number of theorems proven by a given language model when using lemma extraction compared to the same model without.

theorem proving in Coq, with GPT-4 achieving a success rate of 66.44% even without any fine-tuning. This is significantly higher than the 35.91% and 45.72% achieved by `auto` and `intuition` respectively, but lower than the 78.73% achieved by `hammer`.

The Phind CodeLlama model, however, performs remarkably more poorly than GPT-4, with a success rate of only 12.84%. Indeed, even the automation tools built into Coq—`auto` and `intuition`—are able to prove a larger fraction of the theorems in the dataset than Phind CodeLLama, with or without lemma extraction. This is despite such proofs being solvable using a single tactic, indicating that the language model is unable to identify theorems that are simple enough to be provable using these tactics; instead, the proofs output by the model attempt to prove the theorem from scratch, without the help of automation. This is likely due to a combination of the small parameter size of the model and the limited quantity of Coq proofs in its pre-training data.

Next, we look at the performance of both models when augmented with the lemma extraction approach. GPT-4 with lemma extraction is able to prove 79.42% of the theorems in the dataset, a 19.54% improvement over its baseline performance. Notably, this

improved success rate even slightly exceeds the performance of the `hammer` tactic from CoqHammer. In the case of Phind CodeLlama, the success rate more than doubles in the presence of lemma extraction: successful proofs were found for 26.10% of the theorems in the dataset, i.e. a 103.23% improvement over the whole-proof method.

These results have several implications on possibilities for the future of automated theorem provers. Our results suggest that lemma extraction is reasonably effective for large foundational models, but particularly effective for smaller models like Phind CodeLlama. Despite the latter's poor performance on the whole-proof task, the lemma extraction approach is able to significantly improve its success rate, even though the same model is used to prove the extracted lemmas with no additional step-by-step guidance. If such techniques were to be integrated into plugins for Coq, models at the scale of GPT-4 may be far too large to run locally, but there may be possibilities for smaller open-source models like Phind CodeLlama to be used as tools to assist in proof writing. Having said that, further research on fine-tuned versions of these open-source models is needed to determine if they can outperform existing automation tools with the help of lemma extraction.

To conclude, we answer the research questions posed at the beginning of the chapter.

1. **How effective are large language models at generating whole proofs of theorems in Coq?**

   Large language models have shown to be reasonably effective at generating whole proofs of theorems in Coq, with GPT-4 achieving a success rate of 66.44% on the dataset and outperforming built-in automation tools like `auto` and `intuition` but not powerful external automated reasoning tools such as CoqHammer. Smaller models, however, seem to perform much worse without fine-tuning on Coq proofs:

Phind CodeLlama, for instance, has a success rate of only 12.84% and is outperformed by even simple automation tools like `auto`.

2. **Can their performance on the whole-proof generation task be improved by extracting lemmas at failure points during proof generation and recursively generating whole proofs for these lemmas?**
   Yes, there is evidence to show that the performance of large language models can be significantly improved by extracting lemmas at failure points during proof generation and recursively generating whole proofs for these lemmas. GPT-4 solves 79.42% of the theorems in the dataset, thus marginally exceeding the success rate of even the CoqHammer tool. Phind CodeLlama on the other hand solves 26.10% using this approach, representing a remarkable 103.23% improvement over its baseline performance.

4.4.4 Error Analysis

In order to understand the kinds of errors made by a large language model's whole-proof attempt and particularly the errors that lemma extraction is effective in addressing, we manually analyze five randomly sampled examples of the failed proofs generated by each of GPT-4 (1–5) and Phind CodeLlama (6–10) that were successfully proven using lemma extraction. Appendix B contains these examples along with the ground-truth proofs from our dataset, the failed whole-proof attempts, and the successful proofs found by our algorithm.

The most common error made by the models among these examples is the use of incorrect hypotheses, i.e. ones that do not directly apply to the current goal. Examples 3, 4,

5, 6, 8, and 9 show errors due to the use of such incorrect hypotheses. In all these cases, the model correctly predicts the general structure of the proof but falters either within some subgoal or towards the end of the proof, allowing the "smaller subgoal" lemmas to guide it to the correct proof. In some cases, the model uses the right equations but in the wrong order, as seen in example 2. This example shows two failures for this reason, but a reminder of the goal in the form of a single lemma in each case is sufficient to complete the proof. Example 1 depicts a hallucination of a non-existent hypothesis in the proof attempt which is corrected with the help of a chain of three lemmas extracted recursively. Examples 7 and 10 result in errors due to the use of `reflexivity` when the goal is not quite yet a trivial equality but requires some simplification or case analysis first.

One noteworthy observation is that the proofs generated by the lemma extraction procedure are often longer than the ground-truth proofs in the dataset. In some cases, the theorem is very easy to prove—examples 8 and 9, for instance, can be proven directly using a single application of `reflexivity`—but the model generates a longer proof by induction that results in an error. Extracting lemmas on top of this only causes the resulting proof to be longer still, albeit correct. Similarly, example 5 is a simple result about the commutativity of addition that can be proven using induction. While the model correcty identifies this, it makes errors in both the base case and the inductive step that result in four lemmas being necessary to complete the proof.

Additionally, one limitation of our approach is that it may not be very effective when an error occurs at or near the beginning of the proof. In such cases, the goal may not have transformed enough to be useful as a lemma, and the model may well make the same error while attempting to prove the lemma since it is very similar to the original theorem. This is

seen in example 6, where an error is made immediately after the use of `intros`, resulting in a lemma that is almost identical to the original goal except for the use of quantified hypothesis variables instead of a chain of implications. In this case, the model succeeds at finding a proof for the lemma despite faltering initially, but it is possible that this was due to the simplicity of the lemma; more complex lemmas could lead to a sequence of identical lemmas being recursively extracted until the maximum depth is quickly reached.

