Learning-Assisted Reasoning within Proof Assistants via Symbolic, Statistical, and Neural Guidance

cumulative dissertation

by

Liao Zhang

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of Doctor of Philosophy

advisors: Prof. Dr. Cezary Kaliszyk

Innsbruck, 10 February 2025

universität innsbruck

cumulative dissertation

Learning-Assisted Reasoning within Proof Assistants via Symbolic, Statistical, and Neural Guidance

Liao Zhang (1415002)

10 February 2025

advisors: Prof. Dr. Cezary Kaliszyk

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

February 10th, 2025	Liao Zhang
Datum	Unterschrift

Abstract

Proof assistants, developed within the field of Interactive Theorem Proving (ITP), provide mathematicians with tools to construct proofs that can be rigorously verified by computers. This capability is particularly valuable given the extreme complexity of contemporary mathematics. However, the manual development of proofs in ITP systems is significantly more time-intensive compared to traditional handwritten proofs.

One promising approach to mitigating these difficulties involves the application of machine learning techniques to automate proof construction in ITP systems. Despite recent breakthroughs in machine learning across various fields, the domain of mathematical reasoning still presents formidable challenges.

This dissertation aims to leverage a range of machine learning methods to enhance proof automation within ITP environments.

To address this goal, my research centers on devising innovative feature representations and implementing diverse learning models specifically designed to enhance the prediction accuracy of proof steps within ITP. Furthermore, I investigate a new task, referred to as proof transformation learning, which is highly relevant to proof automation. Instead of predicting the next proof step solely relying on the proof state before its application, proof transformation learning utilizes both the proof states before and after the proof step to make predictions. I also explore the first application of inductive logic programming in deriving rules that predict when specific proof steps are appropriate within the context of ITP.

Acknowledgements

First, I want to express my gratitude to my supervisors. Cezary Kaliszyk supervised my PhD study at the University of Innsbruck, while Josef Urban co-supervised my research and employed me for one year and three months at the Czech Technical University. I learned effective time management from Cezary, and Josef's deep research insights guided me to discover my true research interests.

I have aspired to work with both of my supervisors since my master's study. At Kyoto University, my master's supervisor, Atsushi Igarashi, advised me to explore conference papers to uncover my genuine research interests. Through extensive reading, I realized that my passion for research did not lie in formalizing mathematics but in building proof automation systems. The first paper that attracted me was about Coq Hammer, which was developed by Cezary. I was fortunate to become Cezary's PhD student and was partly supported by his funding for the development of Coq Hammer. To demonstrate my passion, I explained CoqHammer to researchers at Kyoto University and gave a presentation on machine learning for theorem proving in the group seminar. I also studied Cezary's lecture slides on *Artificial Intelligence for Theorem Proving*, the only course I found related to machine learning for this field at the time.

I faced significant challenges during my master's research, but fortunately, I obtained support from Lasse Blaauwbroek, a PhD student of Josef Urban. I chose to develop a novel proof automation approach for Coq as my master's thesis. However, Coq's source code is intensively complicated. I made little progress for three months. Some researchers from the Coq development team advised me to contact Lasse, who had implemented a hook for data extraction in Coq. Thankfully, Lasse allowed me to use his code, which enabled me to successfully complete my master's thesis.

After completing my master's degree, I determined to pursue a PhD in Europe, as very few researchers worked on machine learning for theorem proving in Japan. Staying in Japan would limit my opportunities for mentorship. Moreover, Prague and Innsbruck are global centers for research in this area.

After expressing my thanks to my supervisors, I extend my thanks to my collaborators. Lasse Blaauwbroek guided me through Coq's source code and the usage of Coq Tactician. David M. Cerna aided my understanding of inductive logic programming. I learned advanced knowledge of term rewriting from Fabian Mitterwallner. Bartosz Piotrowski instructed me in the usage of various decision tree algorithms. Jan Jakubuv helped me understand Grackle. Xiyu Zhai introduced me to the theoretical machine learning community.

I would also like to thank those with whom I engaged in insightful discussions during my PhD, which broadened my research perspective and shaped my academic taste. These individuals include Thibault Gauthier, Miroslav Olšák, Chad E Brown, Adam Pease, Jan

Hůla, Jelle Piepenbrock, Martin Suda, Zarathustra Goertzel, Kasper Hagens, Stanisław Purgał, Filip Bártek, René Thiemann, and Aart Middeldorp.

Finally, I am deeply grateful to my parents and friends for their unwavering support over the past four years.

Contents

Chapter 1

Introduction

This dissertation investigates the application of various machine learning techniques to facilitate the construction of computer-verifiable proofs.

Interactive Theorem Proving (ITP) refers to the creation of computer-verifiable proofs with guidance from mathematicians. Interactive theorem provers, also known as proof assistants, play a crucial role in constructing ITP proofs, particularly as modern mathematics becomes increasingly complex, making manual verification an exceedingly challenging task.

A notable example is the ABC conjecture, for which mathematician Shinichi Mochizuki proposed a proof in 2012 [Moc12a, Moc12b, Moc12c, Moc12d]. Due to its significant complexity, the proof remains unverified by the broader mathematical community. If this proof had been constructed using ITP, computers could have automatically verified it, reducing the necessity for extended scrutiny by human experts.

Despite these advantages, constructing ITP proofs is substantially more labor-intensive than traditional, manual methods. This challenge has led researchers to investigate how machine learning can streamline the proof construction process and reduce the overall effort required.

Despite extensive research efforts focused on enhancing proof automation, it remains a persistent challenge for contemporary machine learning methodologies due to two key factors:

1. The intrinsic complexity of mathematics, one of the most rigorously studied disciplines over millennia, poses significant obstacles to computational techniques.

2. Current machine learning models exhibit limitations in reasoning tasks [MIB+24], while mathematical theorem proving necessitates sophisticated reasoning abilities.

This dissertation seeks to advance the field of proof automation, driven by the dual motivations of the mathematical community and the growing interest within the machine learning domain. Mathematicians are in need of more robust proof automation tools to expedite the process of formal proof construction, whereas machine learning researchers are increasingly drawn to the challenges presented by proof automation, which may catalyze the development of more advanced machine learning algorithms.

I selected the Coq proof assistant [The19] as the platform to explore advancements in proof automation. To this end, I have crafted novel features to better capture mathematical representations and examined suitable learning models for improving

proof automation [ZBP+21]. Additionally, I have introduced a new task closely aligned with proof automation and explored its potential applications [ZBKU23]. While most current AI research leverages statistical learning models and neural networks for proof automation, seldom applying symbolic approaches, I complete the first work of applying inductive logic programming [CD22] to derive rules that explain when specific proof steps should be executed [ZCK24]. Furthermore, I contribute to the domain known as term rewriting [BN98], which is tightly connected to ITP. Specifically, my work focuses on confluence analysis, an essential question in term rewriting. Here, I have automatically invented novel strategies for the state-of-the-art confluence prover CSI [ZFM11]. To support the application of modern machine learning methods, I also construct an extensive dataset for confluence analysis, as the existing dataset, COPS [cop], is too limited in size. When CSI is equipped with these newly developed strategies, it achieves higher success rates in (dis)proving confluence in both the COPS dataset and the expanded dataset, surpassing results obtained with CSI's default strategy.

1.1 Interactive Theorem Proving

Numerous proof assistants exist for constructing computer-verifiable proofs, including Lean [dMKA+15], Coq [The19], Isabelle/HOL [NWP02], Mizar [BBG+15], HOL4 [SN08a], and HOL Light [Har09].

ITP has seen extensive application in the formalization of mathematical proofs and the development of dependable software systems. Notably, Gonthier employed Coq to formalize the proof of the four-color theorem [G+08], while a combination of HOL Light and Isabelle was utilized in the formal verification of the Kepler conjecture [HAB+17]. Additionally, Isabelle/HOL was instrumental in the design of a verified kernel for an operating system [KEH+09], and Leroy used Coq to achieve formal verification of a realistic compiler [Ler09].

The correctness of proofs generated by ITP tools is ensured by the logical foundations they are built upon. Different proof assistants employ distinct logical frameworks: Isabelle/HOL, HOL4, and HOL Light are based on higher-order logic, while Coq and Lean rely on the calculus of inductive constructions (CIC) [PM15]. Mizar, on the other hand, is founded on set theory. Each logical foundation presents unique strengths and limitations, leading to uncertainty about the most suitable foundation for ITP applications [HUW14].

The proof languages used in ITP systems can be classified as either declarative or procedural. Declarative languages, such as Isar [Wen99] and Mizar [UB07], characterized by their use of English-like syntax, aim to enhance human readability by making proofs more intuitive. Conversely, procedural languages require the user to construct proofs through programmatic commands known as *tactics*. A tactic may represent a combination of basic logical operations, a specific decision procedure, or the synthesis of more elementary tactics. Declarative languages are easier for mathematicians to understand because they use English-like syntax. However, they tend to be more verbose and give you less control over the proof process than procedural languages. Both styles are prevalent in current

proof assistants: Mizar and Isabelle/HOL favor the declarative style, while Coq and Lean adhere to the procedural approach.

Introduction to Coq This dissertation centers on Coq as the platform for advancing proof automation, motivated by two primary considerations. First, the machine learning research community focused on Coq is quite active. Second, Coq remains widely used, particularly in software verification, meaning that improvements in proof automation for Coq have the potential to benefit a significant number of users.

Coq's logical foundation, CIC, is a variant of type theory [Pie02]. In type theory, terms are often formulated using lambda calculus, and each term is accompanied with a type [Pie02]. Simply typed lambda calculus represents the most fundamental form of type theory [Pie02]. Depending on how typing rules are extended, various other type theories emerge, including dependent type theory [SU06], gradual type theory [NLA19], and homotopy type theory [Pro13].

In functional programming languages, lambda calculus models computation, akin to code execution, while typing rules enforce the procedure of type checking before the execution. Type theory is intrinsically linked to logic satisfiability through the Curry-Howard correspondence [SU06]. This establishes that, within a given type system and logical foundation, the type T of a term is inhabited if and only if the proposition T is logically valid.

In the context of Coq, the Curry-Howard correspondence indicates that a proof is represented as a proof object, which constitutes a term within the logical framework. Consequently, the verification of a proof's correctness is equivalent to assessing the type correctness of its associated proof object.

Figure 1.1 illustrates a concrete Coq proof demonstrating the associative property of addition. Natural numbers are defined as an *inductive definition*, where the constructor `O` represents the number 0. The second constructor, `S`, takes a natural number n as its argument, and `S n` denotes $n + 1$. Coq's underlying logical system, CIC, extends the traditional type theory by incorporating inductive definitions, enabling the specification of natural numbers in this manner. The fixed-point function `plus` defines the addition operation of two numbers n and m. When $n = 0$, `plus` directly returns m. For $n > 0$, where n can be expressed as $n' + 1$, the function `plus` computes `plus n' m` recursively and increments the result by one. The operator `+` is introduced as a shorthand notation for `plus`.

The initial proof state corresponds directly to the theorem's statement. A proof state consists of a goal, displayed below the line, and a set of hypotheses, listed above the line. As explained early in this section, Coq employs tactics to construct proofs. First, we apply mathematical induction on the natural number n using the `induction n` tactic, which generates two subgoals. Applying the `induction n` tactic actually constructs a new proof object with two holes to fill. According to the Curry-Howard correspondence, filling a hole with a proof term is equal to proving a subgoal using tactics. The first subgoal involves proving the equation for the base case where $n = 0$. To resolve it, we first apply the tactic `intros` to instantiate the universal variables `m` and `p`. Afterwards, we prove

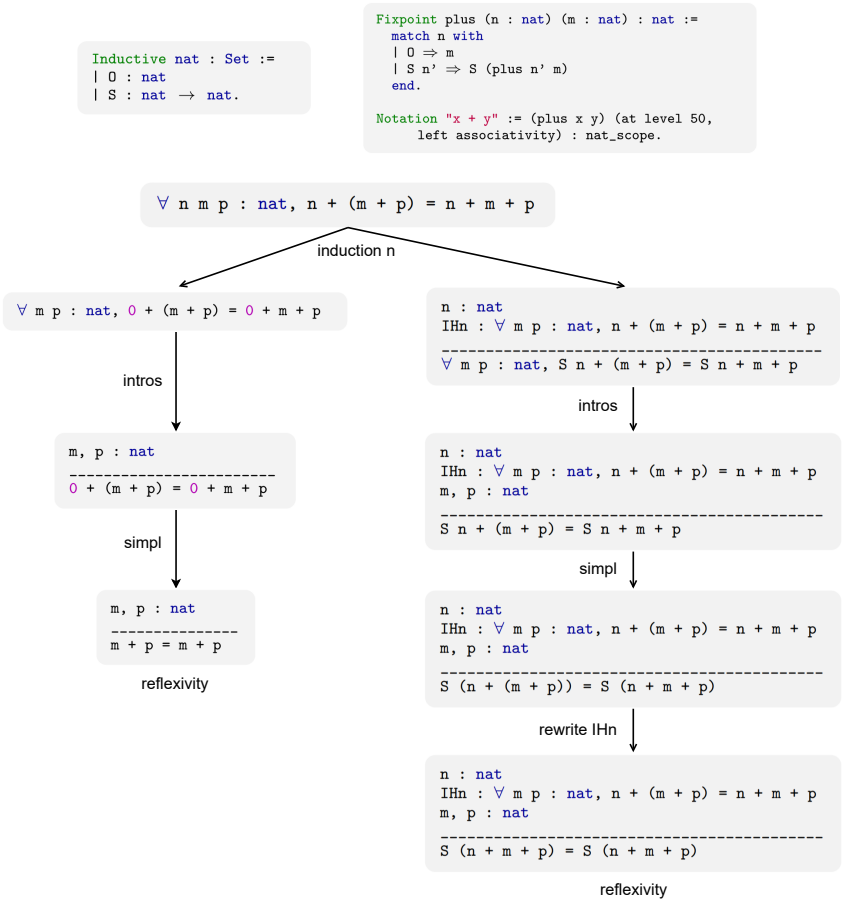


Figure 1.1: An example of a Coq proof of the associative property of addition.

the subgoal by applying the `simpl` tactic for simplification followed by the `reflexivity` tactic to prove the satisfiability of the equation. For the second subgoal, we also first perform instantiation and simplification. Next, we rewrite the goal using the induction hypothesis, denoted `IHn`, and subsequently conclude the proof with `reflexivity`. Once all subgoals are addressed, the proof is complete.

1.2 Automated Theorem Proving

Automated Theorem Proving (ATP) is a field closely associated with ITP. While ITP combines human guidance with automated techniques to construct proofs, ATP seeks to autonomously find proofs for given goals without human involvement.

ATP faces several significant challenges. First, the satisfiability of first- or higher-order

logical formulas is undecidable [Fit12]. Second, logically valid statements may lack a formal proof [Göd92]. Although it is theoretically feasible to implement a program that systematically explores all possible proofs by incrementally increasing their size, the computational time required for such an exhaustive search is prohibitive. In practice, state-of-the-art ATP systems address these limitations by leveraging heuristic algorithms to efficiently navigate the search space for proofs.

Tableaux provers and *saturation provers* stand for two representative categories of ATP provers, distinguished by their underlying proof systems to search for proofs. Prominent examples of saturation provers include E Prover [Sch13] and Vampire [RV99], while notable tableaux provers include Lash [BK22], linTAP [MO99], and Satallax [Bro12].

The *hammer* approach leverages ATP to automate the proof generation within ITP systems [BKPU16a]. Hammers first performs *premise selection*, which selects a set of premises that seem relevant to prove the given theorem. Premise selection reduces the size of the premise search space for ATP. Next, hammers convert the current proof state in ITP into a format that ATP systems can interpret. The conversion is necessary as most ITPs are based on higher-order logic or more expressive foundations. However, most ATPs aim to solve problems in first-order logic. Subsequently, ATPs are employed to search for a solution. If a valid proof is identified by the ATP, the hammer mechanism reconstructs it into a form acceptable within the ITP environment. Researchers have developed hammers for a variety of ITP systems, including Sledgehammer [BBP13] for Isabelle/HOL and CoqHammer [CK18a].

1.3 Learning Models for Proof Automation

Various machine learning techniques have been investigated for proof automation tasks.

k-Nearest Neighbors The k-nearest neighbors (k-NN) algorithm serves as a fundamental technique in statistical learning [Pet09]. Assume a database $\{(x_i, y_i)\}$, where x_i is an example, and y_i is x_i's label. To query the label of a given example x, k-NN first employs a distance function to find x's k-closest examples $\langle (x'_i, y'_i) \rangle$ from the database where $j = 1, \ldots, k$. Next, the label for x is determined by the most frequent label that occurs in y'_i.

In the context of proof automation, it is common to only calculate the k-closest examples, thereby yielding a simplified variant of the k-NN algorithm without computing the most frequent label in the closest examples. The process for tactic prediction using k-NN typically involves the following steps:

1. The distance d_i is computed between the current proof state ps and each proof state ps_i stored within the database $\{(ps_i, tac_i)\}$. This distance d_i is derived by first transforming the proof states into feature representations, followed by the application of a distance metric to assess the similarity. Commonly utilized distance functions include Jaccard distance [Jac01] and Euclidean distance.

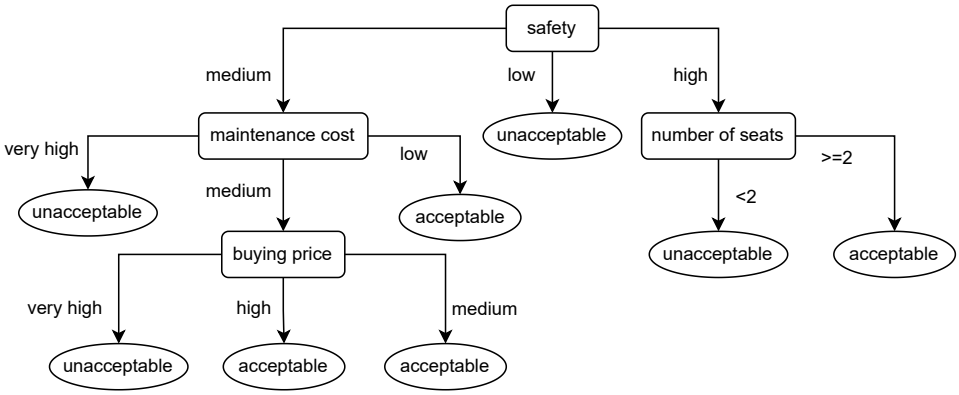


Figure 1.2: An example of a learned decision tree for determining whether buying a car is acceptable.

2. The proof states, along with their corresponding tactics in the database $\{(ps_i, tac_i)\}$, are then ranked based on the computed distances d_i.

3. The tactics tac_i associated with the nearest proof states are subsequently returned.

Decision Trees Decision tree algorithms generate predictive rules and represent these rules as hierarchical tree structures. Figure 1.2 illustrates a decision tree that has been derived from data to determine the acceptability of purchasing a car. In this context, a leaf node signifies a classification label, while the path leading to a leaf node embodies the conjunction of features that yield that label. For example, the leftmost path in Figure 1.2 indicates that a car is deemed unacceptable if it possesses medium safety ratings and a very high maintenance cost.

The original decision tree algorithm is susceptible to overfitting and typically exhibits inferior performance compared to two of its derivative algorithms: random forests and gradient boosted trees (GBT). A comparative analysis of these three algorithms is depicted in Figure 1.3. The original algorithm is trained on the entire training dataset and constructs a single tree for prediction purposes. In contrast, the random forest algorithm generates multiple trees, each trained on distinct subsets of the training data. The collective prediction of the forest is determined through a voting mechanism, wherein the class selected by the majority of trees is adopted. Each tree within the random forest captures a subset of the dataset, reducing the likelihood of overfitting to the training data. The ensemble of multiple trees generates a more generalized model that effectively represents the overall dataset.

In the GBT algorithm, each layer comprises a decision tree. During training, a newly added tree is specifically designed to mitigate the errors incurred by the preceding layers, subsequently augmenting the existing ensemble of trees. Let us denote an imperfect model F_m that comprises m trees from the earlier layers, where x represents the input examples

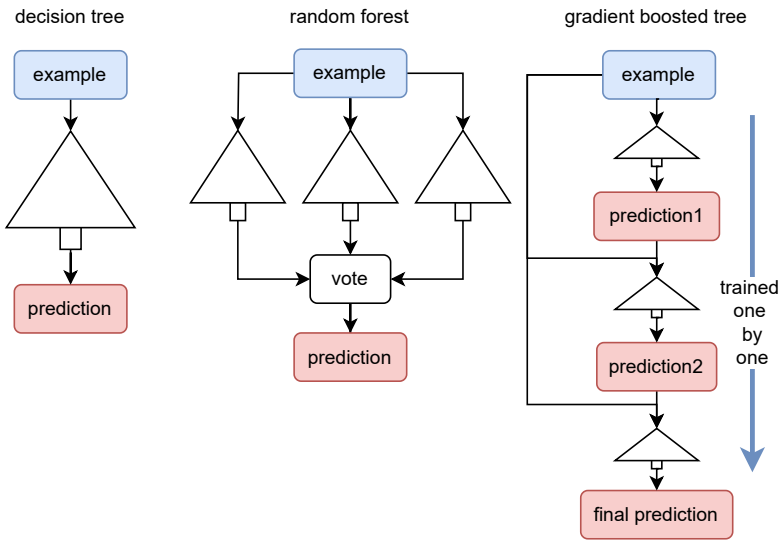


Figure 1.3: Comparison of different decision tree algorithms.

and y denotes the corresponding labels in the dataset. To enhance the model's accuracy, GBT seeks to sequentially learn a new tree $h(m)$ such that $F_{m+1}(x) = F_m(x) + h_m(x)$, aiming to minimize the error represented by $y - F_m(x)$. The output of the final layer is returned as the prediction; hence, all previously trained trees are sequentially utilized in the prediction. GBT is less prone to overfitting compared to a single decision tree. By iteratively minimizing the errors of previous trees, each new tree focuses on fitting a subset of the data. Subsequently, all trees contribute jointly to the final predictions, thereby minimizing the risk of dependence on any single tree that may only be fitted to examples with a similar underlying pattern.

Among the three decision tree algorithms, GBT demonstrates superior practical performance [BCMM21]. However, training a GBT model is slow due to the inherently sequential nature of the tree-building process. To handle the problem, several efficient GBT libraries are implemented, such as XGBoost [CG16], CatBoost [HK20], and LightGBM [KMF+17].

Neural Networks Neural networks have become the leading methodology in machine learning, following the success of AlexNet [KSH12], which achieved top performance on the ImageNet dataset [DDS+09] in 2012. An example of a neural network architecture is shown in Figure 1.4. A neural network consists of artificial neurons, which are mathematical constructs inspired by biological neurons. Each neuron receives inputs from preceding neurons via connections. These inputs, represented as real numbers, are processed by a nonlinear function at each neuron to produce an output. The influence of

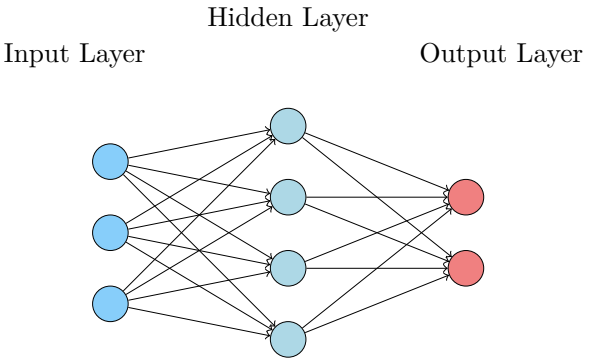


Figure 1.4: An example of an artificial neural network. Each neuron is represented as a node. The arrows between neurons denote the directed connections.

each connection is modulated by weights, which are optimized during the training phase. Artificial neurons are grouped into layers, and a neural network model typically contains multiple such layers. Input data is progressively transformed as it passes through each layer, resulting in a final output at the model's conclusion. The layers between the input layer and the output layer are called hidden layers.