5

Conclusion

In this thesis, we introduced and investigated a novel approach to automated theorem proving in Coq that makes use of whole-proof generation in conjunction with lemma extraction at failure points to come up with proofs of theorems. We presented a proof-of-concept implementation of this in Python. We demonstrated that this approach is effective

when used with large foundational models by showing improvements over the baseline method of using the model on its own; with GPT-4, we were able to achieve performance similar to that of the state-of-the-art automation tool, CoqHammer. We also showed that this procedure can provide large relative improvements over the baseline method for small open-source models, although the absolute performance of these models is still far from that of existing automation tools.

Even the baseline GPT-4 success rate is somewhat surprising, as it suggests that the model is able to perform some degree of reasoning within Coq proofs without needing to be explicitly reminded of the proof state at every step. This is despite Coq using a procedural language unlike Isabelle's declarative Isar. While further investigation is needed to conclusively determine if this scales and will work on other larger datasets and projects, it is nonetheless a promising result.

In the broader context of the field of automated theorem proving, this work differs from most previous approaches described in § 3.3 in that it does not use expensive search-based methods to find proofs, nor does it generate tactics one step at a time. Similar to Baldur [15], it generates whole proofs at once, but it does so in a guided manner by "reminding" the language model of what needs to be shown at each error point through the use of lemmas, which are then proven in the same way.

Finally, we also described and released a Coq playground that allows users to use this lemma extraction approach to generate proofs for their own theorems and projects, in addition to running the experiments described in this thesis.

5.1 Future Work

There are several directions in which this work can be extended.

1. One bottleneck of our current implementation is the limited context window size of large language models. We currently include the statements of all definition and theorems in the global environment of the proof in the prompt, which is not scalable for larger files and projects. One possible solution is to use retrieval augmented generation to fix this issue, where we retrieve relevant premises from the file as well as libraries and other imported files and include them in the prompt, in a manner similar to LeanDojo [61].

2. Although we did not run experiments on this, our Coq playground does support variations of the base approach, such as providing the previous erroneous attempts along with the error message to the LLM. This could help reduce the possibility of the same error being repeated in a chain of recursive lemmas as the model is given additional context as to where and why the error occurred.

3. This thesis only addresses the performance and performance improvement of base models. It may be of interest to investigate the performance of models fine-tuned on Coq proofs, especially open-source models that are small enough to be used locally, to study the effect of fine tuning on whole-proof generation in a procedural proof assistant like Coq, with the goal of surpassing existing automation tools.

4. More broadly speaking, extracting lemmas from points of error is only one way to split a proof into smaller, more manageable pieces. Other methods could also be ex-

plored; one possible direction of research is to attempt to identify lemmas that are likely to be useful in the proof even before the proof is generated. Using the `lfind` tactic in Coq, for instance, in conjunction with whole-proof generation is one possible direction for research (see § 3.2.3). DSP [28] uses another variation of this in the context of autoformalization for Isabelle; instead of explicitly extracting lemmas, they use an informal proof to generate a sketch of a formal proof. In Coq, such a sketch could be created as a sequence of `assert` tactics that act as the lemmas.

We are still some distance away from a future where formal proof writing is so effortless due to automation that every theoretical paper published in mathematics or computer science comes with fully verified formal proofs. We are further still from a future where, given a theorem, a computer can automatically generate a correct proof of it at the push of a button. But we are making progress towards such futures—towards Leibniz's *calculus ratiocinator*—and I hope the work presented in this thesis serves as a small step in that direction.

A

Prompts Used

We provide the prompt templates used in the experiments in this thesis; these are also available in the Coq playground.

Each time a theorem or lemma needs to proven, we make a request to the model to generate a proof, without retaining context from previous requests to avoid bias and repeti-

tion of the same errors. We use a system prompt along with a user message.

System message:

```
You are an automated theorem prover that can prove theorems and lemmas in
Coq. Your entire response must be valid Coq code. You should explain your
reasoning (what the proof steps are attempting to do), but only in
comments inside the Coq code. The following messages will all consist of a
theorem statement (possibly preceded by necessary definitions, imports,
etc.), and your response must be a valid Coq proof of that theorem. Your
response must be in this format: ```coq
Proof.
<proof>. Qed.
```. Remember: do not add any other text besides Coq code and do not
repeat any imports, definitions, lemmas, etc. provided in the prompt.
```

User message:

```
#### CONTEXT

{global environment}

#### THEOREM TO BE PROVEN

{theorem statement}
```

We also use the language model to generate names for the extracted lemmas, which are then suffixed with an index to ensure uniqueness.

System message:

```
You are a proof helper for Coq that can come up with descriptive names for
lemmas and theorems based on the statement of the proposition.
Specifically, Replace `helper_lemma` with a better, more descriptive, name
for the following lemma(s) in Coq. Your entire response must be valid Coq
code. Your response must be in this format: ```coq\nLemma <new_lemma_name>
: <lemma_statement>.\n```.
```
```

User message:

```
{lemma statement}
```

Note that the OpenAI API for GPT-4 accepts the system and user messages as separate inputs. For Phind CodeLlama, we use the following template, in accordance with the Alpaca/Vicuna instruction format as specified in the model's documentation.

```
### System Prompt
{system message}

### User Message
{user message}

### Assistant
```

B

Examples of Proofs Found

This appendix contains examples of theorems that were unable to be proven by a single whole-proof sample from the language model, but were successfully proven by the model when broken down into lemmas. We present five randomly sampled examples each for GPT-4 and Phind CodeLlama. For each theorem, we include the ground-truth proof

from the dataset, the incorrect whole-proof attempt by the model, the error, and the final correct proof (along with lemmas extracted). We retain comments generated by the models, but omit the global environment to save space. The complete proof scripts including the global environment can be found in the GitHub repository.