The universal approximation theorem for neural networks has been rigorously established [HSW89]. This theorem demonstrates that for any function f and any specified tolerance level $\epsilon > 0$, there exists a neural network capable of approximating f within an ϵ-level of accuracy. While the theorem indeed affirms the theoretical expressiveness of neural networks, realizing such an approximation may necessitate an impractically large number of neurons.

A significant advantage of neural networks is their capacity to automatically learn implicit features from data. In contrast, traditional statistical learning methods, such as k-NN and decision trees, rely on manually engineered features. This reliance on handcrafted features often leads to challenges in adequately addressing diverse practical scenarios [KSH12]. Furthermore, features that are meticulously designed tend to be closely associated with specific domains, which hampers their generalizability across broader training datasets.

Neural networks necessitate substantial amounts of data and computational resources to effectively learn implicit patterns for prediction tasks. Their success is also attributable to advancements in computational hardware, particularly GPUs, which enhance the efficiency and practicality of the learning process.

Large Language Models Large language models (LLMs), such as ChatGPT [AAA+23], Qwen [BBC+23], and Gemini [TAB+23], have achieved remarkable progress in recent years. These models exhibit fluency in conversational interactions and provide recommendations in various domains, including software implementation, proofreading, language translation, and responses to general inquiries.

Language models are fundamentally probabilistic models of natural language [Jur00].

A primary training task involves predicting the probability distribution of the subsequent word based on a preceding sequence. Language models are extensively employed in research areas such as speech recognition, machine translation, and information retrieval [JM24].

LLMs are typically pre-trained on extensive general corpora and subsequently fine-tuned for specific applications. This pre-training process is inspired by the human capacity for transfer learning, wherein knowledge from analogous tasks can facilitate the rapid acquisition of new skills. To support this, major companies have amassed large datasets from diverse sources, including the Internet, books, code repositories, and academic publications. For instance, GPT-2 is pre-trained on 40 GB of text [RWC+19], whereas the larger GPT-3 model utilizes 45 TB of text [BMR+20].

LLMs are characterized by a vast number of parameters and are pre-trained on large datasets. For example, the GPT-4 model contains approximately 1.8 trillion parameters, whereas ResNet-50 v1.5 [HZRS16], a widely used neural network for image classification, has only 25.6 million parameters. Although the extensive parameterization contributes to superior performance across various domains, it also results in slower inference times, which poses challenges for proof search in formal mathematics.

LLMs are often built upon the transformer architecture [VSP+17]. Transformers incorporate a self-attention mechanism that facilitates global computation across all input tokens. The efficacy of transformer-based models is contingent upon the size of the training dataset and the computational resources utilized, a relationship described by the scaling law [KMH+20]. According to this principle, the training of a robust LLM requires a considerable volume of data; however, the availability of high-quality datasets for formal mathematics is significantly lower than that for natural language.

1.4 Learning Tasks for Theorem Proving

There are several research directions in machine learning for theorem proving. The primary directions are tactic-based ITP guidance, premise selection, and autoformalization.

1.4.1 Tactic-based ITP Guidance

Tactic-based ITP guidance aims to automatically construct a proof via the applications of tactics. Current tactic-based ITP guidance approaches generally comprise the following components. First, an AI model predicts a sequence of tactics tailored to the initial proof state, which is the root of the proof tree. Subsequently, these predicted tactics are applied to the current proof state, closing it or generating a new proof tree. In each new proof tree, the tactic application derives new open proof states as the children of the original proof state. The AI model again generates tactic predictions for the open proof state in the unfinished proof tree, and the corresponding tactics are applied. This proof search continues until constructing a proof tree such that all its proof states are closed.

Empirical analyses indicate that tactic-based ITP guidance demonstrates superior performance compared to traditional hammers [GKU+21a, YD19]. One potential explanation for this advantage is its capacity of utilizing a diverse array of beneficial tactics.

Some tactics encode useful combinations of logical operations that are challenging for ATP solvers to identify. For example, mathematical induction is notably difficult to implement in ATP [BM14], yet can be easily invoked through induction tactics. Additionally, tactic-based ITP guidance can leverage user-defined, domain-specific tactics, such as those within the Coq Iris project, which encompasses a range of tactics specifically designed for addressing problems in separation logic [JKJ+18]. As hammers are also defined as tactics, this approach can also determine when to appropriately execute them. The observed improvement over hammers may also be attributed to the direct and efficient tactic-level proof search. Hammers need to convert ITP proof states to ATP proof states and to reconstruct ITP proofs from ATP proofs. In contrast, tactic-based ITP guidance directly executes tactics within the ITP environment, thereby eliminating the time required for proof state transformation and reconstruction.

There are two categories of tactic-based ITP guidance systems. The first category aims at developing a user-friendly guidance interface, often employing quick but practically strong learning algorithms. The representative systems are TacticToe [GKU+21a] for HOL4 and Tactician [BUG20b] for Coq. The second category seeks to examine the abilities of state-of-the-art learning techniques in theorem proving. Such systems employ heavy computation, caring less about convenient usages for mathematicians. Some representations are GPT-f [H+21] and Lean Dojo [Y+23].

Challenges in the Development The development of an effective tactic-based ITP guidance presents several challenges.

- The tactic space is extensive, mainly caused by the following two reasons. First, tactics in ITP can accommodate various categories of arguments, resulting in a potentially infinite array of combinations. For instance, tactics in Coq accept different types of arguments, including names of local hypotheses, previous definitions, and established theorems. Second, some ITP systems permit users to create custom tactics using their respective proof languages. In Coq, user-defined tactics are formulated through its versatile tactic languages, Ltac [Del00] and Ltac2 [Ped19]. Users of Coq can devise heuristics to amalgamate multiple tactics into a more robust automated tactic, exemplified by the tactic `crush` [Chl13]. Furthermore, Coq users can establish domain-specific tactics aimed at addressing particular problem types. A notable example of this is the Coq's standard library. It encompasses various tactics developed by Coq experts that are particularly effective for proving theorems grounded in different domains, including real number arithmetic, integer arithmetic, and list operations.

- Multiple tactics may be suitable for the same proof state. In Figure 1.1, the tactic `induction n` can be replaced with `induction n as [|H1]`, with only minor adjustments to the subsequent tactics. In this context, `induction n as [|H1]` not only facilitates mathematical induction but also designates the names of the newly generated hypotheses. Within ITP libraries, numerous instances arise where optimal tactics conflict for specific categories of proof states. The determination of

the most appropriate tactic necessitates an AI's comprehension of the underlying intent behind the application of the tactic.

- There exists a significantly imbalanced distribution of tactics within ITP libraries. Certain tactics, such as the automation tactic `auto` in Coq, are prevalent in the dataset and conduct basic automation in proof construction. Conversely, domain-specific tactics, while infrequently encountered, hold considerable significance for their respective areas. It is exemplified by the `ring` tactic, which is only essential to theorems related to group theory. The infrequent occurrence of some tactics may also be attributed to their requirement for specific arguments, such as the names of lemmas. For instance, the tactic `apply Lemma1` applies `Lemma1` to the current proof state. A particular lemma may be critical for establishing a certain group of theorems but may hold no utility for others.

- The implementation of ITP systems presents considerable complexities, which hinder the development of anticipated modifications. Several factors contribute to these challenges. First, the theoretical underpinnings of ITP systems are inherently complex, often based on advanced logical frameworks such as higher-order logic and dependent type theory [Chl13]. Second, the lack of systematic documentation within ITP systems further complicates the understanding of the source code. This absence of documentation may be due to the fact that ITP systems are typically developed by small research groups in the academic community, in contrast to industrial-grade software, which is generally produced through the collaborative efforts of larger teams with strict guidelines for creating comprehensible documentation.

  The implementation of Coq especially presents several complexities for the following reasons. First, Coq is founded upon a robust and intricate logical framework known as CIC, whose implementation is inherently more complicated than higher-order logic employed by systems such as HOL4 and Isabelle/HOL. Second, Coq has been under continuous development for over 30 years since its initial release in 1989 [Ber08]. Throughout this period, the Coq development team has introduced a significant array of features and modified the codebase with contributions from various individuals, resulting in challenges related to deciphering the original intentions behind certain code segments. Third, the source code of Coq is highly optimized, necessitating advanced skills in functional programming to comprehend the implementation techniques employed. For instance, the data structure of the fundamental terms in Coq is constructed using the methodologies detailed in the paper [Swi08].

Datasets of ITP Systems There exist several ITP datasets for the different ITP systems that can be used for training machine learning models.

The most commonly used datasets are standard libraries of ITP systems. A standard library comprises the frequently reused definitions and theorems for proving new theorems using the underlying ITP system. Two typical examples are Coq's standard library, used for the evaluation of Tactician and CoqHammer, and HOL4's standard library for testing

TacticToe. Although the standard libraries contain dedicatedly designed proofs, their sizes are usually limited for machine learning. Meanwhile, its scope is quite limited as it often merely contains the most reusable proofs.

Some proof assistants' development teams maintain large datasets for their proof assistants. Isabelle perhaps has the largest formal library among all ITP systems, which is called the Archive of Formal Proofs (AFP, https://www.isa-afp.org/). It has been developed for more than 20 years and includes a collection of proofs in scientific papers, proof libraries, and examples. Now, it is approximately composed of 4,565,700 lines of code and 279,600 lemmas, contributed by 513 distinct authors. Similar to Isabelle, Lean's development team also organizes the construction of a large dataset called mathlib. It consists of a significant number of libraries for a variety of mathematical domains. Although Coq is one of the most popular proof assistants, Coq does not officially provide any large dataset. Some Coq researchers collect Coq libraries and publish their own datasets exemplified by CoqGym [YD19] and Graph2Tac's dataset [Bla23]. However, such datasets tend to be less maintained and can only be executed on specific Coq versions.

1.4.2 Premise Selection

The task of premise selection is formally defined as

Definition 1.1 (Premise selection problem)**.** Given a substantial collection of premises P, an ATP system A operating within specified resource constraints, and a new conjecture C, the objective is to identify the subset of premises from P that are most likely to be utilized by A in automatically constructing a proof for C.

Premise selection is particularly helpful in enhancing ATP solvers' performance since they have an intensive number of premises to choose from in the proof search. As explained in Section 1.2, premise selection is also a standard procedure in hammers and can significantly enhance their efficiency [BKPU16b, CK18b, GK15a].

Premise selection represents one of the most extensively investigated tasks in the realm of proof automation. Initial research applies k-NN to discover the lemmas that are similar to the current theorem according to the distances between feature characterizations [KU13]. Several kinds of decision tree methods have also been tried for this task. Färber [FK15a] first tries to apply random forests to selecting relevant premises for Mizar. Later, the random forest premise selection approach is extended to Lean [PMA23]. Most recent works determine to build gradient boosted tree models due to the outstanding efficiency of their modern implementations. Both XGBoost [PU18, JU17, J$^+$20] and LightGBM [GJK$^+$22, GCJ$^+$21] have been investigated for this task. The first application of neural networks to premise selection is conducted by DeepMath [ISA$^+$16]. Despite it requiring significantly more data than simpler methods, the performance merely achieves limited improvements. After DeepMath, different categories of neural networks have also been explored, such as graph neural networks [OKU20], recurrent neural networks [PU20], and transformers [MAT$^+$23].

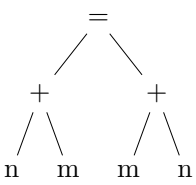


Figure 1.5: The AST of the logical formula $n + m = m + n$.

1.4.3 Autoformalization

Autoformalization focuses on the conversion between formal proofs and natural language. This area has garnered increasing interest due to the scarcity of high-quality formal proofs. State-of-the-art machine learning techniques necessitate a substantial volume of data; however, the quantity of available formal proofs is considerably limited in comparison to the breadth of modern mathematical knowledge. Additionally, the manual development of formal proofs for contemporary mathematics poses significant challenges and is labor-intensive. Despite its potential, autoformalization is particularly difficult due to the necessity for a profound comprehension of both rigorous formal language and the inherent ambiguities of natural language.

Several research works have been conducted for autoformalization. Kaliszyk [KUV17a] first investigates autoformalization by building parsing trees that parse natural language theorems to formal theorems. Wang [WKU18, WBKU20a] first applies neural networks to autoformalization on synthetic data. Recently, LLMs have also been applied to this task [WJL+22].

1.5 Characterization of Mathematical Representation

Machine learning algorithms rely on precise characterization to obtain reasonable performance. Statistical learning requires feature characterization, while neural networks use various embeddings.

1.5.1 Feature Characterization

Syntactic Features capture essential information from the syntactic structures of mathematical representations. A common method for feature extraction involves traversing the abstract syntax tree (AST) of the proof state. For example, the AST of the logical formula $n + m = m + n$ is depicted in Figure 1.5. A typical feature characterization that includes tree walks up to length two would yield features such as $\{n, m, =, = -+, + - n, + - m\}$. Here, n, m, and $=$ represent length-1 walks in the AST, while $= -+, + - n, + - m$ represent length-2 walks.

Numerous sophisticated syntactic features have been developed, such as those employed in the learning-guided ATP prover Enigma [JU17]. Enigma uses anonymous features that abstract the nodes in the AST by replacing them with their abstract identifiers [J+20], allowing for better generalization. This is critical due to the vast number of unique

theorems, constants, and definitions in a theorem-proving library. Another method incorporates feature occurrence frequencies [CJSU19a], and the term frequency-inverse document frequency algorithm has been applied to weigh features based on their frequency of occurrence [BUG20a].

Semantic Features aim to capture deeper insights beyond basic syntactic structures. Syntactic features can sometimes fail to distinguish between semantically different expressions. For example, ϕ and $\neg\phi$ are syntactically similar but differ significantly in meaning.

To address this, one approach is to include generalizations of terms as additional features [KUV15b]. This is motivated by the observation that logical formulas proven with the same technique are often instances of the same generalization, as demonstrated by instances of the same theorem.

In this dissertation, we explore two algorithms—anti-unification and structural tree difference—to extract more semantic information. Anti-unification identifies the least general generalization (lgg) of two terms [CK23]. We specifically use first-order syntactic anti-unification [CK23], which identifies the least general generalization (lgg) of two first-order terms and the relevant substitutions. First-order terms use a signature $\mathcal{F}$, representing a set of function symbols, where each function symbol $f \in \mathcal{F}$ has a defined arity $n \in \mathbb{N}$, with $\mathbb{N}$ denoting the set of natural numbers. Function symbols with zero arity are constants. Additionally, a set $\mathcal{V}$ contains variable symbols, which are disjoint from the symbols in $\mathcal{F}$.

Definition 1.2. The set of *terms* $\mathcal{T}(\mathcal{F}, \mathcal{V})$ over a signature $\mathcal{F}$ and a set of variables $\mathcal{V}$ is inductively defined by the following two rules:

- if $x \in \mathcal{V}$, then $x \in \mathcal{T}(\mathcal{F}, \mathcal{V})$.
- if $f \in \mathcal{F}$ with arity n and $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ for $1 \leq i \leq n$, then $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$.

Definition 1.3. A *substitution* is a mapping σ from $\mathcal{V}$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$. The *application of the substitution σ to the term t* is defined as:

$$t\sigma = \begin{cases} \sigma(t) & \text{if } t \in \mathcal{V} \\ f(t_1\sigma, \ldots, t_n\sigma) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

Definition 1.4. A term u is called a *generalization* of s and t if there exist substitutions σ_1 and σ_2 such that $u\sigma_1 = s \wedge u\sigma_2 = t$.

Definition 1.5. A generalization u' of s and t is called the *lgg* if, for any generalization u of s and t, there is a substitution σ, such that $u'\sigma = u$.

For instance, for the terms $s = f(a, g(b), a)$ and $t = f(b, a, b)$, first-order syntactic anti-unification computes the lgg as $f(x, y, x)$. Furthermore, it derives the substitutions $\{x \to a, y \to g(b)\}$ and $\{x \to b, y \to a\}$ from $\phi(s, t)$, allowing the reconstruction of s and t from the lgg.

Anti-unification can be extended to include various equational theories, leading to different algorithms such as AC anti-unification, which integrates anti-unification with associative and commutative theories [AEEM14]. However, incorporating expressive theories increases computational complexity, making some applications more challenging [YD21].

Structural tree difference [MS19], on the other hand, is concerned with identifying the differences between two tree-structured data. This method is often used to compare different versions of code libraries, but in this dissertation, it is applied to analyze the differences between the ASTs of proof states.

1.5.2 Neural Network Embeddings

Neural networks are capable of encoding and processing mathematical knowledge. To leverage language models, natural language text is segmented into tokens, which are subsequently input into these models. Furthermore, mathematical knowledge can be represented as tree structures or graph structures, serving as inputs for tree neural networks and graph neural networks, respectively.

1.6 Inductive Logic Programming

Inductive Logic Programming (ILP) is categorized within symbolic machine learning approaches. A longstanding discourse exists between symbolic AI and connectionist AI. Symbolic AI aims to derive explicit rules that elucidate the rationale behind specific predictions. In contrast, connectionist AI, exemplified by neural networks, is inspired by biological neurons, where knowledge is represented as implicit weights within the model, rendering it less interpretable. Initially, symbolic AI was the predominant focus of AI research. However, the rise of connectionist AI and statistical learning later emerged as compelling alternatives, primarily due to the inherent challenges in formulating explicit rules to model diverse conditions. A case in point illustrating the limitations of symbolic AI can be found in the domain of machine translation, where representing grammar through rules proves impractical owing to the inherent ambiguities of natural language. Beyond explainability, symbolic AI excels in modeling knowledge within first- or higher-order logic, whereas statistical learning and connectionist AI predominantly represent knowledge in propositional formats, which lack the expressiveness necessary for capturing complex relationships.

ILP tasks are typically expressed using logic programs, often written in programming languages such as Prolog. An example illustrating the learning of grandparent relationships using ILP is depicted in Figure 1.6. In this context, the code `mom(a,b)` signifies the ground fact that `a` is the mother of `b`. The rule *gp(X,Y) :- mom(X,Z), mom(Z,Y)* is interpreted as follows: `X` is a grandparent of `Y` if `X` is the mother of `Z`, and `Z` is the mother of `Y`.

The syntax of a logic program is delineated as follows:

BK

```
mom(a, b). mom(a, c).
mom(b, d). dad(e, b).
dad(c, f). dad(e, c).
```

E^+

```
gp(a,d). gp(e,d).
gp(a,f). gp(e,f).
```

E^-

```
gp(a,b). gp(b,c).
gp(c,f). gp(d,f).
```

Rules to learn

```
gp(X,Y) :- mom(X,Z), mom(Z,Y).
gp(X,Y) :- dad(X,Z), dad(Z,Y).
```

```
gp(X,Y) :- mom(X,Z), dad(Z,Y).
gp(X,Y) :- dad(X,Z), mom(Z,Y).
```

Figure 1.6: Example ILP problem: *grandparent*.

- A *variable* is denoted by a string beginning with capital letters, such as `X`, `Y`, and `Z` in Figure 1.6.
- A *predicate* symbol is represented by a string commencing with lowercase letters, such as `mom`, `dad`, and `gp`. The *arity* of a predicate symbol denotes the number of arguments it accepts. For instance, the predicates `mom`, `dad`, and `gp` all exhibit an arity of two.
- A *constant* is a predicate symbol characterized by an arity of zero, exemplified by `a`, `b`, `c`, `d`, `e`, and `f` in the figure.
- A *term* may be a variable, a constant, or an arity n predicate symbol with n terms as its arguments, such as `a`, `X`, `gp(a,d)`, and `mom(X,Z)`.
- A term is considered *ground* if it contains no variables, as seen in `mom(a, b)` and `dad(e, b)`.
- An *atom* is defined as a formula $p(t_1, \ldots, t_n)$, where p is a predicate symbol and $t_1, \ldots, t_n$ are terms.
- A *definite clause* or a rule takes the form $h \coloneqq b_1, \ldots, b_n$, where h and $b_1, \ldots, b_n$ are atoms, as in `gp(X,Y) :- mom(X,Z), mom(Z,Y)`. The atom h is referred to as the *head* of the clause, while atoms b_i represent the body of the clause. The symbol comma denotes a conjunction. A definite clause can be informally interpreted as indicating that the head is valid if all bodies are valid.

Researchers have developed numerous ILP systems designed to learn logical rules. Aleph is arguably the most prominent ILP system [Sri01], demonstrating strong empirical performance and implementing a diverse array of ILP techniques that facilitate learning across various tasks. Aleph is also employed in one of the works presented in this dissertation. Popper adopts a learning-from-failures approach to minimize the search space and is recognized as one of the most robust ILP systems currently available [CM21]. Additionally, there exist ILP systems that integrate the strengths of statistical learning or neural networks, thereby achieving commendable performance in learning from noisy

```
modeh(1, gp(+person,-person)).    modeb(2, mom(+person,-person)).
modeb(2, dad(+person,-person)).
```

Figure 1.7: Example of the mode declaration for the ILP task in Figure 1.6.

data. Two notable systems are δ-ILP [EG18] and Problog [DRKT07]. My work in this dissertation is based on Aleph.

The learning process in Aleph is defined by the background knowledge BK, a set of positive examples E^+, and a set of negative examples E^-. The objective is to induce a hypothesis H such that $\forall e \in E^+, H \cup B \models e$ and $\forall e \in E^-, H \cup B \not\models e$, where $\models$ denotes the logical entailment. A hypothesis may comprise a single clause or multiple clauses. For instance, the four rules for `gp` in Figure 1.6 constitute a hypothesis.

Given the extensive search space for potential predicates and variables, ILP employs various techniques to narrow this space. Typical techniques include mode declarations [Mug95, Ray09] and bottom clauses [CD22], both of which are implemented in Aleph.

Mode declarations specify the potential predicates to be included in a rule and their corresponding argument types, reducing the related rule's search space. A mode declaration in Aleph is typically formulated as:

$$mode(recall, pred(m_1, m_2, \ldots, m_n))$$

Figure 1.7 presents the mode declaration utilized for the ILP task illustrated in Figure 1.6. Rather than employing *mode* directly to define the constraints for the predicate *pred*, one can also use *modeh* and *modeb* to specify the predicates that occur in the head and bodies, respectively. The argument *recall* indicates the maximum number of occurrences of the predicate *pred* within a clause. The arguments m_i represent the type of the ith argument of *pred*. The symbol $+$ preceding m_i indicates that m_i is an *input variable*, while $-$ signifies that m_i is an *output variable*. Assuming the learning of a clause of the form $h := b_1, b_2, \ldots, b_n$, any input variable in the body atom b_i of type T must appear either as an output variable of the same type in a preceding body atom b_j or as an input variable of type T in the head atom. For instance, `modeb(2, mom(+person,-person))` signifies that the predicate `mom` respectively receives an input variable and an output variable, both of which have the type `person`, as its first and second argument. It also exhibits that `mom` at most occurs two times in a clause.