GPT-4 Examples

Example 1: Theorem 53 (plus2_spec)

Ground-truth proof:

```
Theorem plus2_spec : forall st n st',
  st X = n →
  st =[ plus2 ]⇒ st' →
  st' X = n + 2.
Proof.
  intros st n st' HX Heval.

  (** Inverting [Heval] essentially forces Coq to expand one step of
      the [ceval] computation -- in this case revealing that [st']
      must be [st] extended with the new value of [X], since [plus2]
      is an assignment. *)

  inversion Heval. subst. clear Heval. simpl.
  apply t_update_eq.  Qed.
```

Incorrect whole-proof attempt:

```
Theorem plus2_spec : forall st n st',
  st X = n →
  st =[ plus2 ]⇒ st' →
  st' X = n + 2.
Proof.
  intros st n st' H H1.
  inversion H1; subst; clear H1.
```

```
  (* Apply inversion on H1 to get the structure of plus2
     command and rewrite the state *)
  simpl in H5.
  (* Simplify the expression in the hypothesis H5 *)
  rewrite H in H5.
  (* Replace st X with n in H5 *)
  apply H5.
  (* The conclusion follows directly from H5 *)
Qed.
```

Error in step `simpl in H5.`

```
No such hypothesis: H5.
```

Correct proof after lemma extraction:

```
Lemma equality_of_identical_sum_with_2_2 : forall st : string → nat,
  st X + 2 = st X + 2.
Proof.
  (* We can prove this lemma by reflexivity,
     as both sides are already syntactically equal.
  *)
  intros st.
  reflexivity.
Qed.

Lemma aeval_x_plus_2_eq_st_x_plus_2_1 : forall st : string → nat,
  (if (X =? X)%string
   then aeval st <{ X + 2 }>
   else st X) = st X + 2.
Proof.
  intros st.
  (* Introduce the state `st` *)
  simpl.
  (* Simplify the aexp `<{X + 2}>` *)
  apply (@equality_of_identical_sum_with_2_2 st).
Qed.

Lemma eval_aexp_plus_two_0 : forall st : string → nat,
```

```
  (X !-> aeval st <{ X + 2 }>; st) X = st X + 2.
Proof.
  intros st.
  unfold t_update.
  (* Revealing the behavior of function [t_update] *)
  apply (@aeval_x_plus_2_eq_st_x_plus_2_1 st).
Qed.

Theorem plus2_spec : forall st n st',
  st X = n ->
  st =[ plus2 ]=> st' ->
  st' X = n + 2.
Proof.
  intros st n st' H H1.
  inversion H1; subst; clear H1.
  (* Apply inversion on H1 to get the structure of plus2 command
     and rewrite the state *)
  apply (@eval_aexp_plus_two_0 st).
Qed.
```

Example 2: Theorem 268 (bool_fn_applied_thrice)

Ground-truth proof:

```
Theorem bool_fn_applied_thrice :
  forall (f : bool -> bool) (b : bool),
  f (f (f b)) = f b.
Proof.
  intros f b.
  destruct b.
  + destruct (f true) eqn:T.
    - rewrite -> T. rewrite -> T. reflexivity.
    - destruct (f false) eqn:F.
      * rewrite -> T. reflexivity.
      * rewrite -> F. reflexivity.
  + destruct (f false) eqn:F.
    - destruct (f true) eqn:T.
      * rewrite -> T. reflexivity.
```

```
      * rewrite → F. reflexivity.
    - rewrite → F. rewrite → F. reflexivity.
Qed.
```

Incorrect whole-proof attempt:

```
Theorem bool_fn_applied_thrice : forall (f : bool → bool) (b : bool),
  f (f (f b)) = f b.
Proof.
intros f b.
destruct b eqn:E.
- destruct (f true) eqn:F.
  + rewrite F.
    rewrite F.
    reflexivity.
  + destruct (f false) eqn:FF.
    * rewrite F.
      reflexivity.
    * rewrite FF.
      reflexivity.
- destruct (f false) eqn:F.
  + destruct (f true) eqn:FT.
    * rewrite F.
      reflexivity.
    * rewrite FT.
      reflexivity.
  + rewrite F.
    rewrite F.
    reflexivity.
Qed.
```

Error in step `rewrite F.`

```
Found no subterm matching "f false" in the current goal.
```

Error in step `rewrite FT.`

```
Found no subterm matching "f true" in the current goal.
```

Correct proof after lemma extraction:

```
Lemma f_true_fixed_point_0 : forall f : bool → bool,
  forall b : bool, forall E : b = false, forall F : f false = true,
  forall FT : f true = true, f true = true.
Proof.
(* This theorem states that if `f true` equals `true`, then `f true`
equals `true` under any conditions for `b` and any mapping of `false`
to `true` by `f`.
This is trivially true by assumption `FT`. *)
  intros f b E F FT.
(* We apply the assumption `FT` directly. *)
  apply FT.
Qed.

Lemma f_false_implies_true_1 : forall f : bool → bool, forall b : bool,
forall E : b = false, forall F : f false = true,
forall FT : f true = false, f false = true.
Proof.
(* Directly use the hypothesis F, which already states that
   f false = true. *)
  intros f b E F FT.
  exact F.
Qed.

Theorem bool_fn_applied_thrice : forall (f : bool → bool) (b : bool),
  f (f (f b)) = f b.
Proof.
intros f b.
destruct b eqn:E.
- destruct (f true) eqn:F.
  + rewrite F.
    rewrite F.
    reflexivity.
  + destruct (f false) eqn:FF.
    * rewrite F.
      reflexivity.
    * rewrite FF.
      reflexivity.
```

```
- destruct (f false) eqn:F.
  + destruct (f true) eqn:FT.
    * apply (@f_true_fixed_point_0 f b E F FT).
    * apply (@f_false_implies_true_1 f b E F FT).
  + rewrite F.
    rewrite F.
    reflexivity.
Qed.
```