The utilization of the bottom clause constraint [DR08] is a key feature of Aleph. Aleph's proof search procedure shares significant similarities with the early ILP system FOIL [Qui90], though the primary distinction lies in Aleph's incorporation of the bottom clause constraint. The bottom clause represents the most specific clause that generalizes the given example, bounded by the mode declarations. By disregarding clauses that are not generalizations of the bottom clause, Aleph efficiently reduces the search space.

The integration of mode declarations with the bottom clause constraint substantially optimizes the search space of the ILP system presented in Chapter 5.

Algorithm 1 A simplified algorithm illustrating Aleph's procedure for searching a clause.

Input: a set of positive examples E^+, a set of negative examples E^-, and background knowledge B.
Output: a clause new_c such that $\forall e \in E^-, new\_c \cup B \not\models e$.
select $e^+ \in E^+$
generate the bottom clause based on e^+ and the mode declarations
generate an initial clause *init* containing only a head atom
$active\_clause \leftarrow [init]$
$new\_active\_clause \leftarrow []$
for $c \in active\_clause$ **do**
    for $p \in next\_predicate(c)$ **do**
        $c' \leftarrow$ append p to the body of c
        $new\_active\_clause \leftarrow new\_active\_clause + c'$
    sort new_active_clause in descending order based on the costs calculated by the cost function
    for $new\_c \in new\_active\_clause$ **do**
        if new_c rejects all E^- **then**
            return new_c
    $active\_clause \leftarrow new\_active\_clause$
    $new\_active\_clause \leftarrow []$

Aleph's default search procedure for identifying a clause basically consists of the following steps:

1. Select an example from the positive examples to generalize.
2. Construct the bottom clause based on the mode declarations and the selected example.
3. Search for a clause to add to the hypothesis. The newly generated clause must reject all negative examples.
4. If the hypothesis covers all positive examples, the search terminates. Otherwise, Aleph selects another example, removes the last example from the set of positive examples, and generates a new clause.

Algorithm 1 presents a simplified algorithm depicting how Aleph searches for an individual clause, utilizing a beam search approach. To generate longer clauses, Aleph appends every possible predicate to each of the active clauses. The potential next predicates are all constrained by the mode declarations and the bottom clause. The resulting clauses are ranked according to a cost function. For each clause, the default cost function calculates the difference between the number of covered positive examples and the number of covered negative examples. The search terminates once a clause that rejects all negative examples is generated. If no such clause is found, Aleph continues to generate longer clauses until the desired clause is identified.

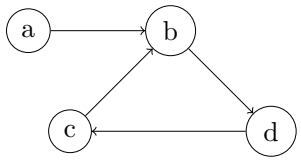


Figure 1.8: An example ARS system.

1.7 Term Rewriting

Rewriting models transformations between objects, representing diverse operations across domains such as symbolic expression simplification in mathematics and program execution in computer science. Term rewriting, which operates under the assumption that objects to be transformed are terms, provides a robust formalism applicable to tasks like simplifying goal states in automated theorem proving [BG94] and optimizing compiler processes [VdBHKO02].

Various types of rewrite systems exist based on the formalization of objects, with the simplest being the abstract rewrite system (ARS).

Definition 1.6. An ARS is a pair $\mathcal{A} = (A, \to)$ of a set A and a binary relation $\to$ on A.

A (possibly infinite) *rewrite sequence* is a sequence $a_0 \to a_1 \to \cdots$ such that $a_i \in A$. We write $a \to^* b$ if there is a rewrite sequence $a \to \cdots \to b$.

Definition 1.7. An ARS $(A, \to)$ is *terminating* if $\forall a \in A$, there are no infinite rewrite sequences starting from a.

The notation $a \downarrow b$ denotes that a and b are *joinable*, meaning that there exists an element $c \in A$ such that $a \to^* c$ and $b \to^* c$.

Definition 1.8. An ARS $(A, \to)$ is *confluent* if $\forall a, b, c \in A$ with $b \leftarrow^* a \to^* c$, we have $b \downarrow c$.

Consider an abstract reduction system (ARS) $\mathcal{E} = (E, \to)$, where $E = \{a, b, c, d\}$ and $\to = \{(a, b), (b, d), (c, b), (d, c)\}$. This system produces the ARS illustrated in Figure 1.8. The ARS $\mathcal{E}$ is non-terminating as it admits an infinite rewrite sequence: $c \to b \to d \to c \to \cdots$.

A key limitation of ARSs is their reliance on concrete instances alone, which restricts their applicability to computations that involve contexts and variables, such as logical inference rules in ATP. In contrast, term rewrite systems (TRSs) extend this capability by incorporating first-order variables and employing first-order terms defined in Definition 1.2.

We then define the notions of rewriting terms using contexts and holes.

Definition 1.9. A *hole* is defined as a special symbol $\square \notin \mathcal{F}$, and a *context* C is a term that contains exactly one hole. The notion $C[t]$ denotes the *application of the term t to the context C*, which is defined as follows:

$$C[t] = \begin{cases} t & \text{if } C = \square \\ f(t_1, \ldots, C'[t], \ldots, t_n) & \text{if } C = f(t_1, \ldots, C', \ldots, t_n) \end{cases}$$

Definition 1.10. The *set of variables in a term* t is defined as

$$Var(t) = \begin{cases} \{t\} & \text{if } t \text{ is a variable} \\ \varnothing & \text{if } t \text{ is a constant} \\ \bigcup_{i=1}^{n} Var(t_i) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

Definition 1.11. A *rewrite rule* for terms l and r is written as $l \to r$ where $l \notin \mathcal{V}$ and $Var(r) \subseteq Var(l)$. A *term rewrite system* $\mathcal{R}$ consists of a set of rewrite rules. Consider the TRS $\mathcal{R}$, we write the *rewrite relation* $t \to_{\mathcal{R}} u$ for terms t, u if there exists a rewrite rule $l \to r \in \mathcal{R}$, a context C, and a substitution σ such that $t = C[l\sigma]$ and $u = C[r\sigma]$.

We write $\to_{\mathcal{R}}^*$ to denote the transitive-reflexive closure of $\to_{\mathcal{R}}$. Similar to ARSs, we obtain the definitions of rewrite sequences and $\downarrow_{\mathcal{R}}$ for TRSs. We drop the subscript $\mathcal{R}$ for the relations on terms in the subsequent introduction if it is contextually inferable.

Research in term rewriting centers on examining various properties of TRSs, each of which requires distinct analytical techniques [BN98]. Notably, a TRS satisfying one property does not imply it satisfies others. Termination and confluence are two critical properties in rewrite systems. They are undecidable for TRSs [BN98]. Given this undecidability, researchers have developed diverse techniques to establish either termination or confluence, with each method suited to a specific subset of related challenges.

1.7.1 Termination Analysis

Termination is an important property of TRSs.

Definition 1.12. A TRS $\mathcal{R}$ is *terminating* if $\forall t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, there is not any infinite rewrite sequence $t \to t_1 \to \cdots$ starting from t.

The TRS $\{f(x) \to g(f(x)), g(y) \to f(g(y))\}$ is not terminating, as it allows the infinite rewrite sequence $f(x) \to g(f(x)) \to f(g(f(x))) \to \cdots$. This sequence is infinite because the term $f(x)$ within $g(f(x))$ can be rewritten back to $g(f(x))$, resulting in an infinite loop.

Given the undecidability, a diverse range of techniques has been developed to either prove or disprove termination for specific categories of TRSs. Some of these techniques involve substantial computational effort to achieve a termination proof. As a result, manually applying many computationally intensive analytical techniques is highly impractical.

Automatic termination provers are therefore designed to execute various analytical techniques autonomously, determining the termination status of TRSs. Among the most notable tools in this domain are AProVE [GSKT06] and T$_{\text{T}}$T$_2$ [KSZM09]. Both systems implement extensive collections of analytical methods, complicating the task of combining these techniques effectively. To maximize efficiency, these tools employ sophisticated scheduling algorithms for the analysis methods.

The Termination Competition (termCOMP) [GMR+15], an annual event, evaluates the effectiveness of termination tools. Originally focused on TRSs, termCOMP has expanded to include the analysis of programs written in various languages such as Haskell, C, and Java. For these programming languages, programs are first translated into corresponding foundations such as rewrite systems, which are then analyzed by termination tools to determine whether they terminate.

1.7.2 Confluence Analysis

Another very important property is confluence.

Definition 1.13. A TRS $\mathcal{R}$ is *confluent* if and only if $\forall s, t, u \in \mathcal{T}(\mathcal{F}, \mathcal{V}), s \to_{\mathcal{R}}^* t \wedge s \to_{\mathcal{R}}^* u \Rightarrow t \downarrow_{\mathcal{R}} u$.

The TRS $\mathcal{B} = \{f(x, x) \to a, f(x, g(x)) \to b, c \to g(c)\}$ is not confluent, as it permits the following rewrite sequences: $a \leftarrow f(c, c) \to f(c, g(c)) \to b$. Since no rules can be applied to a and b, convergence between them is not achievable. Confluence analysis plays a critical role in verifying if a system behaves deterministically for a given input [FGMP97] and in ensuring that the system's behavior remains consistent after transformations [HKT02]. Furthermore, when combined with termination analysis, it helps determine if a TRS is *complete*, an essential aspect for reducing the proof search space in automated theorem proving (ATP) [BG94] and for assessing the satisfiability of logical formulas in equational reasoning [BN98].

In parallel to termination analysis, the confluence research community develops various confluence tools and holds an annual confluence competition (CoCo). Key tools in this field include CSI [ZFM11], ACP [AYT09], and FORT-h [MLM23]. Many of these tools also integrate diverse confluence analysis techniques, such as those termination tools.

Confluence tools also incorporate various termination analysis methods, as many confluence techniques depend on proving the termination of the given TRSs. For example, CSI is built on T$_{\text{T}}$T$_2$ to leverage its termination analysis capabilities. A notable confluence analysis technique based on termination relies on Newman's lemma [BN98].

Definition 1.14. A TRS is *locally confluent* if and only if $\forall s, t, u \in \mathcal{T}(\mathcal{F}, \mathcal{V}), s \to t \wedge s \to u \Rightarrow t \downarrow^* u$.

Theorem 1.15 (Newman's lemma)**.** *Every terminating and locally confluent TRS is confluent.*

Since confluence analysis for terminating TRSs is decidable [Fel15], the major rewriting research focuses on confluence analysis on non-terminating TRSs.

1.7.3 Certification

Automatic rewriting tools face the significant challenge of potentially generating unsound results. This issue may arise from various sources, such as bugs within the implementation of the tools themselves, errors in the ATP solvers on which these tools depend, incorrect

configurations of the rewriting strategies, or inaccuracies in complex rewriting analyses proposed in some studies.

To address these concerns, researchers in the field of rewriting have developed several certification tools, including Certified Termination Analysis (CeTA) [TS09], CiME/Coccinelle [CCF+07, CFU08], and Rainbow/CoLoR [BCGD+06]. These tools are used to verify the correctness of proofs generated by rewriting tools. CeTA is implemented in Isabelle/HOL, while CiME/Coccinelle and Rainbow/CoLoR are based in Coq. CeTA initially can only certify termination analysis proofs, but its ability extends to certifying confluence analysis proofs. Notably, CeTA certifies proofs produced in termCOMP and CoCo.

The implementation of CeTA is based on the *Isabelle Formalization of Rewriting* library (IsaFoR) [TS09]. Researchers first formalize theories and proofs of TRSs in IsaFoR using Isabelle/HOL, then convert these formalizations into a certified Haskell program via Isabelle's code generator [HB13]. The generated Haskell program is ultimately compiled to create the executable certifier CeTA.

For CeTA's certification, rewriting tools must generate certificates in the certification problem format [ST14] alongside the proof. CeTA certifies the soundness of the proof if it confirms the validity of the related certificate.

However, CeTA cannot certify all valid proofs generated by rewriting tools. This limitation arises because rewriting tools often implement more extensive rewriting analysis techniques than those available in CeTA. The discrepancy is due to the significantly greater time investment required for formalization in Isabelle/HOL compared to developing rewriting tools in general-purpose programming languages.

1.8 Content

In this dissertation, I present my research on feature characterization and the implementation of various learning models for tactic-based ITP guidance in Chapter 3.

Chapter 4 introduces my work on the novel ITP learning task of learning proof transformations and its potential applications.

In Chapter 5, I describe my research utilizing ILP to learn rules that delineate when to apply specific tactics.

Chapter 2

Contributions

The subsequent chapter provides an overview of the publications included in this dissertation. Details regarding the publication venues and my specific contributions to each work are outlined in the following sections.

2.1 Online Machine Learning Techniques for Coq: a Comparison

Publication Details

> Liao Zhang, Lasse Blaauwbroek, Bartosz Piotrowski, Prokop Černý, Cezary Kaliszyk, and Josef Urban. Online machine learning techniques for Coq: A comparison. In *International Conference on Intelligent Computer Mathematics*, pages 67–83. Springer, 2021.

We introduce novel feature representations and implement three machine learning techniques for automating proofs in Coq. The original features were limited to a top-down traversal of the proof state up to length two. Our enhanced feature characterization incorporates additional aspects, including goal-hypothesis separation, top-level structures, abstract vertical traversals, and feature occurrence counts.

We developed three distinct machine learning approaches. The first is a locality-sensitive hashing forest, an efficient approximation of k-NN. The second is an online random forest model, which allows dynamic updates as new proof states become available. The third method employs gradient boosted trees.

We evaluated the new feature representations and the developed models on two tasks: tactic prediction and automated proof synthesis. Our empirical results demonstrate the effectiveness of both the novel features and the new machine learning models.

My Contributions

I designed novel features to characterize proof states for making tactic predictions. I implemented gradient boosted tree models and carried out performance evaluations on both k-NN and gradient-boosted trees. Additionally, I, together with Blaauwbroek, conducted experiments for evaluating online random forests on the dataset extracted from Coq. I also contributed by writing the relevant sections of the paper.

2.2 Learning Proof Transformations and its Applications in Interactive Theorem Proving

Publication Details

Liao Zhang, Lasse Blaauwbroek, Cezary Kaliszyk, and Josef Urban. Learning proof transformations and its applications in interactive theorem proving. In *International Symposium on Frontiers of Combining Systems*, pages 236-254. Springer Nature Switzerland Cham, 2023.

We propose a new task called learning proof transformation. A proof transformation refers to the transition from a proof state prior to the application of a tactic to the resulting proof states following the tactic application. Learning proof transformation involves leveraging this transformation to predict the tactic responsible for the transformation. In comparison, previous works merely utilize the proof state before the tactic application to make predictions.

We develop three characterizations for proof transformations: feature difference, anti-unification, and tree difference.

We also discover two applications of this task: making tactic suggestion and optimizing proof length.

We build random forests and GPT-2 for learning proof transformations. Our empirical experiments confirm the effectiveness of our characterizations in both learning proof transformations and the relevant applications.

My Contributions

I implemented the characterizations, developed the learning models, performed the experiments, and wrote the paper. I also identified making tactic suggestions as one of the two primary applications.

2.3 Learning Rules Explaining Interactive Theorem Proving Tactic Prediction

Publication Details

Liao Zhang, David M. Cerna, and Cezary Kaliszyk. Learning Rules Explaining Interactive Theorem Proving Tactic Prediction. In *International Joint Conference on Learning and Reasoning*. 2024.

We present the first study focusing on learning rules to improve tactic suggestion methods for ITP. We formulate the task of deciding the appropriateness of a tactic for a particular proof state as an ILP task. Although previous research has addressed making tactic predictions, it lacks explainability in its decision-making process. Moreover, existing methods solely utilize pre-computed features. However, pre-computation is prohibitively expensive for complex and precise features.

Using the ILP representation, we enrich the feature space by encoding additional, computationally expensive features as background predicates. Besides representing nodes in the AST as representation predicates, we also develop feature predicates and anonymous predicates as the background predicates. Feature predicates capture positional relationships between nodes and the equality between subterms in the AST. To leverage the ILP's generalization ability, anonymous predicates substitute the original nodes with their respective datatype in Coq's source code.

We use the enriched feature space to learn rules to explain the applicability of a tactic to a given proof state and filter the output of an existing tactic prediction approach using the learned rules.

Our empirical analysis validates the effectiveness of anonymous feature predicates in learning precise rules. We also use experiments to demonstrate the combination of ILP with an existing approach enhances the accuracy of making tactic predictions compared to the existing approach.

My Contributions

I conceived the concept of learning rules to explain when to use a tactic. I extracted the dataset, implemented the predicates and the algorithm for reordering predictions, and conducted the experiments. I also wrote the paper.

2.4 Automated Strategy Invention for Confluence of Term Rewrite Systems (contribution beyond the PhD)

The following work has not been published but is also a project that I worked on during my PhD.

We investigate the application of machine learning to term rewriting, a research field closely associated with formal theorem proving. Term rewriting can be leveraged to verify the validity of formulas in equational logic, which serves as the foundation for equational theorem provers [BG94, Sma21]. It is also widely applied to ITP and ATP to simplify the proof state for efficient computation and further application of other proof strategies.

Many properties of term rewrite systems are known to be undecidable. As a consequence, contemporary automatic term rewriting tools employ a diverse set of techniques to prove a certain property. The extensive variety of available rewriting techniques creates a vast strategy space, making the discovery of optimal strategies an intractable task for humans. The difficulty motivates us to automatically invent strategies for automatic term rewriting tools.

We develop the first learning-guided automatic confluence prover, where confluence is an important property of TRSs. We automatically invent a large number of strategies for the state-of-the-art automatic confluence prover CSI. Moreover, we design a mechanism to combine the invented strategies into a unified strategy. To address the limitations of the current confluence analysis dataset, which is too small for machine learning applications, we randomly generate a substantial number of TRSs and construct a new dataset.

Our empirical analysis validates that CSI with the invented unified strategy can (dis)prove confluence for more TRSs than the state-of-the-art approach. The invented strategies also discover proofs for several problems whose proofs have never been found by any participant in the annual confluence competition.

2.5 Transformers are Efficient Compilers, Provably (contribution beyond the PhD)

During my PhD, I worked on a project that has been accepted for presentation at the NeurIPS 2024 Workshop M3L. This work is currently under review for publication in a standard conference.

Transformers-based LLMs have exhibited remarkable performance across diverse fields; however, their theoretical capabilities remain only partially understood. In particular, LLMs have demonstrated notable progress in code generation [Any23, NHX+23] and compilation tasks [Tae23]. For instance, the Meta Large Language Model Compiler [CSG+24] enhances state-of-the-art LLMs by enabling them to better interpret compiler intermediate representations, assembly language, and optimization techniques, leading to improved compiler optimizations.

Given these advancements, a theoretical exploration of transformers' expressive power in compilation tasks becomes imperative. This research is the first to rigorously examine the expressive power of transformers in the context of compilers for programming languages.

We propose a representative programming language, *Mini-Husky*, which encapsulates the core features of modern C-like languages. Assuming that the input code can be tokenized into a sequence of length L, this sequence is characterized by bounded depths in both its AST and a crucial step in type analysis called type inference [Pie02]. Our theoretical results demonstrate that transformers can perform compilation tasks including abstract syntax tree generation, symbol resolution, and type analysis by relying only on the logarithm of the input sequence length L. In contrast, we show that recurrent neural networks (RNNs) require at least a linear dependency on L to accomplish the same tasks. We further validate these findings through empirical experiments, confirming the superiority of transformers over RNNs in this domain.

A key technical contribution of our work is the development of the domain-specific programming language *Cyberton*, which is capable of automatically generating proofs regarding the expressive power of transformers. Its development is driven by the significant complexities inherent in manually constructing expressive power proofs for neural networks.

Chapter 3

Online Machine Learning Techniques for Coq: a Comparison

3.1 Abstract

We present a comparison of several online machine learning techniques for tactical learning and proving in the Coq proof assistant. This work builds on top of Tactician, a plugin for Coq that learns from proofs written by the user to synthesize new proofs. Learning happens in an online manner, meaning that Tactician's machine learning model is updated immediately every time the user performs a step in an interactive proof. This has important advantages compared to the more studied offline learning systems: (1) it provides the user with a seamless, interactive experience with Tactician and, (2) it takes advantage of locality of proof similarity, which means that proofs similar to the current proof are likely to be found close by. We implement two online methods, namely approximate k-nearest neighbors based on locality sensitive hashing forests and random decision forests. Additionally, we conduct experiments with gradient boosted trees in an offline setting using XGBoost. We compare the relative performance of Tactician using these three learning methods on Coq's standard library.

3.2 Introduction

The users of interactive theorem proving systems are in dire need of a digital sidekick, which helps them reduce the time spent proving the mundane parts of their theories, cutting down on the man-hours needed to turn an informal theory into a formal one. The obvious way of creating such a digital assistant is using machine learning. However, creating a practically usable assistant comes with some requirements that are not necessarily conducive to the most trendy machine learning techniques, such as deep learning.

The environment provided by ITPs is highly dynamic, as it maintains an ever-changing global context of definitions, lemmas, and custom tactics. Hence, proving lemmas within such environments requires intimate knowledge of all the defined objects within the global context. This is contrasted by—for example—the game of chess; even though the search space is enormous, the pieces always move according to the same rules, and no new kinds of pieces can be added. Additionally, the interactive nature of ITPs demands that machine

learning techniques do not need absurd amounts of time and resources to train (unless a pre-trained model is highly generic and widely applicable across domains; something that has not been achieved yet). In this paper, we are interested in online learning techniques that quickly learn from user input and immediately utilize this information. We do this in the context of the Coq proof assistant [The19] and specifically Tactician [BUG20c]—a plugin for Coq that is designed to learn from the proofs written by a user and apply that knowledge to prove new lemmas.

Tactician performs a number of functions, such as proof recording, tactic prediction, proof search, and proof reconstruction. In this paper, we focus on tactic prediction. For this, we need a machine learning technique that accepts as input a database of proofs, represented as pairs containing a proof state and the tactic that was used to advance the proof. From this database, a machine learning model is built. The machine learning task is to predict an appropriate tactic when given a proof state. Because the model needs to operate in an interactive environment, we pose four requirements the learning technique needs to satisfy:

1. The model (datastructure) needs to support dynamic updates. That is, the addition of a new pair of a proof state and tactic to the current model needs to be done in (near) constant time.

2. The model should limit its memory usage to fit in a consumer laptop. We have used the arbitrary limit of 4 GB.

3. The model should support querying in (near) constant time.

4. The model should be persistent (in the functional programming sense [DSST89]). This enables the model to be synchronized with the interactive Coq document, in which the user can navigate back and forth.

3.2.1 Contributions

In this work, we have implemented two online learning models. An improved version of the locality sensitive hashing scheme for k-nearest neighbors is described in detail in Section 3.4.1. An implementation of random forest is described in Section 3.4.2. In Section 3.5, we evaluate both models, comparing the number of lemmas of Coq's standard library they can prove in a chronological setting (i.e., emulating the growing library).

In addition to the online models, as a proof of concept, we also experiment in an offline fashion with boosted trees, specifically XGBoost [CG16] in Section 3.4.3. Even though the model learned by XGBoost cannot be used directly in the online setting described above, boosted trees are today among the strongest learning methods. Online algorithms for boosted trees do exist [ZZS+19], and we intend to implement them in the future.

The techniques described here require representing proof states as feature vectors. Tactician already supported proof state representation using simple hand-rolled features [BUG20a]. In addition, Section 3.3 describes our addition of more advanced features of the proof states, which are shown to improve Tactician's performance in Section 3.5.

3.3 Tactic and Proof State Representation

To build a learning model, we need to characterize proof states and the tactics applied to them. To represent tactics, we first perform basic decompositions and simplifications and denote the resulting atomic tactics by their hashes [BUG20a].

Tactician's original proof state features [BUG20a] consist merely of identifiers and adjacent identifier pairs in the abstract syntax tree (AST). Various other, more advanced features have been considered for automated reasoning systems built over large formal mathematical knowledge bases [CJSU19a, GKU17, KUV15b]. To enhance the performance of Tactician, we modify the old feature set and define new features as follows.

Top-down Oriented AST Walks We add top-down oriented walks in the AST of length up to 3 with syntax placeholders. For instance, the unit clause $f(g(x))$ will contain the features:

```
f:AppFun, g:AppFun, x:AppArg, f:AppFun(g:AppFun),
g:AppFun(x:AppArg), f:AppFun(g:AppFun(x:AppArg))
```

The feature `g:AppFun` indicates that g is able to act as a function in the term tree, and `x:AppArg` means that x is only an argument of a function.

Vertical Abstracted Walks We add vertical walks in the term tree from the root to atoms in which nonatomic nodes are substituted by their syntax roles. For the term $f_1(f_2(f_3(a)))$, we can convert each function symbol to `AppFun` whereas the atom a is transformed to `a:AppArg` as above. Subsequently, we can export this as the feature `AppFun(AppFun(AppFun(a:AppArg)))`. Such abstracted features are designed to better capture the overall abstract structure of the AST.

Top-level Structures We add top-level patterns by replacing the atomic nodes and substructures deeper than level 2 with a single symbol `X`. Additionally, to separate the function body and arguments, we append the arity of the function to the corresponding converted symbol. As an example, consider the term $f(g(b,c),a)$ consisting of atoms a, b, c, f, g. We first replace a, f, g with `X` because they are atomic. We further transform f and g to `X2` according to the number of their arguments. However, b and c break the depth constraint and should be merged to a single `X`. Finally, the concrete term is converted to an abstract structure `X2(X2(X),X)`. Abstracting a term to its top-level structure is useful for determining whether a "logical" tactic should be applied. As an illustration, the presence of $X \wedge X$ in the goal often indicates that we should perform case analysis by the `split` tactic. Since we typically do not need all the nodes of a term to decide such structural information, and we want to balance the generalization with specificity, we use the maximum depth 2.

Premise and Goal Separation Because local hypotheses typically play a very different role than the conclusion of a proof state, we separate their feature spaces. This can be

done by serially numbering the features and adding a sufficiently large constant to the goal features.

Adding Occurrence Counts In the first version of Tactician, we have used only a simple boolean version of the features. We try to improve on this by adding the number of occurrences of each feature in the proof state.

3.4 Prediction Models

3.4.1 Locality Sensitive Hashing Forests for Online kNN

One of the simplest methods to find correlations between proof states is to define a metric or similarity function $d(x, y)$ on the proof states. One can then extract an ordered list of length k from a database of proof states that are as similar as possible to the reference proof state according to d. Assuming that d does a good job identifying similar proof states, one can then use tactics known to be useful in a known proof state for an unseen proof state. In this paper, we refer to this technique as the k-nearest neighbor (k-NN) method (even though this terminology is somewhat overloaded in the literature).

Our distance function is based on the features described in Section 3.3. We compare these features using the Jaccard index $\mathrm{J}(f_1, f_2)$. Optionally, features can be weighted using the TfIdf statistic [Jon04], in which case the generalized index $\mathrm{J}_w(f_1, f_2)$ is used.

$$\mathrm{J}(f_1, f_2) = \frac{|f_1 \cap f_2|}{|f_1 \cup f_2|} \quad \mathrm{tfidf}(x) = \log \frac{N}{|x|_N} \quad \mathrm{J}_w(f_1, f_2) = \frac{\sum_{x \in f_1 \cap f_2} \mathrm{tfidf}(x)}{\sum_{x \in f_1 \cup f_2} \mathrm{tfidf}(x)}$$

Here N is the database size, and $|x|_N$ is the number of times feature x occurs in the database. In previous work, we have made a more detailed comparison of similarity functions [BUG20a].

A naive implementation of the k-NN method is not very useful in the online setting because the time complexity for a query grows linearly with the size of the database. Indexing methods, such as k-d trees, exist to speed up queries [Ben75]. However, these methods do not scale well when the dimensionality of the data increases [HIM12]. In this work, we instead implement an approximate version of the k-NN method based on Locality Sensitive Hashing (LSH) [GIM99]. This is an upgrade of our previous LSH implementation that was not persistent and was slower. We also describe our functional implementation of the method in detail for the first time here.

The essential idea of this technique is to hash feature vectors into buckets using a family of hash functions that guarantee that similar vectors hash to the same bucket with high probability (according to the given similarity function). To find a k-NN approximation, one can simply return the contents of the bucket corresponding to the current proof state. For the Jaccard index, the appropriate family of hash functions are the MinHash functions [Bro97].

The downside of the naive LSH method is that its parameters are difficult to tune. The probability that the vectors that hash to the same bucket are similar can be increased by

associating more than one hash function to the bucket. All values of the hash functions then need to pair-wise agree for the items in the bucket. However, this will naturally decrease the size of the bucket, lowering the number of examples k (of k-NN) that can be retrieved. The parameter k can be increased again by simply maintaining multiple independent bucketing datastructures. Tuning these parameters is critically dependent on the size of the database, the length of the feature vectors, and the desired value of k. To overcome this, we implement a highly efficient, persistent, functional variant of Locality Sensitive Hashing Forest [BCG05] (LSHF), which is able to tune these parameters automatically, leaving (almost) no parameters to be tuned manually. Below we give a high-level overview of the algorithm as it is modified for a functional setting. For a more in-depth discussion on the correctness of the algorithm, we refer to the previous reference.

LSHFs consist of a forest (collection) of tries $\mathcal{T}_1 \dots \mathcal{T}_n$. Every trie has an associated hash function h_i that is a member of a (near) universal hashing family mapping a feature down to a single bit (a hash function mapping to an integer can be used by taking the result modulus two). To add a new example to this model, it is inserted into each trie according to a path (sequence) of bits. Every bit of this path can be shown to be locally sensitive for the Jaccard index [BCG05]. The path of an example is calculated using the set of features that represents the proof state in the example.

$$\text{path}_i(f) = \text{sort}(\{h_i(x) \mid x \in f\})$$

For a given trie $\mathcal{T}$, the subtrie starting at a given path $b_1 \dots b_m$ can be seen as the bucket to which examples that agree on the hashes $b_1 \dots b_m$ are assigned. Longer paths point to smaller buckets containing less similar examples, while shorter paths point to larger buckets containing increasingly similar examples. Hence, to retrieve the neighbors of a proof state with features f, one should start by finding examples that share the entire path of f. To retrieve more examples, one starts collecting the subtrees starting at smaller and smaller prefixes of $\text{path}_i(f)$. To increase the accuracy and number of examples retrieved, this procedure can be performed on multiple tries simultaneously, as outlined in Algorithm 2.

Tuning the LSHF model consists mainly of choosing the appropriate number of tries that maximizes the speed versus accuracy trade-off. Experiments show that 11 trees is the optimal value. Additionally, for efficiency reasons, it is a good idea to set a limit on the depth of the tries to prevent highly similar examples from creating a deep trie. For our dataset, a maximum depth of 20 is sufficient.

3.4.2 Online Random Forest

Random forests are a popular machine learning method combining many randomized decision trees into one ensemble, which produces predictions via voting [Bre01a]. Even though the decision trees are not strong learners on their own, because they are intentionally decorrelated, the voting procedure greatly improves on top of their individual predictive performance. The decision trees consist of internal nodes labeled by decision rules and leaves labeled by examples. In our case, these are tactics to be applied in the proofs.

Algorithm 2 Querying the Locality Sensitive Hashing Forest

```
1: function QueryLSHF(F, f)                          ▷ F a forest, f a feature set
2:     P ← ⟨path_i(f) : i ∈ [1..|F|]⟩
3:     neighbors ← FilterDuplicates(SimultaneousDescend(F, P))
4:     Optionally re-sort neighbors according to real Jaccard index
5: function SimultaneousDescend(F, P)
6:     F_rel ← ⟨ if head(P) then left(T) else right(T) : T ∈ F when not leaf(T) ⟩
7:     F_irrel ← ⟨ if leaf(T) then T elseif head(P) then right(T) else left(T) : T ∈ F ⟩
8:     if F_rel is empty then
9:         neighbors ← empty list
10:    else
11:        P' ← ⟨tail(P_i) : i ∈ [1..n]⟩
12:        neighbors ← SimultaneousDescend(F_rel, P')
13:    if |neighbors| ≥ k then
14:        return neighbors
15:    else
16:        return Append(neighbors, Concatenate(⟨ Collect(T : T ∈ F_irrel⟩)))
```

Random forests are a versatile method that requires little tuning of its hyperparameters. Their architecture is also relatively simple, which makes it easy to provide a custom OCaml implementation easily integrable with Tactician, adhering to its requirement of avoiding mutable data structures. Direct usage of existing random forest implementations is impossible due to challenges in Tactician's learning setting. These challenges are: (1) numerous sparse features, (2) the necessity of online learning, as detailed in the next two paragraphs.

The decision rules in nodes of the decision trees are based on the features of the training examples. These rules are chosen to maximize the *information gain*, i.e., to minimize the *impurity* of the set of labels in the node.[1] There are more than $37,000$ binary and sparse features in Tactician. Since the learner integrated with Tactician needs to be fast, one needs to be careful when optimizing the splits in the tree nodes.

Random forests are typically trained in an offline manner where the whole training data is available at the beginning of the training. In Tactician this would be quite suboptimal. To take advantage of the locality of proof similarity and to be able to use data coming from new proofs written by a user, we want to immediately update the machine learning model after each proof.

There are approaches to turn random forests into online learners [DH00, SLS+09] which inspired our implementation. The authors of [DH00] propose a methodology where new training examples are passed to the leaves of the decision trees, and under certain statistical conditions, the leaf is split and converted to a new decision node followed by two new leaves. We take a similar approach, but deciding a split in our implementation

[1] If we have labels $\{a, a, b, b, b\}$, ideally, we would like to produce a split which passes all the examples with label a to one side and the examples with b to the other side.

Algorithm 3 Adding training a example e to a decision tree $\mathcal{T}$

```
1: function AddExampleToTree(𝒯, e)
2:     match 𝒯 with
3:         Node(ℛ, 𝒯_l, 𝒯_r):          ▷ ℛ – binary rule, 𝒯_l, 𝒯_r – left and right subtrees
4:             match ℛ(e) with
5:                 Left: return Node(ℛ, AddExampleToTree(𝒯_l, e), 𝒯_r)
6:                 Right: return Node(ℛ, 𝒯_l, AddExampleToTree(𝒯_r, e))
7:         Leaf(l, ℰ):                  ▷ l – label/tactic, ℰ – examples
8:             ℰ ← Append(ℰ, e)
9:             if SplitCondition(ℰ) then
10:                ℛ ← GenerateSplitRule(ℰ)
11:                ℰ_l, ℰ_r ← Split(ℛ, ℰ)
12:                l_l ← label of random example from ℰ_l
13:                l_r ← label of random example from ℰ_r
14:                return Node(ℛ, Leaf(l_l, ℰ_l), Leaf(l_r, ℰ_r))
15:            else
16:                return Leaf(l, ℰ)
```

is simpler and computationally cheaper.

The pseudocode describing our implementation is presented below. Algorithm 3 shows how the training examples are added to the decision trees. A new training example is passed down the tree to one of its leaves. The trajectory of this pass is governed by binary decision rules in the nodes of the tree. Each rule checks whether a given feature is present in the example. Once the example reaches a leaf, it is saved there, and a decision is made whether to extend the tree (using function SplitCondition). This happens only when the Gini Impurity measure [Mit97] on the set of examples in the leaves is greater than a given impurity threshold i (a hyperparameter of the model). When the split is done, the leaf becomes an internal node with a new split rule, and the collected examples from the leaf are passed down to the two new leaves. The new rule (an output from GenerateSplitRule) is produced in the following way. N features are selected from the features of the examples, where N is the square root of the number of examples. The selection of the features is randomized and made in such a way that features that are distinguishing between the examples have higher probability: First, we randomly select two examples from the leaf, and then we randomly select a feature from the difference of sets of features of the two examples. Among such selected features, the one maximizing the *information gain* [Mit97] of the split rule based on it is selected. The two new leaves get labels randomly selected from the examples belonging to the given leaf.

When adding an example to a random forest (Algorithm 4), first, a decision is made whether a new tree (in the form of a single leaf) should be added to the forest. It happens with probability $\frac{1}{n}$, where n is the number of trees in the forest under the condition that n is lower than a given threshold.

Predicting a tactic for a given example with a random forest (Algorithm 5) is done

Algorithm 4 Adding a training example e to a random forest $\mathcal{F}$

```
1: function AddExampleToForest(F, e, n_max)        ▷ n_max – max number of trees
2:     n ← number of trees in F
3:     m ← random number from {1, ... n}
4:     F_updated ← empty list
5:     if n < n_max and m = 1 then
6:         T ← leaf labeled by tactic used in e
7:         F_updated ← Append(F_updated, T)
8:     for all T ∈ F do
9:         T ← AddExampleToTree(T, e)
10:        F_updated ← Append(F_updated, T)
11:    return F_updated
```

Algorithm 5 Predicting labels for unlabeled e in the random forest $\mathcal{F}$

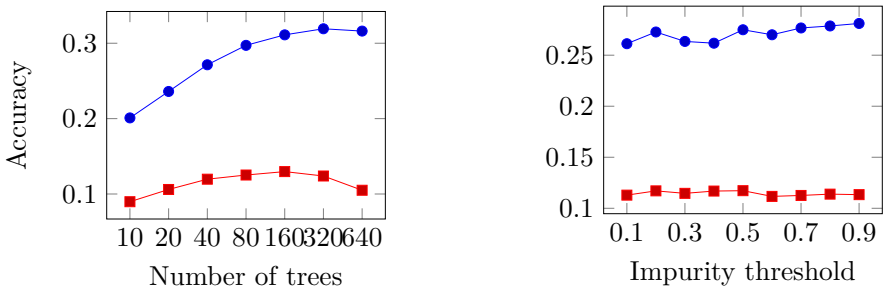
```
1: function PredictForest(F, e)
2:     P ← empty list                              ▷ P – predictions
3:     for all T ∈ F do
4:         t ← PredictTree(e)
5:         append t to P
6:     R ← Vote(P)                                 ▷ R – ranking of tactics
7:     return R
8: function PredictTree(T, e)
9:     match T with
10:        Node(R, T_l, T_r):
11:            match R(e) with
12:                Left: return PredictTree(T_l, e)
13:                Right: return PredictTree(T_r, e)
14:        Leaf(l, E): return l
```

in two steps. First, the example is passed to the leaves of all the trees and the labels (tactics) in the leaves are saved. Then the ranking of the tactics is made based on their frequencies.

Tuning Hyperparameters

There are two hyperparameters in our implementation of random forests: (1) the maximal number of trees in the forest and (2) the impurity threshold for performing the node splits. To determine the influence of these parameters on the predictive power, we perform a grid search. For this, we randomly split the data that is not held out for testing (see Section 3.5.1) into a training and validation part. The results of the grid search are shown in Figure 3.1. The best numbers of trees are 160 (for top-1 accuracy) and 320

Figure 3.1: Results of hyperparameter tuning for random forests. The blue circle corresponds to top-10 accuracy (how often the correct tactic was present in the first 10 predictions) whereas the red square corresponds to top-1 accuracy.



(for top-10 accuracy). We used these two values for the rest of the experiments. For the impurity threshold, it is difficult to see a visible trend in performance; thus we selected 0.5 as our default.

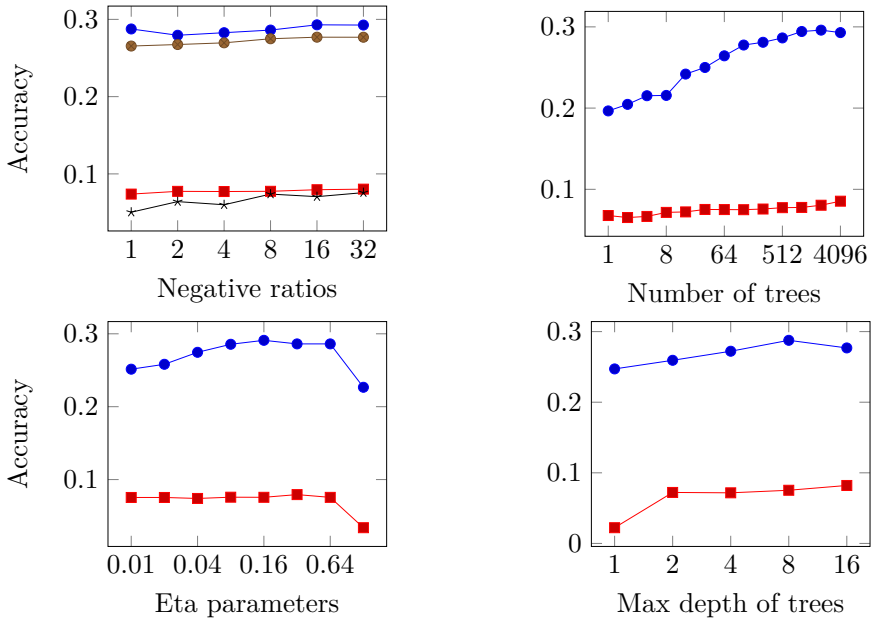
3.4.3 Boosted Trees

Gradient boosted decision trees are a state-of-the-art machine learning algorithm that transforms weak base learners, decision trees, into a method with stronger predictive power by appropriate combinations of the base models. One efficient and powerful implementation is the XGBoost library. Here, we perform some initial experiments in an offline setting for tactic prediction. Although XGBoost can at the moment not be directly integrated with Tactician, this gives us a useful baseline based on existing state-of-the-art technology. Below, we illustrate a procedure of developing our XGBoost model based on binary logistic regression.

The input to XGBoost is a sparse matrix containing rows with the format of (ϕ_P, ϕ_T) where ϕ_P includes the features of a proof state, and ϕ_T characterizes a tactic related to the proof state. We transform each proof state to a sparse feature vector ϕ_P containing the features' occurrence counts. Since there may be a large number of features in a given Coq development environment, which may hinder the efficiency of training and prediction, it is reasonable to decrease the dimension of the vectors. We hash the features to $20,000$ buckets by using the modulo of the feature's index. As above, we also remap the tactic hashes to a $20,000$-dimensional space separated from the state features.

The training examples get labels 1 or 0 based on the tactics being useful or not for the proof state. A tactic for a certain proof state is labeled as positive if it is exactly the one applied to this state in the library. In contrast, negative tactics are elements in the tactic space that differ from the positive instance. We obtain negative data by two approaches: *strong* negatives and *random* negatives. Strong negative instances are obtained by arbitrarily selecting a subset from the best-100 k-NN predictions for this state. In the other approach, negative instances are arbitrarily chosen from the entire tactic space.

Figure 3.2: Results of hyperparameter tuning for gradient boosted trees. In consistence with Figure 3.1, the blue circle (red square) corresponds to top-10 (top-1) accuracy, respectively. The graph of negative ratios contains two additional curves of random negative examples. The brown circle relates to top-10 accuracy, whereas the black star presents the results of top-1 accuracy.



With a trained gradient boosted trees model, we can predict the scores of the tactics for an unseen proof state P. First, the top-100 k-NN predictions are preselected. Then, for each tactic, we input (ϕ_P, ϕ_T) to the model to obtain the score of T. The tactics are then sorted according to their scores.

Tuning Hyperparameters

Similarly as for the random forest model (Section 3.4.2), we optimize the most important hyperparameters of the XGBoost training algorithm on the data coming from the non-sink nodes in the dependency graph of Coq's standard library (see Section 3.5.1). One essential parameter is the *ratio* of negative examples. Ratio n indicates that we generate n negative instances for each recorded proof state. Other influential parameters that we tune are: *eta* (learning-rate), *number of trees*, and *max depth*. Due to the limitations of computing resources, we assume a set of default parameters: *ratio* $= 8$, *eta* $= 0.2$, *number of trees* $= 500$, *max depth* $= 10$, and then separately modify each of these parameters to observe the influence caused by the change, which is depicted in Figure 3.2. Both strong and random negatives are evaluated. Obviously, strong negatives perform better than random negatives, and increasing the negative ratios will certainly lead to

higher success rates. The figure also shows that a higher number of trees results in better performance. Learning rates are between 0.08 and 0.64 give good results. It is also apparent that deeper trees (at least 8) increase the accuracy.

Experimental Setup

The XGBoost model is evaluated on the task of tactic prediction both in the split setting and the chronological setting (illustrated in Section 3.5). We use the strong negative examples and determine the final parameters—*ratio* = 16, *eta* = 0.2, *number of trees* = 1024, *max depth* = 10—for generating a model from non-sink nodes and use that to predict for sink nodes.

Since the entire dataset contains approximately 250,000 proof states, and it is time-consuming to generate a unique XGBoost model for each test case, we propose several ways to speed up the chronological evaluation. Instead of training on the data from all preceding states, we merely provide 1,000 instances occurring previously as the training data. According to the results of parameter tuning depicted in Figure 3.2, we decide on the hyperparameters—*ratio* = 4, *eta* = 0.2, *number of trees* = 256, *max depth* = 10—to balance the accuracy and efficiency.

3.5 Experimental Evaluation

To compare the performance of the described machine learning models, we perform three kinds of experiments: *split* evaluation, *chronological* evaluation, and evaluation in Tactician. Achieving good performance in the last type of evaluation is the main goal. All three machine learning models are evaluated in the first two kinds of experiments, while in Tactician we only evaluate k-NN and online random forest. This is because the XGBoost system, while being potentially the strongest machine learner among tested, may not be easily turned into an online learner and integrated into Tactician. We adopt the original features—term and term pairs—for evaluation outside Tactician, whereas both the original features and the new are tested on Tactician's benchmark. To determine the relative importance of the feature classes described in Section 3.3, we benchmark the addition of each class separately in Tactician. All evaluations are performed on data extracted from the standard library of Coq 8.11.

3.5.1 Split Evaluation

In the directed acyclic graph of dependencies of the Coq modules, there are 545 nodes. 104 of them are *sink nodes*, i.e., these are the modules that do not appear among dependencies of any other module. We used these modules as final testing data for evaluation outside Tactician. The rest of the data was randomly split into training and validation parts and was used for parameter tuning of random forest and gradient boosted trees. The models with tuned hyperparameters were evaluated on the testing data. The results of the evaluation of the three tested models are shown in the first row of Table 3.1.

Table 3.1: Performance of the three tested machine learning models in two types of evaluation: using a split of the dataset and a chronological evaluation through the dataset. top-n refers to the frequency of the correct tactic being present in the first n predictions from a machine learning model.