Example 3: Theorem 300 (ev_sum)

Ground-truth proof:

```
Theorem ev_sum : forall n m, ev n → ev m → ev (n + m).
Proof.
  intros n m En Em.
  induction En.
  - apply Em.
  - simpl. apply ev_SS. apply IHEn.
Qed.
```

Incorrect whole-proof attempt:

```
Theorem ev_sum : forall n m, ev n → ev m → ev (n + m).
Proof.
  intros n m Hn Hm.
  induction Hn as [| n' IHn].
  - (* case: ev_0, when n is 0, which is trivially even *)
    simpl.
    (* 0 + m simplifies to m *)
    assumption.
    (* since m is even by hypothesis, we're done *)
  - (* case: ev_SS, the inductive case when n = S (S n') *)
    simpl.
    (* (S (S n')) + m simplifies to S (S (n' + m)) *)
    apply ev_SS.
    (* we want to show ev (n' + m), which is our IH *)
    apply IHn.
```

```
    (* apply the inductive hypothesis to conclude ev (n' + m) *)
Qed.
```

Error in step `apply IHn.`

```
In environment
m, n' : nat
IHn : ev n'
Hm : ev m
IHIHn : ev (n' + m)
Unable to unify "ev n'" with
"ev (n' + m)".
```

Correct proof after lemma extraction:

```
Lemma ev_sum_preservation_1 : forall m n' : nat,
  ev n' → ev m → ev (n' + m) → ev (n' + m).
Proof.
  (* This Lemma is trivial because the goal is the same as one
   of the premises. *)
  intros m n' Hn' Hm Hev.
  assumption.
Qed.

Lemma ev_sum_preserved_0 : forall m n' : nat, forall IHn : ev n',
  forall Hm : ev m, forall IHIHn : ev (n' + m), ev (n' + m).
Proof.
(* Since we know that IHIHn already states `ev (n' + m)`,
the goal `ev (n' + m)` is directly proven by assumption. *)
  apply (@ev_sum_preservation_1 ).
Qed.

Theorem ev_sum : forall n m, ev n → ev m → ev (n + m).
Proof.
intros n m Hn Hm.
induction Hn as [| n' IHn].
  - (* case: ev_0, when n is 0, which is trivially even *)
    simpl.
  (* 0 + m simplifies to m *)
```

```
    assumption.
  (* since m is even by hypothesis, we're done *)
  - (* case: ev_SS, the inductive case when n = S (S n') *)
    simpl.
  (* (S (S n')) + m simplifies to S (S (n' + m)) *)
    apply ev_SS.
  (* we want to show ev (n' + m), which is our IH *)
    apply (@ev_sum_preserved_0 m n' IHn Hm IHIHn).
Qed.
```

Example 4: Theorem 315 (plus_le)

Ground-truth proof:

```
Theorem plus_le : forall n1 n2 m,
  n1 + n2 ⩽ m →
  n1 ⩽ m ∧ n2 ⩽ m.
Proof.
  intros n1 n2 m H.
  induction H.
  - split.
    + apply le_plus_l.
    + rewrite add_comm. apply le_plus_l.
  - destruct IHle as [H1 H2].
    split.
    + apply (le_S n1 m H1).
    + apply (le_S n2 m H2).
Qed.
```

Incorrect whole-proof attempt:

```
Theorem plus_le : forall n1 n2 m,
  n1 + n2 ⩽ m →
  n1 ⩽ m ∧ n2 ⩽ m.
Proof.
(* Assume given natural numbers n1, n2, and m such that n1 + n2
   is less than or equal to m. *)
  intros n1 n2 m H.
```

```
(* We have to prove the conjunction of n1 ⩽ m and n2 ⩽ m.
   To do so, we will split the goal into two subgoals using the
   and_intro lemma. *)
  split.
- (* Prove the first subgoal: n1 ⩽ m.
  We will show that any number is less than or equal to itself plus
  something (le_plus_l).
  Hence, n1 ⩽ n1 + n2, and using transitivity of le (le_trans),
  we can combine it with the hypothesis to conclude n1 ⩽ m. *)
  apply (le_trans _ (n1 + n2) _).
  + apply le_plus_l.
  + apply H.
- (* Prove the second subgoal: n2 ⩽ m.
     We will use the same strategy as for the first subgoal,
     but instea of le_plus_l, we need a lemma stating n2 ⩽ n2 + n1,
     which is a variation and application of le_plus_l with
     added commutativity of addition. *)
  apply (le_trans _ (n2 + n1) _).
  + rewrite add_comm.
    apply le_plus_l.
  + rewrite add_comm in H.
    apply H.
Qed.
```

Error in step `apply le_plus_l.`

```
In environment
n1, n2, m : nat
H : n1 + n2 ⩽ m
Unable to unify "n1" with "n2".
```