	Machine learning system					
	k-NN		Random Forest		XGBoost	
Evaluation type	top-1	top-10	top-1	top-10	top-1	top-10
split	18.8%	34.2%	32.1%	41.2%	18.2%	38.2%
chronological	17.3%	43.7%	29.9%	58.9%	18.2%	43.4%

3.5.2 Chronological Evaluation

Although the split evaluation from the previous experiment is interesting, it does not correspond entirely to the Tactician's internal mode of operation. To simulate the real-world scenario in an offline setting, we create an individual model for each proof state by learning from all the previous states—data from dependent files and preceding lines in the local file. The second row of Table 3.1 presents the results of the evaluation in chronological order.

3.5.3 Evaluation in Tactician

Table 3.2 shows the results of the evaluation of two online learners—the k-NN and the random forest—within Tactician. The hyperparameters of the random forest model were chosen based on the grid search in Section 3.4.2. We run the proof search for every lemma in the library with a 40-second time limit on both the original and the improved features.

The random forest performed marginally better than k-NN on both kinds of features. With old features the k-NN proved 3831 lemmas (being 33.7% out of all 11370), whereas the random forest proved 4011 lemmas (35.3% of all). With the new features, both models performed better, and again, the random forest proved more lemmas (4117, 36.2% of all) than k-NN (3945, 34.7% of all).

It is somewhat surprising that the random forest, which performed much better than k-NN on the split in the offline evaluation, is only better by a small margin in Tactician. This may be related to the time and memory consumption of random forest, which may be higher than for k-NN on certain kinds of data.[2]

It is worth noting that k-NN and random forest resulted in quite different sets of proofs. The columns marked as *union* show that the size of the union of proofs constructed by the two models is significantly larger than the number of proofs found by each model separately. In total, both models resulted in 4503 (39.6%) proofs using old features and 4597 (40.4%) proofs using the new features.

[2]Doing the splits in the leaves has quadratic time complexity with respect to the number of examples stored in the leaf; sometimes it happens, that leaves of the trees store large number of examples.

Table 3.2: Proving performance of two online learners integrated with Tactician, k-NN and random forest, in the Coq Standard Library. The percentages in the table correspond to the fraction of lemmas proved in a given Coq module. The columns *union* show what fraction of the lemmas was proved by at least one of the learners. RF is an abbreviation of random forest.

Coq module	**#Lemmas**	**Features type**					
		Original			New		
		k-NN	RF	*union*	k-NN	RF	*union*
All	**1137**	**33.7%**	**35.3%**	**39.6%**	**34.7%**	**36.2%**	**40.4%**
Arith	293	52%	59%	65%	56%	59%	66%
Bool	130	93%	87%	93%	92%	88%	92%
Classes	191	80%	76%	81%	79%	79%	83%
FSets	1137	32%	34%	37%	32%	35%	39%
Floats	5	20%	20%	20%	40%	19%	40%
Init	164	73%	51%	73%	73%	56%	73%
Lists	388	38%	43%	47%	38%	44%	49%
Logic	341	31%	27%	34%	32%	31%	35%
MSets	830	38%	40%	43%	36%	40%	43%
NArith	288	37%	43%	44%	35%	42%	47%
Numbers	2198	23%	22%	27%	24%	23%	27%
PArith	280	31%	40%	44%	35%	39%	45%
Program	28	75%	64%	75%	78%	66%	78%
QArith	295	33%	40%	43%	31%	39%	45%
Reals	1756	19%	23%	25%	21%	24%	26%
Relations	37	29%	24%	40%	27%	26%	29%
Setoids	4	1.00	1.00	1.00	1.00	97%	1.00
Sets	222	43%	42%	49%	49%	47%	53%
Sorting	136	26%	29%	33%	25%	30%	33%
Strings	74	22%	22%	27%	17%	14%	20%
Structures	390	45%	49%	54%	51%	51%	56%
Vectors	37	37%	29%	40%	21%	23%	27%
Wellfounded	36	19%	05%	19%	16%	13%	16%
ZArith	953	41%	46%	49%	40%	43%	46%
btauto	44	11%	20%	20%	20%	17%	22%
funind	4	75%	50%	75%	50%	73%	75%
micromega	339	21%	27%	29%	27%	25%	30%
nsatz	27	33%	33%	37%	40%	26%	40%
omega	37	40%	67%	67%	48%	63%	64%
rtauto	33	30%	39%	48%	33%	44%	51%
setoid_ring	362	21%	23%	26%	27%	27%	30%
ssr	311	68%	55%	69%	70%	57%	71%

3.5.4 Feature Evaluation

Table 3.3 depicts the influence of adding the new classes of features described in Section 3.3 to the previous baseline.[3] All of the newly produced features improve the success rates.

[3]The results here are not directly comparable to those in Table 3.2 mainly due to the usage of a non-indexed version of k-NN in contrast to the algorithm presented in 2.

Table 3.3: Proving performance of each feature modification. $\mathcal{O}, \mathcal{W}, \mathcal{V}, \mathcal{T}, \mathcal{S}, \mathcal{C}$ denote original features, top-down oriented AST walks, vertical abstract walks, top-level structures, premise and goal separation, and adding feature occurrence, respectively. The symbol $\oplus$ denotes that we combine the original features and a new modification during the experiments.

Features	$\mathcal{O}$	$\mathcal{O} \oplus \mathcal{W}$	$\mathcal{O} \oplus \mathcal{V}$	$\mathcal{O} \oplus \mathcal{T}$	$\mathcal{O} \oplus \mathcal{S}$	$\mathcal{O} \oplus \mathcal{C}$
Success rates (%)	32.75	32.82	34.16	33.65	34.42	34.97

However, the top-down oriented AST walks contribute little, probably due to Tactician having already included term tree walks up to length 2. Every other modification obtains a reasonable improvement, which confirms the intuitions described in Section 3.3.

3.6 Related Work

Random forests were first used in the context of theorem proving by Färber [FK15b], where multi-path querying of a random forest would improve on k-NN results for premise selection. Nagashima and He [NH18] proposed a proof method recommendation system for Isabelle/HOL based on decision trees on top of precisely engineered features. A small number of trees and features allowed for explainable recommendations. Frameworks based on random boosted trees (XGBoost, LightGBM) have also been used in automated reasoning, in the context of guiding tableaux connection proof search [KUMO18b] and the superposition calculus proof search [CJSU19a], as well as for handling negative examples in premise selection [PU18].

Machine learning to predict tactics was first considered by Gauthier et al. [GKU17] in the context of the HOL4 theorem prover. His later improvements [GKU$^+$21b] added Monte-Carlo tree search, tactic orthogonalization, and integration of both Metis and a hammer [GK15a]. A similar system for HOL Light was developed by Bansal et al. [BLR$^+$19b]. Nagashima and Kumar developed the proof search component [NK17] of such a system for Isabelle/HOL. This work builds upon Tactician [BUG20c, BUG20a], adapting and improving these works for dependent type theory and the Coq proof assistant.

3.7 Conclusion

We have implemented several new methods for learning tactical guidance of Coq proofs in the Tactician system. This includes better proof state features and an improved version of approximate k-nearest neighbor based on locality sensitive hashing forests. A completely new addition is our online implementation of random forest in Coq, which can now be used instead of or together with the k-nearest neighbor. We have also started to experiment with strong state-of-the-art learners based on gradient boosted trees, so far in an offline setting using binary learning with negative examples.

Our random forest improves very significantly on the k-nearest neighbor in an offline accuracy-based evaluation. In an online theorem-proving evaluation, the improvement is not as big, possibly due to the speed of the two methods and the importance of backtracking during the proof search. The methods are, however, quite complementary and running both of them in parallel increases the overall performance of Tactician from 33.7% (k-NN with the old features) to 40.4% in 40s. Our best new method (RF with the new features) now solves 36.2% of the problems in 40s.

The offline experiments with gradient boosted trees are so far inconclusive. They outperform k-nearest neighbor in top-10 accuracy, but the difference is small, and the random forest performs much better in this metric. Since the random forest learns only from positive examples, this likely shows that learning in the binary setting with negative examples is challenging on our Tactician data. In particular, we likely need good semantic feature characterizations of the tactics, obtained e.g., by computing the difference between the features of the proof states before and after the tactic application. The experiments, however, already confirm the importance of choosing good negative data to learn from in the binary setting.

Chapter 4

Learning Proof Transformations and its Applications in Interactive Theorem Proving

4.1 abstract

Interactive theorem provers are today increasingly used to certify mathematical theories. To formally prove a theorem, reasoning procedures called tactics are invoked successively on the proof states starting with the initial theorem statement, transforming them into subsequent intermediate goals, and ultimately discharging all proof obligations. In this work, we develop and experimentally evaluate approaches that predict the most likely tactics that will achieve particular desired transformations of proof states. First, we design several characterizations to efficiently capture the semantics of the proof transformations. Then we use them to create large datasets on which we train state-of-the-art random forests and language models. The trained models are evaluated experimentally, and we show that our best model is able to guess the right tactic for a given proof transformation in 74% of the cases. Finally, we use the trained methods in two applications: proof shortening and tactic suggesting. To the best of our knowledge, this is the first time that tactic synthesis is trained on proof transformations and assists interactive theorem proving in these ways.

4.2 Introduction

Interactive theorem provers (ITPs) [HUW14] are sophisticated systems used for constructing machine-verified proofs. Various proof assistants, such as HOL4 [SN08b], HOL Light [Har96], Lean [dMKA+15], Isabelle/HOL [NWP02], and Mizar [BBG+15], are used by formalizers. Coq [The20] is one of the most popular proof assistant systems. Coq formalizers invoke reasoning procedures called *tactics* that transform proof states into simpler proof states, eventually discharging all proof obligations and thus proving the initial proof state.

To give a simple example, we show a Coq proof of the equality of the lengths of a list and its reverse (Figure 4.1). To complete the proof, one can perform induction on the list `l` (with the help of the tactic `induction l as [| n l' IHl']`), splitting the proof state into a case where `l` is empty and a case where `l` is nonempty. In the first case,

```
Theorem rev_length : ∀ l : list nat, length (rev l) = length l.
Proof.
  intros l. induction l as [| n l' IHl'].
  - reflexivity.
  - simpl. rewrite → app_length. simpl. rewrite → IHl'.
    rewrite add_comm. reflexivity.
Qed.
```

Figure 4.1: A formal Coq proof, showing the equality property of the lengths of a list and its reverse

```
n : nat
l' : list nat
IHl' : length (rev l') = length l'
____________________________________(1/1)
length l' + 1 = S (length l')
```

(a) Before state

```
n : nat
l' : list nat
IHl' : length (rev l') = length l'
____________________________________(1/1)
1 + length l' = S (length l')
```

(b) After state

Figure 4.2: The before and after states of `rewrite add_comm` in Figure 4.1, with hypotheses above the dashed line and the required goal below it.

the goal reduces to `length (rev []) = length []`, which is easily discharged using simple computation. In the second case, we obtain the induction hypothesis `IHl'` that states `length (rev l') = length l'` and need to prove that the equation still holds when the original list has a natural number `n` prepended to it. After some simplification, we transform the length of the concatenation of two lists into the summation of their individual lengths. Then, with the help of the induction hypothesis, we simplify the goal. Finally, we rewrite the goal by the commutative property of addition and obtain a simple equation to prove.

A Coq proof state consists of a list of hypotheses and a goal that needs to be proven. Given a proof state before the tactic application, the tactic may either transform the *before state* to several *after states* or finish the proof. The *semantic* of a tactic is captured by the (usually infinite) set of proof state transformations that can potentially be generated by that tactic. In this work, we approximate that infinite set with a finite dataset of transformations that occur in real proofs written by Coq users. We then use machine learning models to gain an understanding of tactics using their approximated semantics.

As an example, Figure 4.2 presents the before and after states of the tactic `rewrite add_comm` at its position in Figure 4.1. In this particular case, the hypotheses remain unchanged, but in the goal, the two sides of the addition are swapped.

In this paper, we consider the machine learning task of predicting a tactic capable of generating a given proof state transformation and investigate the applications of this task. Formally, given a before state ps and n after states $\{ps'\}_{1..n}$, we attempt to predict a tactic t that transforms ps to $\{ps''\}_{1..n}$ such that ps'_i is equal to ps''_i modulo α-equivalence for every i.

4.2.1 Motivation

Tactic prediction methods have so far relied solely on before states, typically to guide automated tactical proof search in systems like Tactician [BUG20b]. We are interested in synthesizing tactics based both on the before and after states for a number of reasons.

First, there are multiple interesting applications of this task. For example, formalizers may want to arrive at a particular proof state, given a particular initial proof state. Or, given particular before and after states that were generated with a sequence of tactics, we may want to find a *single* tactic capturing the transformation, thus shortening and simplifying the proof, and teaching the formalizer how to use the available tactics.

Second, our work is the first step to designing a novel human-like proof search strategy. When mathematicians write pencil-and-pen proofs, they often first imagine some intermediate goals and then sequentially fill in the gaps. This provides another motivation: our trained predictors can recommend the tactics that will bridge the gaps between such intermediate human-designed proof goals.

Third, the task can be of particular importance for the ITPs which support constructing proofs in a declarative proof style, such as Isabelle, Mizar, and Lean. In declarative-style proofs often the after states are specified by the user manually. A large formal library, Mizar Mathematical Library [BBG$^+$18], is developed in a declarative style. The Isabelle Archive of Formal Proofs (one of the most developed libraries today) is also predominantly written in a declarative style. Our approach can be directly applied to predict tactics able to fill the gap between two subsequent declarative statements.

Finally, the learned tactic embeddings could be used to perform MuZero-style [SAH$^+$20] reinforcement learning, which means obtaining the after states by combining the embeddings of the before states and of the tactics without actually running the ITP. This could be particularly useful when some tactic applications require large computational resources.

4.2.2 Contributions

The main contributions of our paper can be summarized as follows.

1. To our best knowledge, we are the first to predict tactics based on the transformation they make between before and after states.

2. In Section 4.3, to capture the semantics of tactics, we design three characterizations: feature difference, anti-unification, and tree difference.

3. In Section 4.5, we conduct experiments to verify the strengths of our characterizations with a random forests classifier and the GPT-2 language model.

4. In Section 4.6, we propose and evaluate two applications of the task, namely tactic suggestion and proof shortening.

Besides the above-mentioned contributions, Section 4.4 introduces the preliminaries of the learning technology used in this paper. We discuss two related research fields in Section 4.7. The conclusions and future work are presented in Section 4.8.

4.3 Proof State Characterizations

To train the machine learning models, we need to provide characterizations of the before and after states. Apart from directly using the unprocessed textual representation of proof states, we design three characterizations: feature difference, anti-unification, and tree difference.

4.3.1 Feature Difference

To characterize the proof states, we start with the features used by [ZBP+21]. In that work, the features were used to apply machine learning to predict tactics for proof states. For example, `GOAL-$l'` and `HYPS-Coq.Lists.List.rev-$l'` are two features extracted from the before state in Figure 4.2. The prefixes `GOAL` and `HYPS` denote whether a feature belongs to the goal or the hypotheses. The symbol `$l'` denotes a node that occurs in the abstract syntax tree (AST) of the proof state. The prefix `$` means that `l'` denotes a named variable. We subsequently consider the nodes connected in the AST. For example, the feature `Coq.Lists.List.rev-$l'` means that the identifier of the reversion operation of a list and the list `l'` are connected in the AST.

For the current work, we additionally consider feature difference. From the before state ps and after states $\{ps'\}_{1..n}$, we extract features f and $\{f'\}_{1..n}$, respectively using the procedure discussed above. We define f' as the union of $\{f'\}_{1..n}$. By set difference, we compute the *disappeared features* $f - f'$ and the *appearing features* $f' - f$. The disappeared features and appearing features are together used as feature difference characterization of the tactic.

4.3.2 Anti-unification

Anti-unification, first proposed by Plotkin [Plo71] and Reynolds [Rey70], aims to calculate generalizations of the given objects. Since Coq is based on the Calculus of Inductive Constructions (CIC) [PM15], an appropriate anti-unification algorithm for Coq should be higher-order. However, higher-order anti-unification is undecidable [Pfe91]. Therefore, we first convert Coq terms to first-order terms so that we can execute a decidable and efficient first-order anti-unification algorithm.

To encode Coq terms into first-order logic, we transform them recursively following the AST. First-order applications and constants are encoded directly, other applications use the apply functor `app` and all other cases use special first-order functions (e.g., a dependent product is encoded as a first-order function `prod`). The goal of the before state in Figure 4.2 will be converted to the first-order term $= (+(length(l'), S(O)), S(length(l')))$. The non-leaves $=, +, length, S$ denote function symbols. The leaves l' and O denote constants.

Terms in first-order anti-unification are defined as $t ::= x \mid a \mid f(t_1, ..., t_n)$ where x is a variable, a is a constant, f is an n-ary function symbol, and t is a term. In this paper, letters s, t, u denote terms, letters f, g, h denote function symbols, letters a, b

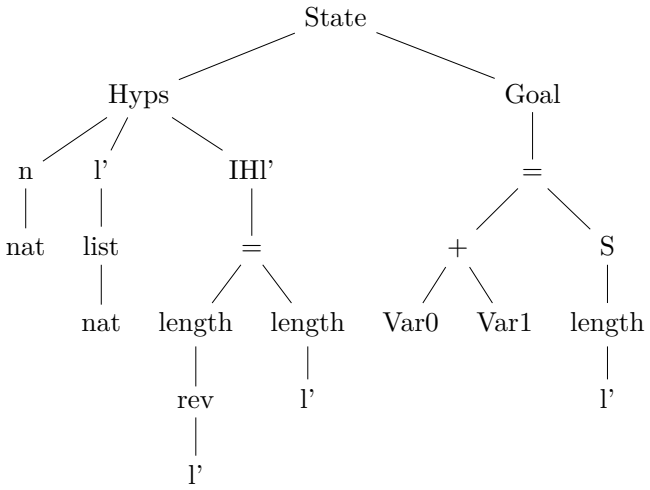


Figure 4.3: The least general generalization of the before and after states in Figure 4.2

denote constants, and letters x, y denote variables. *Substitutions* map variables to terms and are usually written in the form of sets. We can represent a substitution σ as a set $\{x \mapsto \sigma(x) \mid x \neq \sigma(x)\}$ where $\sigma(x)$ is the term mapped by x. The application of a substitution σ to a term t is represented as $t\sigma$. If t is a variable, then $t\sigma = \sigma(t)$. If $t = f(t_1, ..., t_n)$, then $t\sigma = f(t_1\sigma, ..., t_n\sigma)$. A term u is called a *generalization* of a term t if there exists a substitution σ such that $u\sigma = t$. For instance, the term $f(g(x), y)$ is a generalization of the term $f(g(a), h(a, b))$. The substitution σ is $\{x \mapsto a, y \mapsto h(a, b)\}$ such that $f(g(x), y)\sigma = f(g(a), h(a, b))$.

Anti-unification aims to obtain the *least general generalization (lgg)* of two terms s and t. A term u is called a generalization of s and t if there exist substitutions σ_1 and σ_2 such that $u\sigma_1 = s \wedge u\sigma_2 = t$. A generalization u' of s and t is called the lgg if, for any generalization u of s and t, there is a substitution σ, such that $u'\sigma = u$. Assuming ϕ is a bijective function from a pair of terms to a variable, given two terms s and t, the anti-unification algorithm AU calculates the lgg using the two rules below.

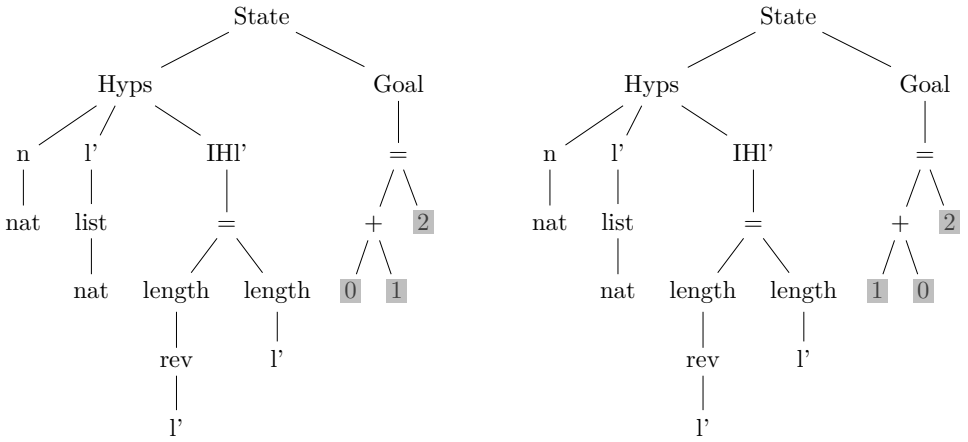
- $AU(s, t) = f(AU(s_1, t_1), ..., AU(s_n, t_n))$ if $s = f(s_1, ..., s_n)$, $t = f(t_1, ..., t_n)$
- $AU(s, t) = \phi(s, t)$ if the preceding rule does not match.

Figure 4.3 presents the lgg of the before and after states considered in Figure 4.2. Compared to the before state, most of the nodes in the lgg remain the same. The differences stay in the left side of the equality in the goal: `length l'` is substituted with `Var0`, and the natural number `1` is substituted with `Var1`. We need to apply the substitutions $\{var_0 \mapsto length\ l', var_1 \mapsto 1\}$ and $\{var_0 \mapsto 1, var_1 \mapsto length\ l'\}$ to the lgg to obtain the before and after states, respectively.

We compute the *lggs* of the goals and the hypotheses separately. We can directly anti-unify the goals of the before and after states. However, the number of hypotheses

may be changed by the tactic application. For instance, the tactic `intros` introduces new hypotheses, while the tactic `clear H` removes the hypothesis `H`. Suppose we are anti-unifying the hypotheses $hyps(h_1, ..., h_n)$ and $hyps(h_1, ..., h_n, h_{n+1})$. The first rule of anti-unification immediately fails, and the second rule will generate a variable that corresponds to all hypotheses in the before state and all hypotheses in the after states. Therefore, anti-unifying all hypotheses together prevents us from developing a compact characterization. To calculate the lggs of hypotheses, we first match the hypotheses with the same names. Then, we compute an lgg on each pair. We refer to the hypotheses that are only in the before state and only in the after state as respectively *deleted hypotheses* and *inserted hypotheses*. Different from the pairwise hypotheses, we do not perform anti-unification on the deleted hypotheses and inserted hypotheses, and they remain unchanged.

We choose anti-unification because it can generate a more compact representation compared with directly utilizing the before and after states. Consider Figure 4.2, we need a Coq string of the before state and another Coq string of the after state to characterize the transformation. Notice that many parts of the before state are unchanged after the tactic application. It is redundant to represent these unchanged parts twice in both the before and after states. However, anti-unification enables us to use a single lgg and the substitutions to characterize the transformation. The unchanged parts of the before and after states are shared in the lgg. Moreover, previous research has demonstrated that features based on generalization are very helpful for theorem proving [KUV15a].



(a) Deletion context

(b) Insertion context

Figure 4.4: The deletion and insertion contexts of the before and after states in Figure 4.2. `Hole0`, `Hole1`, and `Hole2` denote `length l'`, `1`, and `S(length l')`, respectively.

4.3.3 Tree Difference

In addition to anti-unification, we propose a characterization based on a tree difference algorithm [MS19]. Compared to anti-unification, tree difference is better at generalizing the differences between the before and after states. Tree difference extends the standard Unix diff [HM76] algorithm by the capability to compute the differences according to the tree structures. Since proof states have tree structures, such tree differences can be used to characterize the transformations.

Take the before and after states in Figure 4.2 for demonstration. First, for the hypotheses that are the same in the before and after states, we keep them unchanged. Therefore, the hypotheses `n`, `l'`, and `IHl'` remain the same.

The next step is to extract common subtrees from the original trees (except for the unchanged hypotheses) to obtain more compact characterizations. We focus on the ASTs of Coq terms. Assuming there is an oracle to judge whether the current subtree is a common subtree, we traverse a tree from the root. The calculation of the oracle is explained in the original paper [MS19]. If the current subtree is a common subtree and not a leaf node, we substitute it with a hole. We do not substitute leaves with holes because, in practice, the substitutions of leaves lead to many unexpected holes. The same common subtrees should always be substituted with the same hole. The results of applying the substitutions to the before and after states are called the *deletion context* and the *insertion context*, respectively. After the substitutions, the deletion and insertion contexts are shown in Figure 4.4.

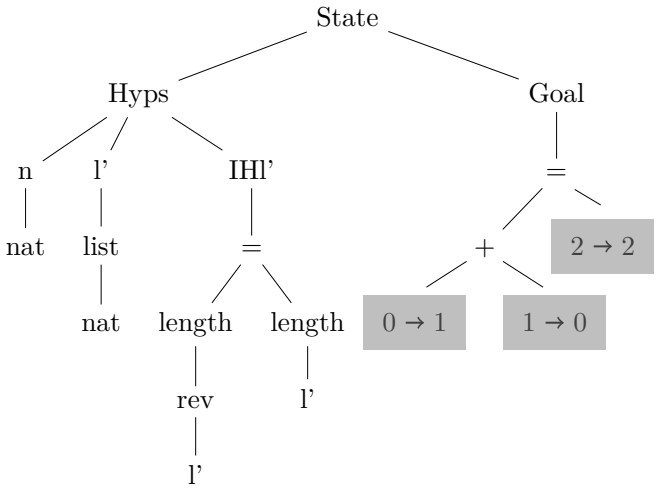


Figure 4.5: The patch of the before and after states in Figure 4.2

Afterward, we calculate the *greatest common prefix (gcp)* of the deletion and insertion contexts and obtain a *patch*. According to the original algorithm, if the two trees have the same non-hole node, we keep the node unchanged and execute the algorithm on their children. Otherwise, we denote them as a *change*.

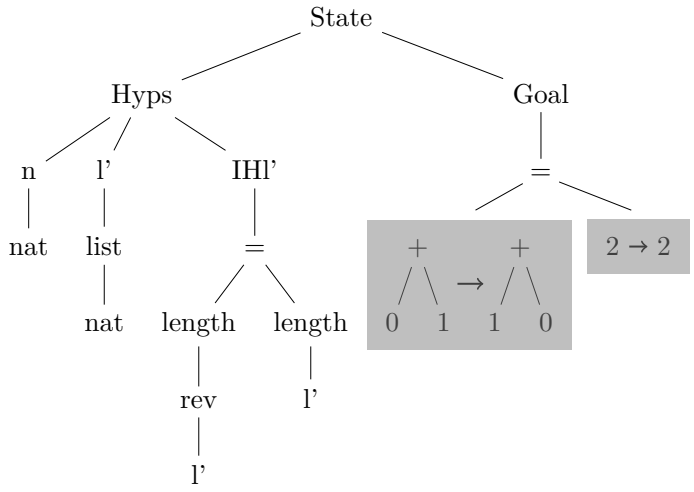


Figure 4.6: The result of applying the closure function to the patch in Figure 4.5

Similar to anti-unification, due to the deletion, insertion, and reordering of the hypotheses, we need to adjust the gcp algorithm for proof states. We match hypotheses by their names and obtain the deleted hypotheses, inserted hypotheses, and matched hypotheses as in Section 4.3.2. We only calculate gcps on the matched hypotheses. The deleted hypotheses and inserted hypotheses are represented as a change. Executing gcp on proof states returns a patch in the format of $state(hyps\_patch, goal\_patch)$ where $hyps\_patch$ is constructed by $hyps(h_1, ..., h_n, change(del\_hyps, ins\_hyps))$. Each h_i is the patch of two matched hypotheses. Figure 4.5 depicts the patch of the before and after states in Figure 4.2.

Subsequently, we need to calculate the *closure* of a patch. The intention is to confirm that every change is *closed*: the left and right sides contain the same holes. Notice that the patch in Figure 4.5 contains two unclosed changes, `Change(Hole0, Hole1)` and `Change(Hole1, Hole0)`. The closure function will go to the subtree, whose root is the parent node of the unclosed change. Then, restore the subtree with the deletion and insertion contexts before we execute gcp on them. The procedure repeats until all changes are closed. Since the gcp function on proof states also returns a patch in a tree structure, we can run the closure function on it. If any patch of matched hypotheses h_i or $change(del\_hyps, ins\_hyps)$ are not closed, we restore the $hyps\_patch$ with the original deletion and insertion contexts of the hypotheses. Then, if the $goal\_patch$ or the deletion and insertion contexts of the hypotheses are not closed, we restore the patch of the proof states with the entire deletion and insertion contexts of the two proof states. Figure 4.6 depicts the patch after the execution of the closure function.

The final step is to replace the identical changes with their origin term. The original algorithm may cause identical changes, such as `Change(Hole2, Hole2)` in Figure 4.6. Since we want a compact characterization, they are not necessary.

Tree difference is better at generalizing the differences compared to anti-unification.

Take the example in Figure 4.2 for instance. The lgg in Figure 4.3 merely shows that the proof state changes in the position of the variables. The substitutions may be different if we execute `rewrite add_comm` on different proof states. However, in the patch generated by the tree difference in Figure 4.6, the changes are generalized because we substitute common subterms with holes and will be the same even if we execute `rewrite add_comm` on different proof states.

4.3.4 Input Formats

During training, the language model receives the string `<Characterization> Tactic: <Tactic>` as input. `<Characterization>` has four variations:

- `Before:<Before State>`
- `Before:<Before State> After:[<After State>]`
- `Anti:[<Substs> <Delete_hyps> <Insert_hyps> <Lgg>]`
- `TreeDiff:[<Patch> <Hole>]`

A proof state is represented as a sequent `<Hyps> |- <Goal>`. The plain texts (like `Tactic:`) serve as prompts, while the placeholders (such as `<Before State>` and `<Tactic>`) are substituted according to the proof context. `[]` denotes a list. During prediction, the language model receives `<Characterization> Tactic:` as input and outputs the predicted tactics.

Random forests are fed discrete features as input. For feature difference, the disappeared features and appearing features are distinguished from each other (appearing features and disappeared features as introduced in section 4.3.1). To utilize anti-unification, we convert the lgg and the terms in the substitution that should be used to obtain the before and after states to features in three disjoint spaces. For anti-unification, we also distinguish the features of deleted hypotheses and inserted hypotheses from other ones. For tree difference, we distinguish the gcp of the proof states, the origin and the destination of changes, and the common subterms into four spaces.

4.4 Learning Models

We consider two machine learning models for the task. The models will be compared experimentally in the next section.

The first model is a random forest classifier [Bre01b]. Random forests are based on decision trees. In decision trees, leaves represent labels (tactics in our case), and internal nodes correspond to features. A rule is a path from the root to a non-leaf. It represents the conjunction of all features on the path. A rule is determined by maximizing the *information gain* of examples. For instance, if we have examples with labels $\{b, b, b, a, a\}$, we want to generate a rule that passes all examples with the label a to its left child and all examples with the label b to its right child. A forest makes predictions by voting based on a large number of decision trees. Random forests contain several sub-forests.

Each sub-forest is built on a random subset of the entire dataset. We choose a random forest implementation that has previously been used to predict tactics for Coq [ZBP+21].

The other used machine learning technique is the pre-trained language model GPT-2 [RWC+19]. GPT-2 is based on neural networks, which consist of many artificial neurons to learn from training data. The self-attention [VSP+17] technique is intensively applied in GPT-2 to differentially weigh every part of the input data. As a language model, GPT-2 predicts the probability distribution of the next word given a sequence of words as the input. GPT-2 is a pre-trained language model. The concept of pre-training imitates the learning process of humans. When humans encounter a new task, humans do not need to learn it from scratch. They will transfer and reuse their old knowledge to learn to solve it. Similarly, GPT-2 is pre-trained on a large natural language dataset BooksCorpus [ZKZ+15]. Afterward, GPT-2 can reuse the knowledge of natural language learned from pre-training to solve new tasks. To be adapted to a new task, we need to fine-tune GPT-2 on a relatively small dataset and slightly modify the weights learned from pre-training. We decide on GPT-2 because pre-trained language models have recently demonstrated outstanding achievements in natural language process (NLP) [BMR+20] and formal mathematics [UJ20, WJL+22].

4.5 Experiments

We perform the experiments on the dataset extracted from the Coq standard library. The dataset consists of 158,494 states extracted from 11,372 lemmas. We randomly split the dataset into three subsets for training, validation, and testing in an 80-10-10% ratio. First, we use 100 trees by default and optimize the Gini Impurity [MM97]. Gini Impurity is a metric of the information gain. After the optimization, we set the Gini Impurity to its best value, try various numbers of trees and obtain the optimized number of trees. Finally, the best combination of Gini Impurity and the number of trees is determined for each characterization. The experiments with GPT-2 are based on the Hugging Face library [WDS+19]. In particular, we employ the smallest GPT-2. The hyper-parameters are: $eta = 3e-4, num\_beams = 3, batch\_size = 32$. During training, we apply a linear schedule with the first 20% training steps for warm-up. The remaining parameters are left as their default values. At most 50 tokens are predicted for a single tactic. We truncate the input on the left side if it is longer than the maximal length limitation of GPT-2 (1024 tokens). Language models have length limitations for efficiency. The attention mechanism used by them causes a quadratic usage of memory as the length of tokens scales. Every model is trained for 25 epochs on an NVIDIA V100 GPU, and the snapshot with the highest accuracy on the validation dataset is selected for testing.

Table 4.1 depicts the results of our experiments. The accuracies of the combinations of before states with after states are significantly better than only relying on the before states in both random forests and GPT-2. Thus, we conclude that taking after states into consideration is very helpful to learn the semantics of tactics. The accuracies of GPT-2 are significantly higher than random forests, which confirms that the pre-trained language model is a more advanced machine learning technique compared to random forests.

Table 4.1: Results on the test dataset, showing how often the prediction makes the same transformation as the tactic in the library. The transformations are considered modulo α-equivalence.

	random forests	**GPT-2**
before	43.23%	46.84%
before after	52.17%	67.45%
feature difference	59.34%	–
anti-unification	58.59%	71.74%
tree difference	58.98%	73.83%

For random forests, all of the feature difference, anti-unification, and tree difference perform better than the unprocessed before and after states. This indicates that our characterizations can extract more precise features for random forests. We do not apply GPT-2 to feature differences, as it relies on natural language. In principle, it would be possible to give it feature differences directly as input, but as there are very few similarities between features and natural language it would be a serious disadvantage to the model. The knowledge grasped by pretraining is difficult to be used to understand features. Although feature difference is a little better than anti-unification and tree difference, their results are quite similar. The probable explanation is that random forests are not good at learning from sophisticated features. Random forests cannot learn meaningful knowledge from all three characterizations and almost only learn to make correct predictions for the simple tactics. Similarly, with GPT-2, anti-unification and tree difference provide more accurate predictions than the unprocessed before and after states. We suppose the explanation is that we are able to appropriately shorten the length of the input and also keep important information about the proof transformation. Appropriately shortening the input length is beneficial for GPT-2 because it has a maximal limitation on the number of input tokens. Table 4.2 compares the percentages of the inputs that are longer than the maximal length limitation. The statistics show that our implementation significantly reduces the probability that the input is over the maximal length limitation. Tree difference can provide more accurate predictions compared to anti-unification with both random forests and GPT-2. This may be attributed to that the generalization made by tree difference is easier to learn by machine learning models.

Table 4.2: The ratios of how many inputs exceed the maximal length limitation

	before	**before after**	**anti-unification**	**tree difference**
ratio	2.07%	7.96%	4.07%	3.90%

Table 4.3: The first five tactics suggested by each characterization. The tactics displayed in bold result in the desired after states.

	before	before after	anti-unification	tree difference
1	trivial	**rewrite <- minus_n_O**	**rewrite <- minus_n_O**	**rewrite sub_0_r**
2	simpl	**rewrite sub_0_r**	**rewrite Nat.sub_0_r**	**rewrite Nat.sub_0_r**
3	**rewrite <- minus_n_O**	rewrite <- minus_n_O	simpl	simpl
4	rewrite <- plus_n_O	simpl	**rewrite sub_0_r**	rewrite <- sub_0_r
5	auto	rewrite <- sub_0_r	rewrite <- plus_n_O	apply sub_0_r

4.6 Applications

In this section, we propose two promising applications of the task. We only evaluate the most accurate of the methods proposed in the previous Section 4.5 (GPT-2) on the two tasks.

The first, more direct application, is making tactic suggestions. Given a before state, it is common for an ITP user to have an intuition of the intermediate proof states that are necessary to complete the proof. However, sometimes the user cannot guess the appropriate tactic needed to make the transformations. Using our model with the before state and the imagined intermediate states, the user can get a complete proposed proof as output. Hence, our model will predict the likely tactics to perform the transformations.

The other application is shortening existing Coq proofs. Specifically, for the transformation $ps_0 \Rightarrow_{t_0} ps_1 \Rightarrow_{t_1} ps_2 ... \Rightarrow_{t_n} ps_{n+1}$, where ps is a proof state and t is a tactic, we want to predict a tactic t' such that $ps_0 \Rightarrow_{t'} ps'$ where ps' and ps_{n+1} are equal under α-equivalence. Thus, we can replace the tactic sequence with a single tactic and decrease the length of the Coq proof. A restriction for this task is that because we are only interested in exploring shorter paths between proof states, ps_{n+1} should not be a finishing state.

4.6.1 Tactic Suggestion

We view the experiments in Section 4.5 as the evaluation of tactic suggestions. The before and after states extracted from the Coq standard library are considered as the states that are presented in the Coq editor and those in users' minds, respectively. The results show that taking the after states into consideration, together with the more compact characterization, is essential for correctly suggesting tactics.

The following is an actual tactic suggestion question taken from the Coq Discourse

Table 4.4: The shortening ratios and amounts of redundant tactics with different characterizations and sequence lengths.

length		before	before after	anti-unification	tree difference
2	ratio	0.379%	0.824%	0.891%	0.833%
	number	215	468	506	473
3	ratio	0.039%	0.148%	0.151%	0.148%
	number	22	84	86	84

Forum[1]. The question can be summarized as finding a tactic that transforms the following before state to the after state. The goal of the before state is to prove that the element indexed by $m - 0$ in a list equals the element indexed by m.

- Before state: `l : list nat, x:nat, m : nat, H0 : 1 <= m |- nth (m - 0) l 0 = nth m l 0`
- After state: `l : list nat, x:nat, m : nat, H0 : 1 <= m |- nth m l 0 = nth m l 0`

Table 4.3 shows the first five tactics predicted by each model. If we consider only the before state, we will obtain the correct prediction in the third place. However, the first two synthesized tactics using anti-unification, tree difference as well as unprocessed before and after states are appropriate. Besides the tactics displayed in bold, other tactics do not perform the expected transformation due to various reasons. Some tactics such as `trivial`, `simpl`, and `auto` do not change the proof state. The tactics `rewrite <- plus_n_O` and `apply sub_0_r` are not applicable and cause errors. The lemma `minus_n_O` used in `rewrite <- minus_n_0` does not exist in the Coq standard library. Although `rewrite <- sub_0_r` does not cause an error, it leads to an unexpected after state `l : list nat, x:nat, m : nat, H0 : 1 <= m |- nth (m - 0) l 0 = nth m l 0 - 0`. Since the operations executed by `trivial`, `simpl`, and `auto` are quite complicated and may depend on the context, we assume it is difficult for the model to comprehensively understand them. Their occurrences in the first five predictions may be mainly because they occur quite frequently in the training data. The results confirm that the combination of before and after states is beneficial for suitably suggesting tactics.

4.6.2 Shortening Proofs

The results presented in the previous Section 4.5 focused on decomposed tactics. This means compound tactic expressions that perform several steps at once have been decomposed into individual tactic invocations. We apply the technique that is developed by [BUG20a] to decompose the tactics. Here, we utilize the same models; however, we focus on the original human-written tactics and try to shorten these (shortening expanded

[1] https://coq.discourse.group/t/how-to-avoid-awkward-assertions/1153/2

tactics would be unfair). For all tactic sequences of lengths two and three in the training dataset, we input their before and after states into the model. In our experiment, we can only consider the states in the training dataset since our model is trained on all present tactics. Compared to the validation dataset and testing dataset, our model should be able to give better predictions on proof shortening for the training dataset. The amount of original tactics in the training dataset is 56,788. The model synthesizes 10 tactics for each sequence, and we execute them in Coq to verify that they perform the same transformation as the sequence modulo α-equivalence.

The results are presented in Table 4.4. We define the number of *redundant tactics* of $ps_0 \Rightarrow_{t_0} ps_1 \Rightarrow_{t_1} ps_2 ... \Rightarrow_{t_n} ps_{n+1}$ as n. The *shortening ratio* is defined as the number of all discovered redundant tactics divided by the total number of occurrences of tactics in the training dataset. In this section, our method only applies to a tactic sequence that, besides the last tactic, every intermediate tactic produces a single after state. While in Section 4.5, our experiments apply to tactic applications that may produce several after states. The reason is that it is difficult to calculate the number of redundant tactics if intermediate tactics produce several after states. The tactic sequence will become a tree of tactics, and each path consists of a sequence of tactics. We initially expected that the shortening ratios would not be very high because of the selected dataset. Indeed, the Coq standard library is written by Coq experts and has been edited and improved for decades, so we expected that there is not much room to improve. However, given the size of the dataset, the proposed technique can find a number of redundant tactics, which lets us conclude that taking the after states into consideration is useful for proof shortening.

We discover many interesting cases, where proofs can be optimized. We present two examples of such proofs in Table 4.5. The first is about the Riemann integral where *ring* and *field* denote algebraic structures. The Coq user first substituted a subterm in the proof state, rewrote the goal by several lemmas, and finally applied a lemma about rings. However, our model discovers the non-trivial transformation on ring can be completed with a single transformation in field.

In the second example, the Coq library authors first applied the lemma `Qle_lteq` to transform the goal into a disjunction. Later, they selected the left side of the disjunction to continue the proof. Our model is able to figure out that the operation is redundant. Indeed it finds another lemma `Qlt_le_weak` that is able to immediately transform the goal to the left part of the disjunction.

In addition to such more impressive examples of simpler, shorter proofs, our model is also able to find a few abbreviations. Such abbreviations make the proof shorter but do not necessarily improve their readability. For instance, our model sometimes combines `unfold Un_growing` and `intro` into `intros x y P H n`. It uses the implicit mechanism of `intros` to unfold `Un_growing`. However, a Coq user will not be able to understand what operation `intros x y P H n` conducts without actually executing the Coq script.

Table 4.5: Two examples of shortening of proofs using the prediction. In both of the presented cases, a single tactic provides an equivalent transformation as a sequence of tactics. Since the hypotheses are not changed in any of the presented examples, we omit them and only present the goals for simplicity.

1	`field` makes the same transformation as (`Tactic1. Tactic2.`)
State	`1 = (x - (x + h0)) * - / h0`
Tactic1	`replace (x - (x + h0)) with (- h0); [ | ring ]`
State	`1 = - h0 * - / h0`

Tactic2	`rewrite Ropp_mult_d istr_l_reverse;`
	`rewrite Ropp_mult_distr_r_reverse;`
	`rewrite Ropp_involutive; apply Rinv_r_sym`

State	`h0 <> 0`

2	`apply Qlt_le_weak` makes the same transformation as (`Tactic1. Tactic2.`)
State	`(Qabs (xn p - yn q) <= 1 # z * k)%Q`
Tactic1	`apply Qle_lteq`
State	`(Qabs (xn p - yn q) < 1 # z * k)%Q` ∨ `(Qabs (xn p - yn q) == 1 # z * k)%Q`
Tactic2	`left`
State	`(Qabs (xn p - yn q) < 1 # z * k)%Q`

4.7 Related Work

Several problems originating in formal mathematics and theorem proving have been considered from the machine learning point of view. One of the most explored ones is premise selection [AHK+14a]. The goal of this task is to find lemmas in a large library, that are most likely to prove a given conjecture. For premise selection, the meaning of dependency in formal mathematics has been explored using both approaches that try to explicitly define the logical semantics [KUV15a], as well as approaches that use deep learning for this [WTWD17]. Next, it is possible to apply machine learning to guide inference-based theorem provers. As part of this task, implicitly the meaning of provability and step usefulness are derived by the learning methods. This has been explored in the two top-performing first-order theorem provers [Sud21, JU17] as well as in higher-order logic automated theorem proving [FB16]. Similarly, the meaning of the usefulness of a proof step has been considered, for example as part of the HOLStep [KCS17], where various machine learning methods try to predict if particular inferences are needed in a proof. All these tasks are different from the task that we propose in the

current paper.

Various proof automation systems have emerged to construct proofs by tactic prediction and proof search. SEPIA infers tactics for Coq by tactic trace and automata [GWR15]. TacticToe [GKU+21a] and Tactician [BUG20a, ZBP+21] apply classical statistical learning techniques such k-nearest neighbors [Dud76] and random forests [Bre01b] to generate tactic predictions based on the before states. Several systems use neural networks for the same task, e.g. HOList [BLR+19a], CoqGym [YD19], and Lime [WRL+21]. These are all different from the current work that considers the after states as well.

Autoformalization [KUV17b] is a machine translation task applied to formal mathematical proofs. The accuracy of the best methods applied to the task is still very weak in comparison with human formalization [WBKU20b], however, the neural methods already show some minimal understanding of the meaning of formalization, for example by finding equivalent formulations. Again this is a different task from the one considered in the current work.

4.8 Conclusion

In this paper, we propose a new machine learning task, with which we aim to capture the semantics of tactics in formal mathematics. Based on a dataset of almost 160 thousand proof states we consider synthesizing a tactic that transforms a before state to the expected after states. We implement three novel characterizations to describe the transformation: feature difference, anti-unification, and tree difference. The results of the experiments confirm the effectiveness of our characterizations. Two applications of the task are discussed: tactic suggestion for declarative proofs and proof shortening.

In the future, we will investigate if tactic embeddings can be used directly. We can also try to estimate the after states by calculating the embeddings of the before state and the tactic or align tactics between systems in a similar way to how concepts are already aligned between systems [GK19].

Acknowledgements This work was partially supported by the ERC Starting Grant *SMART* no. 714034, the ERC Consolidator grant *AI4REASON* no. 649043, the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466, the Cost action CA20111 EuroProofNet, the ERC-CZ project *POSTMAN* no. LL1902, Amazon Research Awards, and the EU ICT-48 2020 project TAILOR no. 952215.