Correct proof after lemma extraction:

```
Lemma add_increases_right_0 : forall n1 n2 m : nat,
  forall H : n1 + n2 ⩽ m, n2 ⩽ n1 + n2.
Proof.
(* We can prove this by applying the theorem le_plus_l which
   states that for any a b, a ⩽ a + b.
   In this context, it straightforwardly translates to n2 ⩽ n2 + n1,
```

```
    which is precisely n2 ⩽ n1 + n2
    when adding n1 and n2 in the reverse order because addition in
    natural numbers is commutative. *)
  intros n1 n2 m H.
rewrite ← add_comm.
(* Use commutativity of addition to transform n1 + n2 into n2 + n1 *)
  apply le_plus_l.
(* Apply the theorem le_plus_l *)
Qed.

Theorem plus_le : forall n1 n2 m,
  n1 + n2 ⩽ m →
  n1 ⩽ m ∧ n2 ⩽ m.
Proof.
(* Assume given natural numbers n1, n2, and m such that n1 + n2
   is less than or equal to m. *)
  intros n1 n2 m H.
(* We have to prove the conjunction of n1 ⩽ m and n2 ⩽ m.
   To do so, we will split the goal into two subgoals using the
   and_intro lemma. *)
  split.
- (* Prove the first subgoal: n1 ⩽ m.
     We will show that any number is less than or equal to itself plus
     something (le_plus_l).
     Hence, n1 ⩽ n1 + n2, and using transitivity of le (le_trans),
     we can combine it with the hypothesis to conclude n1 ⩽ m. *)
  apply (le_trans _ (n1 + n2) _).
  + apply le_plus_l.
  + apply H.
- (* Prove the second subgoal: n2 ⩽ m.
We will use the same strategy as for the first subgoal,
       but instea of le_plus_l, we need a lemma stating n2 ⩽ n2 + n1,
       which is a variation and application of le_plus_l
       with added commutativity of addition. *)
  apply (le_trans _ (n2 + n1) _).
  + rewrite add_comm.
    apply (@add_increases_right_0 n1 n2 m H).
  + rewrite add_comm in H.
```

```
    apply H.
Qed.
```

Example 5: Theorem 676 (add_comm)

Ground-truth proof:

```
Theorem add_comm : forall n m : nat,
  n + m = m + n.
Proof.
  intros n m.
  induction n as [| n' IHn'].
  - (* n = 0 *)
    simpl. rewrite ← plus_n_O. reflexivity.
  - (* n = S n' *)
    simpl. rewrite ← plus_n_Sm. rewrite ← IHn'. reflexivity.
Qed.
```

Incorrect whole-proof attempt:

```
Theorem add_comm : forall n m : nat,
  n + m = m + n.
Proof.
(* We prove this lemma by induction on n. *)
  intros n.
  induction n as [| n' IHn'].
  - (* The base case: 0 + 0 = 0 *)
      reflexivity.
  - (* The inductive case: S n' + 0 = S n' *)
      simpl.
  rewrite IHn'.
  reflexivity.
Qed.
```

Error in step `reflexivity.`

```
In environment
m : nat
```

```
Unable to unify "m + 0" with
"0 + m".
```

Error in step `rewrite IHn'.`

```
Found no subterm matching
"n' + ?M1094" in the current goal.
```

Correct proof after lemma extraction:

```
Lemma add_zero_comm_1 : forall m : nat, 0 + m = m + 0.
Proof.
  intros m.
  rewrite ← plus_n_O.
  reflexivity.
Qed.

Lemma add_zero_comm_0 : forall m : nat, 0 + m = m + 0.
Proof.
  (* We use induction on `n` to prove that for all natural numbers
   `n`, `n + 0 = n`. *)
  apply (@add_zero_comm_1 ).
Qed.

Lemma succ_plus_comm_2 : forall n' : nat, forall IHn' : forall m : nat,
  n' + m = m + n', forall m : nat, S (n' + m) = m + S n'.
Proof.
(* n' is fixed by forall so we can introduce the second premise (IHn')
   which is the inductive hypothesis that will be used later in the proof *)
  intros n' IHn' m.
(* Apply the previously introduced plus_n_Sm theorem which states that
   S (n + m) = n + S m *)
  rewrite ← plus_n_Sm.
(* Now apply our inductive hypothesis which will transform
   n' + m into m + n' *)
  rewrite IHn'.
(* At this point we have S m + n', which we want to show is equal to
   m + S n'.*)
  reflexivity.
```

```
Qed.

Lemma succ_plus_comm_1 : forall n' : nat, forall IHn' : forall m : nat,
  n' + m = m + n', forall m : nat, S (n' + m) = m + S n'.
Proof.
(* Let's prove the lemma by introducing the necessary variables. *)
  intros n' IHn' m.
(* Use the provided induction hypothesis (IHn'). *)
  apply (@succ_plus_comm_2 n' IHn' m).
Qed.

Theorem add_comm : forall n m : nat,
  n + m = m + n.
Proof.
(* We prove this lemma by induction on n. *)
  intros n.
  induction n as [| n' IHn'].
  - (* The base case: 0 + 0 = 0 *)
      apply (@add_zero_comm_0 ).
  - (* The inductive case: S n' + 0 = S n' *)
      simpl.
  apply (@succ_plus_comm_1 n' IHn').
Qed.
```

Phind CodeLlama Examples

Example 6: Theorem 81 (and_example2'')

Ground-truth proof:

```
Lemma and_example2'' :
  forall n m : nat, n = 0 → m = 0 → n + m = 0.
Proof.
  intros n m Hn Hm.
  rewrite Hn. rewrite Hm.
  reflexivity.
Qed.
```