Chapter 5

Learning Rules Explaining Interactive Theorem Proving Tactic Prediction

5.1 abstract

Formally verifying the correctness of mathematical proofs is more accessible than ever, however, the learning curve remains steep for many of the state-of-the-art interactive theorem provers (ITP). Deriving the most appropriate subsequent proof step, and reasoning about it, given the multitude of possibilities, remains a daunting task for novice users. To improve the situation, several investigations have developed machine learning based guidance for *tactic* selection. Such approaches struggle to learn non-trivial relationships between the chosen tactic and the structure of the proof state and represent them as symbolic expressions.

To address these issues we (i) We represent the problem as an *Inductive Logic Programming (ILP)* task, (ii) Using the ILP representation we enriched the feature space by encoding additional, computationally expensive properties as *background knowledge* predicates, (iii) We use this enriched feature space to learn rules explaining when a tactic is applicable to a given proof state, (iv) We use the learned rules to filter the output of an existing tactic selection approach and empirically show improvement over the non-filtering approaches.

5.2 Introduction

Interactive Theorem Provers (ITP), such as Coq [The20], Lean [dMKA+15], and Isabelle [Pau94], are powerful tools that combine human instruction with computer verification to construct formal mathematical proofs, providing a reliable means of certification and ensuring safety in critical applications.

These systems operate as follows: the user specifies a goal to prove, *the initial proof state*. Then the user specifies *tactics* (an operation transforming a proof state into proof states). Certain tactics close proof states. The proof is complete if there are no remaining open proof states, i.e., the goal has been proved.

Given the complexity of ITP systems, a fully automated approach to proving user specified goals is intractable. Numerous investigations have instead focused on providing

the user with guidance through tactic suggestion.

The methods used in practice by ITP users are statistical machine learning methods such as k-nearest neighbors (k-NN) and naive Bayes [GKU+21a]. These methods take a goal g, select a goal g' most similar goal to g, and rank the particular tactics relevant for solving g' based on their likelihood of solving g.

Neural network and LLM-based approaches addressing the task include: CoqGym [YD19] trains tree neural networks to automatically construct proofs for *Coq*. Thor [JLT+22] combines LLMs and external symbolic solvers to search for proofs for Isabelle. LLMs are also applied to synthesising training data to enhance the performance of theorem proving [X+23]. Despite showing slight improvement in performance during machine learning evaluations, in practice these methods require long training for each new theory, which makes them less useful for day to day proof development.

Additionally, they lack interpretability. When a user receives predictions, they may want to know why a particular tactic was chosen over another tactic to better understand what actions they should take in the future.

Furthermore, guidance based on statistical learning approaches often requires propositionalisation of features, calculated based on the structure of the *abstract syntax tree (AST)* of a proof state [ZBP+21], e.g., *there is a path between nodes X and Y in tree T*. For complex and precise features, pre-computation is prohibitively expensive.

Moreover, logical inference is significantly influenced by the small error margins present in the statistical inferencing mechanisms of LLMs and similar models. Thus, predictions based on chained logical inferences will quickly suffer a loss of predicative accuracy [LeC23].

In contrast to pre-computed features, we represent such features as logic programs and compute them only when needed for learning. For example, we define logic programs for the existence of two particular nodes on a path (of arbitrary length) from the root of the tree as $(above(AST, X, Y))$. Below, we present a learned rule for the simplification tactic which states that the tactic is applicable to a proof state when the goal node of the proof state contains a constant above two constructs (also in the goal) which differ.

```
tac(A,"simpl") :-
  goal_node(const,A,B,C), goal_node(construct,A,D,E),
  goal_above(A,B,D), goal_node(construct,A,F,E),dif(F,D),
  goal_above(A,B,F).
```

The rules, as presented above, are learned using inductive logic programming (ILP), in particular, *Aleph* [Sri01]. In addition to providing rules explaining tactic prediction, we use the resulting rules to filter the output of k-NN, in particular, the classifier presented in [BUG20b, GKU+21a] (Tactician and TacticToe). Essentially, we want to determine whether $ps, r \vDash p_t$ where ps is a logic program representing the proof state, r is a learned rule for the tactic t, and p_t is the head predicate of r denoting that t should be applied to ps. Thus, given the list of recommended tactics by a k-NN classifier, we can further filter this list using the learned rules. Our hypothesis is that features of proof state defined through logic programs can be used to learn rules which can be used to filter the output of a k-NN model to improve accuracy.

In addition to improved performance, our approach produces rules to explain the predictions. Consider again the aforementioned rule of `simpl` that specifies that the goal may be simplified if it contains a constant above two constructors with different positions. Here, the constructor and the constant denote the datatypes of Coq's terms. The same variable E confirms that the two constructors must correspond to the same identifier in Coq. This rule may suit the Coq structure $S\ x - S\ y$ which denotes $(1 + x) - (1 + y)$. It can be simplified to $x - y$. S denotes a constructor, and $-$ denotes a constant. The first argument of `goal_node` is a constant that is constrained by us via mode declarations [Sri01].

We use the ILP system Aleph [Sri01] together with a user-defined cost function to evaluate the learned rules on the Coq standard library. We chose Aleph because it has empirically good results [CD22]. We refrain from using modern ILP approaches such as *Popper* [CM21] as the underlying ASP solvers have difficulty generating models when many variables are required and high-arity definitions are included in the background. We develop representation predicates (`goal_node`) to efficiently denote the nodes of the AST. We also develop feature predicates (`goal_above`) which denote the properties of the AST calculated based on the representation predicates. The motivation for developing feature predicates is that propositionalization of it would significantly enlarge the representation making it impractical to use. Our experiments confirm that feature predicates can learn more precise rules (rules with higher F-1 scores [S$^+$07]) compared to representation predicates. Additionally, the experiments demonstrate that the combination of ILP and k-NN can improve the accuracy of tactic suggestions in Tactician, the main tactic prediction system for Coq.

Our contributions can be summarized as follows:

- First, we express the task of predicting the best tactic to apply to the given proof state as an ILP task.
- Second, using the ILP representation we enriched the feature space by encoding additional, computationally expensive features as *background knowledge* predicates, allowing us to avoid grounding the features which are computationally expensive.
- Third, We use this enriched feature space to learn rules explaining when a tactic is applicable to a given proof state and filter the output of an existing tactic selection approach using these rules.
- Finally, We empirically show improvement over the non-filtering approaches.

This is the first time an investigation has considered ILP as a tool for improving tactic suggestion methods for ITPs.

5.3 Background

5.3.1 Theorem Proving in Coq

Coq is one of the most popular proof assistants and has been widely used for building trustworthy software [Ler21] and verifying the correctness of mathematical proofs [G$^+$08].

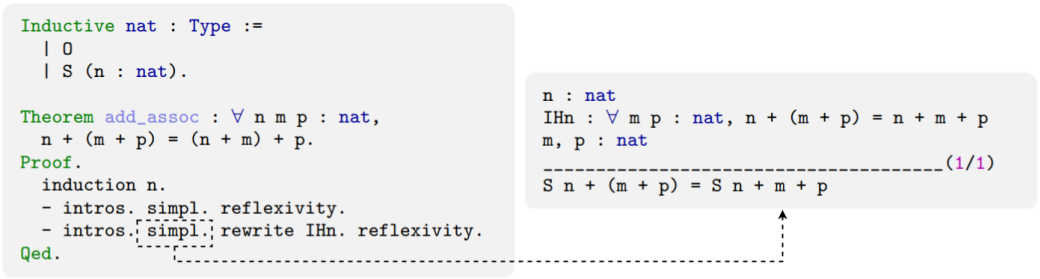


Figure 5.1: A Coq proof of the associative property of addition and the proof state before `simpl`.

Coq tactics are proof state transformations that provide a high-level combination of underlying logical inferences. To illustrate how theorems are formalized in Coq, we present a simple example in Figure 5.1. Here, we want to prove the associative property of addition. The natural numbers in Coq are defined by two constructors `O` and `S`. `O` denotes 0, and `S n` denotes $n+1$. Here, the initial proof state is the same as the statement of the theorem. We first apply induction on `n` and obtain two cases corresponding to the two constructors. In the first case, `n` equals 0. After some simplifications, we can prove $0 = 0$ by the tactic `reflexivity`. The second case is a bit more complicated, and we need to apply the induction hypothesis `IHn` to finish the proof. Figure 5.1 also presents a concrete example of a proof state. A proof state consists of a *goal* to prove and several *hypotheses*. The goal is below the dashed line. `IHn`, `n`, `m`, and `p` are the names of hypotheses. A proof state is often represented as a sequent $E \vdash g$ where E and g denote the hypotheses and the goal, respectively.

5.3.2 k-NN adaptations to theorem proving

Several machine learning algorithms have been adapted to theorem proving tasks. In most cases, simpler algorithms adapted to formal reasoning tasks perform better than deep learning based methods in practice. For this reason, modified k-NN (explained in [BUG20a]) is the main algorithm for TacticToe and Tactician. Even if evaluations with deep learning or large tree-based classifiers have shown some theoretical improvements, the simpler algorithms training for the particular theories developed by users, give a larger practical advantage to the users. As such, we focus on the modified k-NN in this work. Standard k-NN starts by calculating the distance between a new proof state and all known proof states in the database. The distance is measured by the similarity between the features of the proof states, usually using tree walks in the AST of the proof state [KUV15c]. The dependencies of such selected neighbours, with additional scaling by their distances, inclusion of the neighbours themselves, and further modifications exhibit commendable empirical performance [R$^+$24], and are therefore the default algorithm both in Tactictoe and Tactician.

5.4 Background Knowledge

To utilize ILP, we need to appropriately define the background knowledge. We start this section by encoding the nodes of AST as representation predicates. Then we propose new feature predicates that will allow leveraging the power of ILP. We finally add predicate anonymization, already very useful in automated reasoning systems, to representation and feature predicates.

5.4.1 Representation Predicates

Every node in the AST of the proof state is converted to a fact. There are two categories of nodes: identifiers of existing objects and constructors of Coq's datatype. A node in the goal is converted to $goal\_node(name, nat, goal\_idx)$. The argument $name$ refers to the value of the node. A unique natural number is assigned to every proof state to identify it. The argument $goal\_idx$ uses a sequence of natural numbers to specify the position of the node in the goal. A node in a hypothesis is converted to a fact $hyp\_node(name, nat, hyp\_name, hyp\_idx)$. Compared to a $goal\_idx$, a hyp_idx starts with the name of a hypothesis so that two hyp_idx from different hypotheses have different prefixes. The $goal\_node$ and hyp_node predicates are called *representation predicates* in this work.

5.4.2 Feature Predicates

We also define two categories of *feature predicates* which represent the properties of AST based on the representation predicates.

Positional Predicates represents the relative relationships between nodes' positions. The predicate `goal_left(Goal_idx1, Goal_idx2)` and `goal_above(Nat, Goal_idx1, Goal_idx2)` respectively checks whether the node is left (above) to another node in the goal. They are inspired by the horizontal features and vertical features used in previous works [CJSU19b, ZBP+21]. Similarly, we define `hyp_left(Hyp_idx1, Hyp_idx2)` and `hyp_above(Nat, Hyp_idx1, Hyp_idx2)`. Previous works have confirmed the usefulness of using the occurrence numbers of features in feature characterization, which inspires us to develop the predicate `dif(Goal_idx1, Goal_idx2)`. It denotes that the same node multiply occurs in different positions in the goal.

Equational Predicates check the equality between two terms. The predicate `eq_goal_term(Nat, Goal_idx1, Goal_idx2)` checks that the two subterms in the goal are the same. The root nodes of the two subterms are located in the positions `Goal_idx1` and `Goal_idx2`, respectively. It pertains to `reflexivity` which proves a goal of the equation if the equality holds after some normalization. Thus, it can prove $x = x$ and inspires us to develop `eq_goal_term`. The predicate `eq_goal_hyp_term(Nat, Goal_idx, Hyp_idx)` is inspired by a number of tactics that check the equality between the goal and the hypotheses, such as `assumption`, `apply`, and `auto`. For instance, `assumption` proves a goal

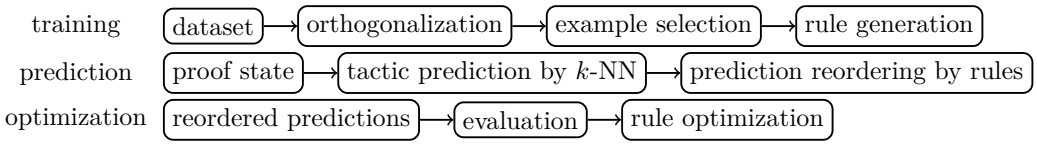


Figure 5.2: An overview of the procedures of the learning framework.

if it equals a hypothesis. Assume a proof state $H_1 : Q\ x, H_2 : P\ x \to Q\ x \vdash Q\ x$ which can be proved by `assumption`. The predicate `eq_goal_hyp_term` checks the equality between the goal and $Q\ x$ in a hypothesis. The predicate `is_hyp_root(Nat, Hyp_idx)` ensures the node is the root of a hypothesis. Thus, it can show the equality only holds between the goal and H_1 instead of H_2. With a reason akin to that of `is_hyp_root`, we define `is_goal_root(Nat, Goal_idx)`. The equality between two terms in different hypotheses is checked by `eq_hyp_term(Nat, Hyp_idx1, Hyp_idx2)`. It is useful for tactics that can apply hypotheses several times, e.g., `auto`. Assume a proof state $H_1 : P\ x, H_2 : P\ x \to Q\ x \vdash Q\ x$. First, `auto` applies H_2 to the goal and changes the goal to $P\ x$. Then, it applies H_1 to prove the new goal. The description of the operation requires to show that H_1 equals to the premise of H_2.

5.4.3 Anonymous Predicates

We also substitute identifiers with more abstract descriptions to facilitate the generalization ability of ILP. The substitution is similar to that in ENIGMA anonymous [J+20]. The predicates that accept original nodes and abstract nodes as their first arguments are called *original predicates* and *anonymous predicates*, respectively. We substitute identifiers with their categories, consisting of inductive types, constants, constructors, and variables. Besides the abstract nodes, we also include the original nodes as arguments in `goal_node` and `hyp_node`. We need them because when checking the equality, we want to compare the original nodes. Afterward, the anonymous predicates of nodes change to $goal\_node(anonym\_name, nat, goal\_idx, origin\_name)$ and $hyp\_node(anonym\_name, nat, hyp\_name, hyp\_idx, origin\_name)$. Some basic identifiers are not substituted, which consist of `logic_false`, `logic_true`, `and`, `or`, `iff`, `not`, `eq`, `bool_true`, and `bool_false`. There are both logic and boolean values of true and false because Coq can represent objects in logic or programs. Concerning the constructors of Coq's datatypes, we only retain four important constructors: `rel`, `prod`, `lambda`, and `evar`.

5.5 Method

Figure 5.2 presents an overview of our learning framework. During the training, we first perform orthogonalization, a technique introduced in TacticToe, to clean the dataset. Then, we select examples and apply ILP to generate rules. To make predictions, first, k-NN predicts a sequence of likely helpful tactics. Afterward, the rules are used as a filter

to reorder the predictions. The optimization procedure denotes removing some low-quality rules. This is achieved by evaluating the reordered predictions in the validation dataset and removing the low-quality ones. In the next subsections, we describe these parts.

5.5.1 Orthogonalization

In some cases, different tactics could transform the same proof state in the same way. This raises ambiguity and makes learning difficult. Orthogonalization is used to reduce such ambiguity. In the orthogonalization, we only focus on four very popular tactics in the Coq standard library: `assumption`, `reflexivity`, `trivial`, and `auto`. We denote the sets of proof states which can be closed by `assumption`, `reflexivity`, `trivial`, and `auto` as AS, R, T, and AT, respectively. There exist the relations $AS \subsetneq T$, $R \subsetneq T$, and $T \subsetneq AT$. For each proof state ps to which the tactic t is applied, the above four automation tactics are sequentially tried. If ps can be finished by the automation tactic t', we replace t by t'. If none of the four tactics can finish the proof state, the original t is preserved. The orthogonalization procedure is simpler than in TacticToe, which orthogonalizes all tactics. This is because our current predicates can only capture a part of the usage of tactics. We leave full orthogonalization as future work.

5.5.2 Example Selection

Choosing appropriate training examples is crucial for learning reasonable rules. For a specific tactic *tac*, the proof states to which it is applied are regarded as the positive examples. The proof states to which the tactics different from *tac* are applied are regarded as the negative examples. We experimentally determine the number of positive and negative examples for learning rules. We develop a clustering mechanism to split positive examples into roughly equal-sized clusters. We experimentally evaluate the combinations of different numbers of negative examples and different numbers of positive examples.

We choose an implementation of a constrained k-means algorithm [LK18] to split positive examples into clusters of roughly the same size. The original k-means algorithm [HW79] can only split examples into a certain number of clusters. In contrast, constrained k-means can also specify the lower bound and the upper bound of the size of the clusters, which is important to give good sizes of training examples for each ILP learning task.

We apply k-NN to discover negative examples for each positive example. As this pre-processing step is not theorem-proving specific, we use the general k-NN from the scikit-learn library [P+11]. We use the same features as Tactician [ZBP+21]. For each positive example, k-NN calculates the distance between it and every negative example in the training data. Then, we rank the negative examples in an ascending order of distance.

5.5.3 Training and Prediction

For each tactic, we use Aleph to generate ILP rules for each cluster of positive examples and its associated negative examples. Afterwards, all the rules are merged together, and

Algorithm 6 Preselection Reorder

Input: a sequence of tactics $tac_{1..50}$ preselected by k-NN for a proof state
Output: a sequence of tactics which is a reorder of the preselection
$goods \leftarrow [\,]$
$bads \leftarrow [\,]$
for $i \in \{1..50\}$ **do**
  if tac_i is accepted by learned rules **then**
    append tac_i to the end of $goods$
  else
    append tac_i to the end of $bads$
$reorder \leftarrow$ the sequence of appending $bads$ to the end of $goods$
return $reorder$

duplicated rules are removed. Finally, we remove the rules of tactics that are logically subsumed by other rules of the same tactic.

Algorithm 6 illustrates the procedures of making predictions. We use the state-of-the-art k-NN in Tactician. The features are the same as those used in Section 5.5.2. Assume a pair of a proof state and a tactic (ps, tac). To make predictions, first, k-NN preselects a sequence of likely tactics $tac_{1..50}$. For each tac_i, we use the learned rules to determine whether to accept it. During the evaluation, the prediction tac_i is (un)expected if tac_i is (un)equal to tac. If the rules accept (reject) a tactic, the prediction is a declared positive (negative). If the rules reject a tac_i equal to tac, we regard the prediction made by the rules as a false negative (FN). Based on the expected tactics and acceptances, we also obtain true positives (TPs), true negatives (TNs), and false positives (FPs).

5.5.4 Rule Optimization

The idea of rule optimization is to remove some low-quality rules to increase the overall performance of rules. For the evaluation of all rules, we chose the F-1 score as the metric, defined as $\frac{2TP}{2TP+FP+FN}$, because it is a standard metric for evaluating imbalanced data. As an illustration of the imbalance, given a pair of a proof state and a tactic tac, rules make predictions for 50 preselected tactics. However, at most one is the same as tac. If a rule is overly general, which means that the number of FPs introduced by it is much larger than the number of TPs introduced by it, removing it will increase the overall F-1 score.

Although a large number of negative examples prevents generating overly general rules, using them may not produce the best rules for two reasons. First, our background can merely capture a portion of the usage of the tactics; thus, a significantly large number of negative examples cannot produce perfect rules but may produce overly specific rules. Second, some negative examples in our dataset are actually false negatives. A mathematician may be able to choose the next step from a couple of tactics that make different proof transformations. Orthogonalization in Section 5.5.1 can only partially remove such overlaps between tactics, thereby decreasing the number of false negatives.

It is computationally prohibitive to perform full orthogonalization of our data. Observe that, our experiments still show an increase in accuracy in light of the noisy data.

Our approach allows us to learn many rules explaining a particular tactic. Over the training set, some of these rules capture the usage of the given tactic better than others. Before, moving to testing on unseen data, we prune the learn rules and keep only those that performed well on the validation set.

To determine which rules to include, we evaluate the quality of each rule in the validation dataset and remove those with low qualities. The left rules are used for the evaluation on the test dataset. We measure the quality of each rule and remove it if its quality is below a certain threshold. We set different thresholds and choose the threshold leading to the highest F-1 score via experiments. For the metric of the quality of a single rule, we use *precision*, defined as $\frac{TP}{TP+FP}$. Here, FP and TP are produced by a single rule. Precision is a good metric because if a rule is too general, its precision will be low and we will be able to remove it to improve the overall F-1 score.

5.6 Experiments

We conducted the experiments on the Coq standard library [CK18a]. We chose the Coq standard library as the benchmark because it is a standard dataset for evaluating machine learning for Coq. Moreover, it comprises well-crafted proofs developed by Coq experts and has been optimized for decades. The Coq standard library consists of 41 theories and 151,678 proof states in total. The code for this paper is available at https://github.com/Zhang-Liao/ilp_coq.

Most parameters of Aleph were left as default besides three parameters. We set the maximal length of a clause to 1,000, the upper bound of proof depth to 1,000, and the largest number of nodes to be explored during the search to 30,000. We define a cost function similar to the default cost function because, by default, Aleph cannot learn with no negative examples or only one positive example. The user-defined cost function was only used when there were no negative examples or exactly one positive example. We set a timeout of ten minutes for learning.

We conducted the experiments in the transfer-theory setting, which means different Coq theories are used for training, validation, and testing. We use this setting because it simulates a practical application scenario of ILP. Mathematicians develop new theories based on the definitions and proven theorems in the developed theories. To be practically beneficial, ILP should also learn rules from training theories, and the learned rules should help make tactic suggestions for theories that do not depend on the training theories. The training theory should be carefully chosen before conducting experiments. The theory `Structures` was chosen for training because it has a balanced distribution of various tactics. To be consistent with the transfer-theory setting, the testing theories should not depend on `Structures`. From the Coq standard library, we chose all theories which do not depend on `Structures` for testing including `rtauto`, `FSets`, `Wellfounded`, `funind`, `btauto`, `nsatz`, and `MSets`. Afterward, from all the theories that do not depend on the testing theories, we randomly chose five theories: `PArith`, `Relations`, `Bool`, `Logic`, and

Figure 5.3: F-1 scores of different parameters when *qualt* is set to 0, 0.18, or 0.30. *AF*, *AR*, *OF*, and *OR* denote the anonymous feature predicates, the anonymous representation predicates, the original feature predicates, and the original representation predicates, respectively. In the x-axis caption *P* and *N* denote *pos* and *neg*, respectively.

`Lists`, merged as the validation dataset.

5.6.1 Parameter Optimization

In Section 5.5, we introduced three additional hyper-parameters beyond those already present in Aleph. They are the size of the cluster of positive examples (*pos*), the number of negative examples of each positive example (*neg*), and the quality-theshold (*qualt*) below which the rule should be removed.