Incorrect whole-proof attempt:

```
Lemma and_example2'' : forall n m : nat, n = 0 → m = 0 → n + m = 0.
Proof.
(* The goal is to prove that if n is 0 and m is 0, then n + m is 0.
   We will do this by applying the and_intro lemma, which states that
    if A and B are propositions and we have proofs of A and B,
    then we can construct a proof of A ∧ B. *)
  intros n m Hn Hm.
(* We now need to apply the and_intro lemma with A = n = 0 and B = m = 0.
   We have proofs of these propositions (Hn and Hm), so we can apply and_intro. *)
  apply (and_intro n m Hn Hm).
(* This concludes the proof. *)
Qed.
```

Error in step `apply (and_intro n m Hn Hm).`

```
In environment
n, m : nat
Hn : n = 0
Hm : m = 0
The term "n" has type "nat"
while it is expected to have type "Prop".
```

Correct proof after lemma extraction:

```
Lemma zero_add_zero_0 : forall n m : nat, forall Hn : n = 0,
  forall Hm : m = 0, n + m = 0.
Proof.
  intros n m Hn Hm.
  rewrite Hn, Hm.
  reflexivity.
Qed.

Lemma and_example2'' : forall n m : nat, n = 0 → m = 0 → n + m = 0.
Proof.
(* The goal is to prove that if n is 0 and m is 0, then n + m is 0.
   We will do this by applying the and_intro lemma, which states that
     if A and B are propositions and we have proofs of A and B,
```

```
      then we can construct a proof of A ∧ B. *)
  intros n m Hn Hm.
(* We now need to apply the and_intro lemma with A = n = 0 and B = m = 0.
   We have proofs of these propositions (Hn and Hm), so we can apply and_intro. *)
  apply (@zero_add_zero_0 n m Hn Hm).
Qed.
```

Example 7: Theorem 87 (factor_is_0)

Ground-truth proof:

```
Lemma factor_is_0:
  forall n m : nat, n = 0 ∨ m = 0 → n * m = 0.
Proof.
  (* This pattern implicitly does case analysis on
     [n = 0 ∨ m = 0] *)
  intros n m [Hn | Hm].
  - (* Here, [n = 0] *)
    rewrite Hn. reflexivity.
  - (* Here, [m = 0] *)
    rewrite Hm. rewrite ← mult_n_0.
    reflexivity.
Qed.
```

Incorrect whole-proof attempt:

```
Lemma factor_is_0 : forall n m : nat, n = 0 ∨ m = 0 → n * m = 0.
Proof.
  intros n m H.
  destruct H as [Hn | Hm].
  - (* n = 0 *)
    rewrite Hn.
  reflexivity.
  - (* m = 0 *)
    rewrite Hm.
  reflexivity.
Qed.
```

Error in step `reflexivity`.

```
In environment
n, m : nat
Hm : m = 0
Unable to unify "0" with "n * 0".
```

Correct proof after lemma extraction:

```
Lemma mult_zero_right_neutral_1 : forall n' m : nat,
  forall Hm : m = 0, forall IHn' : n' * 0 = 0, n' * m = m.
Proof.
intros n' m Hm IHn'.
rewrite Hm.
(* We know that m = 0, so n' * m = n' * 0 by the equality m = 0. *)
  rewrite IHn'.
(* We know that n' * 0 = 0 by the induction hypothesis. *)
(* Therefore, n' * m = 0. *)
  reflexivity.
Qed.

Lemma multiplication_of_zero_0 : forall n m : nat,
  forall Hm : m = 0, n * 0 = 0.
Proof.
  intros n m Hm.
  induction n as [|n' IHn'].
  - (* n = 0 *)
    simpl.
  reflexivity.
  - (* n = S n' *)
    simpl.
  rewrite ← Hm.
  apply (@mult_zero_right_neutral_1 n' m Hm IHn').
Qed.

Lemma factor_is_O : forall n m : nat, n = 0 ∨ m = 0 → n * m = 0.
Proof.
  intros n m H.
  destruct H as [Hn | Hm].
```

```
  - (* n = 0 *)
    rewrite Hn.
  reflexivity.
  - (* m = 0 *)
    rewrite Hm.
    apply (@multiplication_of_zero_0 n m Hm).
Qed.
```

Example 8: Theorem 172 (test_length1)

Ground-truth proof:

```
Example test_length1: length (cons 1 (cons 2 (cons 3 nil))) = 3.
Proof. reflexivity. Qed.
```

Incorrect whole-proof attempt:

```
Example test_length1 : length (cons 1 (cons 2 (cons 3 nil))) = 3.
Proof.
(* We will use induction on the list to prove this theorem. *)
  induction list123'' as [|x l' IHl'].
- (* Case for nil: The length of an empty list is 0. *)
  reflexivity.
- (* Case for cons: The length of a non-empty list is 1 plus
     the length of its tail. *)
  simpl.
  rewrite IHl'.
  reflexivity.
Qed.
```

Error in step `rewrite IHl'.`

```
Found no subterm matching
"length (cons 1 (cons 2 (cons 3 nil)))" in the current goal.
```

Correct proof after lemma extraction:

```
Lemma length_of_cons_list_0 : forall x : nat, forall l' : list nat,
  forall IHl' : length (cons 1 (cons 2 (cons 3 nil))) = 3, 3 = 3.
Proof.
  intros x l' IHl'.
  simpl.
  reflexivity.
Qed.

Example test_length1 : length (cons 1 (cons 2 (cons 3 nil))) = 3.
Proof.
(* We will use induction on the list to prove this theorem. *)
  induction list123'' as [| x l' IHl'].
- (* Case for nil: The length of an empty list is 0.
  *)
    reflexivity.
- (* Case for cons: The length of a non-empty list is 1 plus the
     length of its tail.
  *)
    simpl.
apply (@length_of_cons_list_0 x l' IHl').
Qed.
```

Example 9: Theorem 479 (nil_app)

Ground-truth proof:

```
Theorem nil_app : forall l : natlist,
  [] ++ l = l.
Proof. reflexivity. Qed.
```

Incorrect whole-proof attempt:

```
Theorem nil_app : forall l : natlist,
  [] ++ l = l.
Proof.
  intros l.
  induction l as [| n l' IHl'].
  - (* nil case *)
```

```
      reflexivity.
  - (* cons case *)
      simpl.
  rewrite IHl'.
  reflexivity.
Qed.
```

Error in step `rewrite IHl'.`

```
Found no subterm matching
"[ ] ++ l'" in the current goal.
```

Correct proof after lemma extraction:

```
Lemma append_nil_to_list_0 : forall n : nat, forall l' : natlist,
  forall IHl' : [ ] ++ l' = l', n :: l' = n :: l'.
Proof.
  intros n l' IHl'.
  simpl.
  reflexivity.
Qed.

Theorem nil_app : forall l : natlist,
  [] ++ l = l.
Proof.
  intros l.
  induction l as [| n l' IHl'].
  - (* nil case *)
      reflexivity.
  - (* cons case *)
      simpl.
  apply (@append_nil_to_list_0 n l' IHl').
Qed.
```

Example 10: Theorem 592 (andb_commutative)

Ground-truth proof:

```
Theorem andb_commutative : forall b c, andb b c = andb c b.
Proof.
  intros b c. destruct b.
  - destruct c.
    + reflexivity.
    + reflexivity.
  - destruct c.
    + reflexivity.
    + reflexivity.
Qed.
```

Incorrect whole-proof attempt:

```
Theorem andb_commutative : forall b c, andb b c = andb c b.
Proof.
(* The theorem andb_commutative states that the and operation
   is commutative for booleans.
   That is, for any two booleans b and c, andb b c is equal to andb c b.
   We will prove this by induction on b and then c. *)
  intros b c.
(* We first consider the base case where b is false. *)
  destruct b as [|b_true].
- (* If b is false, then andb b c is false regardless of the value of c. *)
    simpl.
reflexivity.
- (* If b is true, we need to prove that andb b c = andb c b. *)
    destruct c as [|c_true].
+ (* If c is false, then andb b c is false, and andb c b is also false. *)
      simpl.
reflexivity.
+ (* If c is true, then andb b c = andb c b because both are true. *)
      simpl.
reflexivity.
Qed.
```

Error in step `reflexivity.`

```
In environment
c : bool
```

```
Unable to unify "c && true" with "c"..
Goal: c = c && true.
```

Correct proof after lemma extraction:

```
Lemma false_equals_false_and_true_1 : false = false && true.
Proof.
(* We will use the axiom of equality to prove this lemma.
    The axiom states that for any two terms t and t', we have
    t = t' if and only if t' is a proof of t.
    In this case, we want to prove false = false && true.
    We know that false && true is false, so we need to prove false = false.
    This is trivially true by the axiom of reflexivity,
    which states that any term is equal to itself. *)
  reflexivity.
Qed.

Lemma bool_idempotent_lemma_0 : forall c : bool, c = c && true.
Proof.
(* The goal is to prove that for any boolean c, c && true is equal to c.
We will proceed by case analysis on c. *)
  intros c.
(* First, we consider the case where c is true. *)
  destruct c as [c_true | c_false].
(* In this case, c && true is true. *)
  simpl.
  reflexivity.
(* Now, we consider the case where c is false. *)
  apply (@false_equals_false_and_true_1 ).
Qed.

Theorem andb_commutative : forall b c, andb b c = andb c b.
Proof.
(* The theorem andb_commutative states that the and operation is
   commutative for booleans.
   That is, for any two booleans b and c, andb b c is equal to andb c b.
   We will prove this by induction on b and then c. *)
  intros b c.
(* We first consider the base case where b is false. *)
```

```
35    destruct b as [|b_true].
36  - (* If b is false, then andb b c is false regardless of the value of c. *)
37    simpl.
38    apply (@bool_idempotent_lemma_0 c).
39  - (* If b is true, we need to prove that andb b c = andb c b. *)
40    destruct c as [|c_true].
41  + (* If c is false, then andb b c is false, and andb c b is also false. *)
42      simpl.
43      reflexivity.
44  + (* If c is true, then andb b c = andb c b because both are true. *)
45      simpl.
46      reflexivity.
47  Qed.
```

References

[1] Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W Ayers, Dragomir Radev, and Jeremy Avigad. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint arXiv:2302.12433*, 2023.

[2] Grzegorz Bancerek. Automatic translation in formalized mathematics. *Mechanized Mathematics and Its Applications*, 5(2):19–31, 2006.

[3] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi:10.1007/978-3-642-22110-1_14.

[4] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pages 2397–2430. PMLR, 2023.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios

Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021. arXiv:2107.03374.

[7] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[8] Thierry Coquand and Gérard Huet. *The calculus of constructions*. PhD thesis, INRIA, 1986.

[9] Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for dependent type theory. *Journal of automated reasoning*, 61:423–453, 2018.

[10] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[11] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 378–388. Springer, 2015.

[12] Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Junjie Shan, Caishuang Huang, Wei Shen, Xiaoran Fan, Zhiheng Xi, et al. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*, 2024.

[13] Emily First and Yuriy Brun. Diversity-driven automated formal verification. In *Proceedings of the 44th International Conference on Software Engineering*, pages 749–761, 2022.

[14] Emily First, Yuriy Brun, and Arjun Guha. TacTok: semantics-aware proof synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–31, 2020. doi:10.1145/3428299.

[15] Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1229–1241, 2023.

[16] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. Learning to reason with HOL4 tactics. *arXiv preprint arXiv:1804.00595*, 2018.

[17] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. TacticToe: learning to prove with tactics. *Journal of Automated Reasoning*, 65:257–286, 2021.

[18] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al. A machine-checked proof of the odd order theorem. In *International conference on interactive theorem proving*, pages 163–179. Springer, 2013.

[19] Georges Gonthier et al. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[20] Jesse Michael Han, Igor Babuschkin, Harrison Edwards, Arvind Neelakantan, Tao Xu, Stanislas Polu, Alex Ray, Pranav Shyam, Aditya Ramesh, Alec Radford, et al. Unsupervised neural machine translation with generative language models only. *arXiv preprint arXiv:2110.05448*, 2021.

[21] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. URL: https://openreview.net/forum?id=7Bywt2mQsCe.

[22] William A Howard et al. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

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

[24] Jie Huang and Kevin Chen-Chuan Chang. Towards reasoning in large language models: A survey. *arXiv preprint arXiv:2212.10403*, 2022.

[25] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Een, Francois Chollet, and Josef Urban. Deepmath - deep sequence models for premise selection. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL: https://proceedings.neurips.cc/paper_files/paper/2016/file/f197002b9a0853eca5e046d9ca4663d5-Paper.pdf.

[26] Albert Qiaochu Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. LISA: Language models of isabelle proofs. In *6th Conference on Artificial Intelligence and Theorem Proving*, pages 378–392, 2021.

[27] Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdź, Piotr Miłoś, Yuhuai Wu, and Mateja Jamnik. Thor: Wielding hammers to integrate language models and automated theorem provers. *Advances in Neural Information Processing Systems*, 35:8360–8373, 2022.

[28] Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations*, 2023. URL: https://openreview.net/forum?id=SMa9EAovKMC.

[29] Cezary Kaliszyk, François Chollet, and Christian Szegedy. HolStep: A machine learning dataset for higher-order logic theorem proving. In *International Conference on Learning Representations*, 2017. URL: https://openreview.net/forum?id=ryuxYmvel.

[30] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906*, 2020.

[31] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

[32] Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. *Advances in Neural Information Processing Systems*, 35:26337–26349, 2022.

[33] Gottfried Wilhelm Leibniz. *Selections*. Scribner, New York, 1951.

[34] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[35] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *Advances in Neural Information Processing Systems*, 35:3843–3857, 2022.

[36] Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024. URL: https://openreview.net/forum?id=v8L0pN6EOi.

[37] Norman D. Megill. *Metamath: A Computer Language for Mathematical Proofs*. Lulu Press, Morrisville, North Carolina, 2019. URL: http://us.metamath.org/downloads/metamath.pdf.

[38] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: A proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[39] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010. URL: https://doi.org/10.29007/36dt, doi:10.29007/36DT.

[40] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*, volume 1 of *Software Foundations*. Electronic textbook, 2023. Version 6.5. URL: https://softwarefoundations.cis.upenn.edu.

[41] Clément Pit-Claudel. Untangling mechanized proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2020, page 155–174, New York, NY, USA, 2020. Association for Computing Machinery. URL: https://pit-claudel.fr/clement/papers/alectryon-SLE20.pdf, doi:10.1145/3426425.3426940.

[42] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

[43] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[44] Hartley Rogers Jr. An example in mathematical logic. *The American Mathematical Monthly*, 70(9):929–945, 1963.

[45] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[46] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 1–10, 2020.

[47] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[48] Stephan Schulz. E–a brainiac theorem prover. *Ai Communications*, 15(2-3):111–126, 2002.

[49] Aishwarya Sivaraman, Alex Sanchez-Stern, Bretton Chen, Sorin Lerner, and Todd Millstein. Data-driven lemma synthesis for interactive proofs. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):505–531, 2022.

[50] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[51] The Coq Development Team. The Coq proof assistant, July 2023. doi:10.5281/zenodo.8161141.

[52] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[53] Trieu H Trinh, Yuhuai Wu, Quoc V Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024.

[54] Karthik Valmeekam, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Large language models still can't plan (a benchmark for LLMs on planning and reasoning about change). In *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022. URL: https://openreview.net/forum?id=wUU-7XTL5XO.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[56] Qingxiang Wang, Cezary Kaliszyk, and Josef Urban. First experiments with neural translation of informal to formal mathematics. In *Intelligent Computer Mathematics: 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings 11*, pages 255–270. Springer, 2018.

[57] Sean Welleck, Jiacheng Liu, Ronan Le Bras, Hannaneh Hajishirzi, Yejin Choi, and Kyunghyun Cho. NaturalProofs: Mathematical theorem proving in natural language.

[58] Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. TacticZero: Learning to prove theorems from scratch with deep reinforcement learning. *Advances in Neural Information Processing Systems*, 34:9330–9342, 2021.

[59] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *Advances in Neural Information Processing Systems*, 35:32353–32368, 2022.

[60] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning*, pages 6984–6994. PMLR, 2019.

[61] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[62] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. miniF2F: a cross-system benchmark for formal Olympiad-level mathematics. In *International Conference on Learning Representations*, 2022. URL: https://openreview.net/forum?id=9ZPegFuFTFv.

[63] Daniel M Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593*, 2019.



This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Egenolff-Berner Garamond, a revival of Claude Garamont's humanist typeface. The above illustration, "Science Experiment 02", was created by Ben Schlitter and released under CC BY-NC-ND 3.0. A template that can be used to format a PhD thesis with this look and feel has been released under the permissive MIT (X11) license, and can be found online at github.com/suchow/Dissertate or from its author, Jordan Suchow, at suchow@post.harvard.edu.