We evaluated the F-1 scores of different predicate categories with different parameters. There are four predicate categories AF, AR, OF, and OR, respectively denoting the anonymous feature predicates, the anonymous representation predicates, the original feature predicates, and the original representation predicates. *AF* and *OF* contain both representation predicates and feature predicates, while *AF* and *AR* only contain anonymous predicates. We chose *pos* between 0 and 32. For *neg*, we chose it between 0 and 64. For all the combinations of *pos* and *neg*, rules were generated. Afterward, the learned rules were evaluated in the validation dataset. Finally, we calculated the F-1 scores with different values of *qualt*. The range of *qualt* was set between 0 and 0.30, with intervals of 0.06.

Figure 5.3 depicts the F-1 scores when $qualt = \{0, 0.18, 0.30\}$. The significance of *qualt* is evident. When $qualt = 0$, the best F-1 scores of all predicate categories hover around 0.10. The best scores become significantly higher than 0.10 when $qualt = 0.18$. The low F-1 scores of $qualt = 0$ are caused by some overly general rules which are discussed in

Table 5.1: The best parameters of each predicate category.

Parameter	AF	AR	OF	OR
Precision	0.18	0.12	0.18	0.12
Positive	1	16	4	1
Negative	32	1	1	1

Table 5.2: The F-1 scores in the test dataset.

Theory	AF	AR	OF	OR
rtauto	**0.564**	0.401	0.502	0.440
FSets	**0.266**	0.125	0.193	0.144
Wellfounded	**0.229**	0.049	0.134	0.135
funind	**0.545**	0.0	0.0	0.0
btauto	**0.339**	0.125	0.162	0.122
nsatz	**0.164**	0.070	0.163	0.116
MSets	**0.272**	0.084	0.143	0.095

Section 5.5.4. An example of such an overly general rule is provided below, showing the necessity of employing an appropriate *qualt*.

```
tac(A,"reflexivity") :- goal_node(coq_Init_Logic_eq,A,B,C).
```

The above rule denotes that `reflexivity` is appropriate whenever there is an equal sign in the goal. It is too general and irrelevant to the usage of `reflexivity` as explained in Section 5.4.2. With $qualt = 0.30$, the best F-1 scores decrease again. The decline is attributed to the fact that most rules remained by a very high *qualt* are excessively specific, thereby producing a limited number of TPs.

Afterward, we analyze the results obtained with $qualt = 0.18$ since the F-1 scores are notably higher than those with $qualt = \{0, 0.30\}$. None of the predicate categories obtains the highest F-1 score with $pos = 32$. We assume the reason is that an overly large *pos* may gather many irrelevant positive examples, which causes difficulties in choosing negative examples. If two positive examples significantly differ, an appropriate negative example for one of them may be inappropriate for the other. A large value of *neg* generally decreases the F-1 scores of all predicates except for AF. A possible explanation is that too many negative examples cause AR, OF, and OR to learn overly specific rules. Due to the expressivity of AF, it can still learn some reasonable rules.

Table 5.1 displays the optimal parameters of all predicate categories. The generalization does not work well for OF and OR, but already with AF its F-1 score peak necessitates a large *neg*, indicating its superior ability to distinguish positive examples from negative examples and to learn precise rules. Perhaps due to the reason that our background knowledge is incapable of perfectly capturing the usage of tactics, AF also uses $pos = 1$. A small *pos* allows AF to learn many rules for diverse situations. AR requires $pos = 16$ to achieve its peak F-1 score, possibly due to its limitation of representing AST in a highly generalized manner.

The training times for each category of predicates are presented in Figure 5.4. The total training time is calculated by summing the training times for the examples associated with each cluster of tactics. Learning definitions of feature predicates is computationally more costly than learning definitions of representation predicates, likely due to the resource overhead required for the former. Moreover, learning anonymous predicates requires more time than original predicates. We hypothesize that obfuscating some of the structure of the proof terms (when learning anonymous predicates) correlates with an increase in

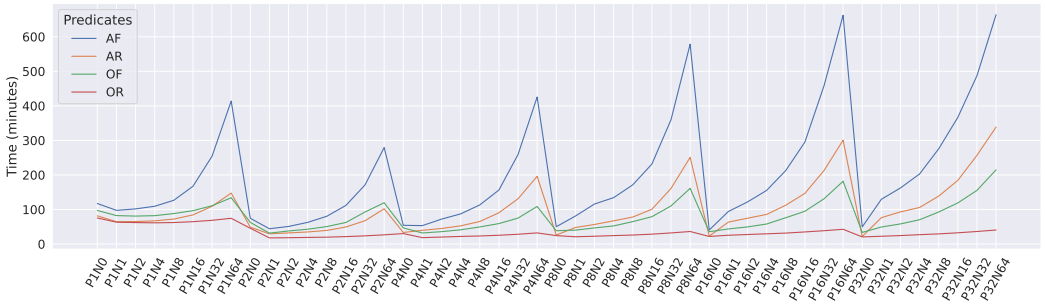


Figure 5.4: Training times for different categories of predicates.

the size of the search space and thus longer learning times when searching for programs distinguishing a large number of positive examples from negative examples. In some cases the learning phase took more than ten hours, indicating potential future work to improve the computational efficiency. Nevertheless, our learning is still significantly faster than contemporary neural networks trained to solve similar problems and, in addition, we provide an explanation of our decisions (in the form of a logic program).

5.6.2 Testing

According to parameter optimization, we choose the rules with the best parameter and test the performance in the test dataset. Table 5.2 shows the F-1 scores in the test dataset. Using a background knowledge consisting of AF predicate definitions during training results in rules which perform best during testing. This owes to that AF can learn precise rules to characterize the usage of tactics. In comparison, the rules learned by AR are too general, and the rules learned by OF and OR are too specific. In all theories, except those consisting of only a few proof states (`funind` has only 14 proof states), training with OF, OR, and AR background knowledge results in rules performing well on the test data F-1 scores. We also evaluated whether the combination of ILP and k-NN can improve the accuracy of k-NN. The algorithm of reordering is explained in Section 5.5.3. Figure 5.5 shows the results of the top-k accuracies in different theories. In all the theories, the combination of ILP and k-NN increases the accuracies of k-NN.

5.7 Case Studies and Limitations

To illustrate that we indeed learn precise rules, besides the example of `simpl` presented in Section 5.2, we present three more examples in this section. The rule of `trivial` suits the goal $A \to B = B$. First, `trivial` introduces A as a hypothesis, changing the proof state to $H : A \vdash B = B$. Next, `trivial` can automatically prove $B = B$. The rule of `auto` aligns the proof state of the structure $H : B \vdash A \vee B$. The tactic `auto` decomposes the disjunction, and the goal changes to either proving A or proving B. Then, it proves B with the hypothesis. In contrast, `trivial` cannot decompose the disjunction. The rule

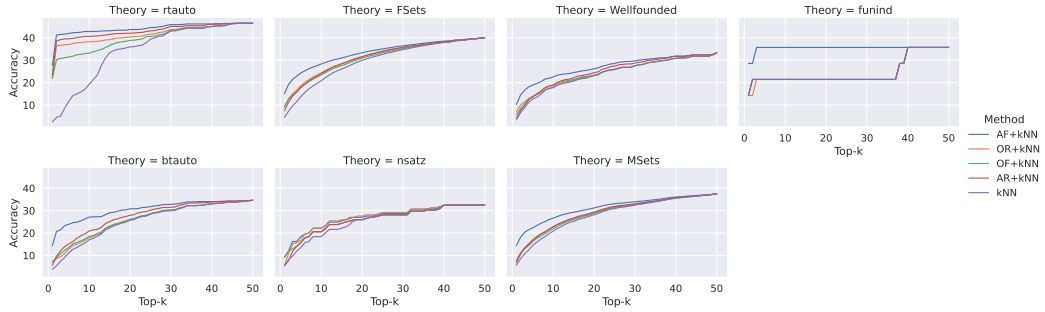


Figure 5.5: Top-k accuracies in the test theories. It denotes how often the label is predicted in the first k predictions. The symbol $+$ denotes using the rules learned by a certain predicate category to reorder the preselections.

of `intuition` suits the goal $A \leftrightarrow A$ which cannot be proved by `auto`. In comparison, `intuition` can perform stronger automation than `auto` and can prove it.

```
tac(A,"trivial") :-
  goal_node(prod,A,B,C),goal_node(const,A,D,E), goal_above(A,B,D),
  goal_node(const,A,F,E),goal_above(A,B,F),eq_goal_term(A,F,D).
tac(A,"auto") :-
  goal_node(coq_Init_Logic_or,A,B,C),goal_node(const,A,D,E),
  goal_above(A,B,D),hyp_node(const,A,F,G,E),eq_goal_hyp_term(A,D,G).
tac(A,"intuition") :-
  goal_node(coq_Init_Logic_iff,A,B,C),goal_node(const,A,D,E),
  goal_above(A,B,D),goal_node(const,A,F,E),eq_goal_term(A,F,D)
```

Albeit we can learn several reasonable rules, many tactics are difficult to describe. There are several reasons for the difficulties. First, our current work cannot generalize tactics with different arguments. For instance, assume there are two tactics `apply H1` and `apply H2` where `H1` and `H2` are names of hypotheses. They are regarded as different tactics but may have the same behavior. Second, the usage of some tactics such as `induction` is inherently complicated [Nag19]. Third, the same mathematical theorem can proved in various ways which leads to many overlaps between the usage of tactics.

5.8 Related Work

There are several tasks of machine learning for theorem proving. *Premise selection* is probably the most well-discovered task. It studies the question of how to predict possibly useful lemmas for a given theorem. Quite a lot of classical learning methods [AHK+14b, GK15b] and neural networks [ISA+16] have been applied to premise selection. The most relevant task to our work is learning-based formal theory proving. Researchers have investigated both employing machine learning to learn from human-written proofs [GKU+21a] and guide some sophisticated software to automatically construct proofs [KUMO18a].

5.9 Conclusion and Future Work

We have developed the first application of ILP to interactive theorem proving. For this, we have developed new feature predicates, able to dynamically calculate features based on the representation of AST of the proof state. We proposed a method for using ILP effectively for tactic prediction. We experimentally evaluated the rules learned by ILP and compared them to practically used prediction mechanisms in ITPs. The experiments confirm that the method gives explainable tactic predictions. Our work shows the potential of applications of ILP to improve ITP tactic suggestion methods.

Several improvements are possible. We would like to use our work with stronger ILP systems, such as Popper. However, given that our background knowledge includes predicates with high arity and our method builds large rules with many variables, the underlying ASP (SAT) solver used by Popper struggles with the generation of models. Improvements to our encoding and recent work on improving the performance of Popper can make this research direction viable in the near future. Next, it is interesting to use ILP to capture the relations between arguments of tactics and the objects to which the arguments refer. Finally, we plan to investigate the application of ILP to other ITP tasks [ZBKU23].

Acknowledgments This work was supported by the Czech Science Foundation Grant No. 22-06414L, the Cost action CA20111 EuroProofNet, the ERC PoC grant no. 101156734 *FormalWeb3*, Amazon Research Awards, the EU ICT-48 2020 project no. 952215 TAILOR, the ERC CZ project no. LL1902 POSTMAN, and the University of Innsbruck doctoral scholarship *promotion of young talent*.

Chapter 6

Conclusion

6.1 Summary

This dissertation exhibits various approaches aimed at enhancing proof automation for the ITP, with a specific focus on the Coq proof assistant.

I have developed innovative features and several learning models for tactic-based ITP guidance within the Coq proof assistant. These features extend those used in Tactician, an existing proof automation system for Coq. The learning models implemented include locality-sensitive hashing forests, online random forests, and gradient-boosted trees. Empirical results demonstrate that the newly developed features and models outperform Tactician's original features and the original k-NN model.

In addition, I have proposed a novel task—learning proof transformations—which closely aligns with tactic-based ITP guidance. I have introduced three distinct characterizations for proof transformations: feature difference, anti-unification, and tree difference. Two key applications have been identified for this task: reducing proof length and providing tactic suggestions. Experimental results validate the accuracy of these characterizations and their practical applications.

Furthermore, I have applied ILP to infer rules for determining when specific tactics should be employed. I have designed an approach where these learned rules serve as filters to reorder tactic predictions generated by statistical learning algorithms. Several predicates have been introduced to ensure precise rule learning, with experiments confirming their ability to improve prediction accuracy through tactic reordering using ILP rules.

6.2 Future Work

6.2.1 Stronger Online Learning

Online learning plays a pivotal role in developing tactic-based ITP guidance. This is because the construction of formal proofs often requires incorporating local hypotheses, recently proven lemmas, newly defined tactics, and proof patterns from nearby theorems, all of which may be relevant to the current theorem under consideration.

Despite its importance, research into online learning for ITP remains underexplored. While we have developed two models in the dissertation—locality-sensitive hashing forests and online random forests—these approaches face certain limitations. First, they are

not widely considered state-of-the-art in the machine learning community, where neural networks and LLMs are preferred due to their versatility across diverse domains. Second, the techniques underlying our models are outdated, originating from methods developed over two decades ago.

Limitations are also apparent in other research on online learning for ITP. For instance, Graph2Tac [BOR+24] decomposes unseen definitions into sub-components and predicts embeddings using those of known sub-components. However, this domain-specific approach contrasts with the broader focus of the online learning community, which seeks to address challenges at a foundational level by enhancing learning techniques, such as modifying cost functions or altering model structures [HSLZ21]. These generalized approaches can be applied across a wide range of tasks, whereas Graph2Tac's method requires specialized decomposition and network architecture tailored to the problem domain.

In the future, I aim to integrate advanced online learning methods into tactic-based ITP guidance. Given the complexity of both online learning and theorem proving, further advancements in online learning algorithms may be necessary to achieve significant improvements in performance.

6.2.2 Neuro-Symbolic Learning for Proof Automation

Symbolic learning may be essential for advancing robust tactic-based ITP guidance, given its reliance on sophisticated logical reasoning. While neural networks and LLMs have demonstrated remarkable success across numerous research domains, they continue to exhibit limitations in acquiring strong reasoning capabilities [FPG24]. One contributing factor to this deficiency is the probabilistic nature of inferences generated by neural networks, which introduces the potential for errors. Since reasoning typically involves multiple inferential steps, the cumulative effect of these errors can ultimately result in extremely incorrect predictions [LeC23].

In contrast, symbolic learning excels at reasoning by learning logical rules. When these rules are valid, each inference step is guaranteed to be correct, thereby ensuring the reliability of complex reasoning processes involving multiple steps.

I aim to leverage cutting-edge neuro-symbolic learning approaches to enhance tactic-based ITP guidance. Although I integrate statistical and symbolic learning methods in Chapter 5, this integration remains rudimentary. A more promising approach would be to explore the application of advanced neuro-symbolic methods, such as α-ILP [SPDK23] and abductive learning [DXYZ19].

Nevertheless, I am concerned that current neuro-symbolic learning techniques may fall short in addressing formal reasoning tasks. A potentially more fruitful strategy would involve developing stronger neuro-symbolic methods in simpler domains, such as image classification. Once we achieve significant advances in neuro-symbolic AI, these methods can be applied to formal reasoning. A promising direction is to investigate neuro-symbolic approaches for learning object concepts from images, a topic explored in prior research [MGK+19, ERSLT18].

6.2.3 Large Language Models for Proof Automation

I aim to investigate the application of LLMs to proof automation, given their status as state-of-the-art learning models. The GPT-2 model employed in Chapter 4, with only 117 million parameters, pales in comparison to more recent architectures like GPT-4, which comprises approximately 1.76 trillion parameters. Empirical evidence from diverse fields such as natural language processing and image classification indicates that larger models tend to achieve superior performance. It is therefore plausible that deploying larger models in the context of proof automation will yield similarly enhanced outcomes.

Additionally, I intend to leverage cutting-edge learning techniques developed for LLMs in the domain of proof automation. One prominent research area in the LLM community focuses on the creation of more intelligent AI agents [GCW+24]. Another compelling avenue is post-training, a technique that has contributed significantly to the success of models such as GPT-o1 [Ope24]. This model has substantially advanced the reasoning capabilities of LLMs, achieving results that closely approach human-level performance, particularly in tasks requiring advanced reasoning. The central idea behind post-training techniques is to perform more comprehensive reasoning before generating outputs and to iteratively expand high-quality training data, employing methods akin to reinforcement learning [ZWMG22, ZHS+24, JYX+23]. Given that both intelligent agent architectures and post-training strategies markedly improve LLMs' reasoning capabilities, it is reasonable to expect that the application of these novel techniques could also enhance the performance of proof automation systems.

Another promising direction involves generating additional formal proof data using LLMs. A key advantage of formal mathematics is its verifiability: once a proof is verified by ITP, its correctness is guaranteed, unlike natural language question-answering, which suffers from the inherent ambiguity of human language. However, the corpus of high-quality formal mathematical data remains significantly smaller than that of natural language, rendering it insufficient for training contemporary LLMs. A feasible strategy involves proposing formal definitions and theorems, then employing LLMs to search for proofs of these theorems, with the correctness of the generated proofs subsequently verified by ITPs. This newly generated formal data can then be incorporated into the training set, thereby expanding the dataset for future iterations. Several studies have already demonstrated early success with this approach [WJL+22, X+23, TWL+24, XGS+24].

6.2.4 Learning for Rewriting

A considerable body of machine learning research has been applied to ATP and ITP, while the exploration of its use in term rewriting remains comparatively limited. In future work, I aim to extend machine learning techniques to various term rewriting challenges. These include: (1) predicting strategies for a given TRS, (2) forecasting interpretations for termination algorithms, (3) designing strategies for automatic term rewriting tools beyond CSI, and (4) automatically generating invariants to verify program equivalences.

Another area of interest is the development of more diverse and systematic benchmarks for term rewriting. Although the ATP and term rewriting communities are closely

related, ATP research attracts greater attention. One notable advantage of the ATP field is that research progress is facilitated by a robust set of benchmarks, enabling systematic evaluation. These benchmarks span diverse problem domains and allow for both practical assessments, such as problem-solving speed, and theoretical investigations. In the future, I intend to construct similarly systematic evaluation benchmarks tailored to term rewriting.

Bibliography

[AAA+23] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[AEEM14] María Alpuente, Santiago Escobar, Javier Espert, and José Meseguer. A modular order-sorted equational generalization algorithm. *Information and Computation*, 235:98–136, 2014.

[AHK+14a] Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reason.*, 52(2):191–213, 2014.

[AHK+14b] Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. Premise selection for mathematics by corpus analysis and kernel methods. *Journal of automated reasoning*, 52:191–213, 2014.

[Any23] Anysphere. Cursor, 2023.

[AYT09] Takahito Aoto, Junichi Yoshida, and Yoshihito Toyama. Proving confluence of term rewriting systems automatically. In *Rewriting Techniques and Applications: 20th International Conference, RTA 2009 Brasília, Brazil, June 29-July 1, 2009 Proceedings 20*, pages 93–102. Springer, 2009.

[BBC+23] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[BBG+15] Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, Karol Pak, and Josef Urban. Mizar: State-of-the-art and beyond. In *Intelligent Computer Mathematics: International Conference, CICM 2015*, pages 261–279. Springer, 2015.

[BBG+18] Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, and Karol Pąk. The role of the mizar mathematical library for interactive proof development in mizar. *Journal of Automated Reasoning*, 61:9–32, 2018.

[BBP13] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C Paulson. Extending sledgehammer with smt solvers. *Journal of automated reasoning*, 51(1):109–128, 2013.

[BCG05] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. LSH Forest: Self-tuning indexes for similarity search. In Allan Ellis and Tatsuya Hagino, editors, *Proceedings of the 14th International Conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, pages 651–660. ACM, 2005.

[BCGD+06] Frédéric Blanqui, Solange Coupet-Grimal, William Delobel, Sébastien Hinderer, and Adam Koprowski. Color: a coq library on rewriting and termination. In *Eighth International Workshop on Termination-WST 2006*, 2006.

[BCMM21] Candice Bentéjac, Anna Csörgő, and Gonzalo Martínez-Muñoz. A comparative analysis of gradient boosting algorithms. *Artificial Intelligence Review*, 54:1937–1967, 2021.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[Ber08] Yves Bertot. A short presentation of coq. In *Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings 21*, pages 12–16. Springer, 2008.

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[BK22] Chad E Brown and Cezary Kaliszyk. Lash 1.0 (system description). In *International Joint Conference on Automated Reasoning*, pages 350–358. Springer, 2022.

[BKPU16a] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C Paulson, and Josef Urban. Hammering towards qed. *Journal of Formalized Reasoning*, 9(1):101–148, 2016.

[BKPU16b] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formalized Reasoning*, 9(1):101–148, 2016.

[Bla23] Lasse Blaauwbroek. Tactician's web of large-scale formal knowledge, December 2023.

[BLR+19a] Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. HOList: An environment for machine learning of higher order logic theorem proving. In *International Conference on Machine Learning*, pages 454–463. PMLR, 2019.

[BLR+19b] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher order logic theorem proving. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 454–463. PMLR, 2019.

[BM14] Robert S Boyer and J Strother Moore. *A computational logic handbook: Formerly notes and reports in computer science and applied mathematics*. Elsevier, 2014.

[BMR+20] 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.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1998.

[BOR+24] Lasse Blaauwbroek, Mirek Olšák, Jason Rute, Fidel Ivan Schaposnik Massolo, Jelle Piepenbrock, and Vasily Pestun. Graph2tac: Online representation learning of formal math concepts. In *Forty-first International Conference on Machine Learning*, 2024.

[Bre01a] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.

[Bre01b] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[Bro97] Andrei Z. Broder. On the resemblance and containment of documents. In Bruno Carpentieri, Alfredo De Santis, Ugo Vaccaro, and James A. Storer, editors, *Compression and Complexity of SEQUENCES 1997, Positano, Amalfitan Coast, Salerno, Italy, June 11-13, 1997, Proceedings*, pages 21–29. IEEE, 1997.

[Bro12] Chad E Brown. Satallax: An automatic higher-order prover. In *Automated Reasoning: 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings 6*, pages 111–117. Springer, 2012.

[BUG20a] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. Tactic learning and proving for the Coq proof assistant. In Elvira Albert and Laura Kovács, editors, *Proceedings of the 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2020*, volume 73 of *EPiC Series in Computing*, pages 138–150. EasyChair, 2020.

[BUG20b] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. The Tactician – A seamless, interactive tactic learner and prover for Coq. In Christoph

Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics - 13th International Conference, CICM 2020*, volume 12236 of *LNCS*, pages 271–277. Springer, 2020.

[BUG20c] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. The Tactician – A seamless, interactive tactic learner and prover for Coq. In Christoph Benzmüller and Bruce R. Miller, editors, *Proceedings of the 13th International Conference on Intelligent Computer Mathematics CICM 2020, Bertinoro, Italy, July 26-31, 2020*, volume 12236 of *LNCS*, pages 271–277. Springer, 2020.

[CCF+07] Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Certification of automated termination proofs. In *Frontiers of Combining Systems: 6th International Symposium, FroCoS 2007 Liverpool, UK, September 10-12, 2007 Proceedings 6*, pages 148–162. Springer, 2007.

[CD22] Andrew Cropper and Sebastijan Dumančić. Inductive logic programming at 30: a new introduction. *Journal of Artificial Intelligence Research*, 74:765–850, 2022.

[CFU08] Pierre Courtieu, Julien Forest, and Xavier Urbain. Certifying a termination criterion based on graphs, without graphs. In *International Conference on Theorem Proving in Higher Order Logics*, pages 183–198. Springer, 2008.

[CG16] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016.

[Chl13] Adam Chlipala. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2013.

[CJSU19a] Karel Chvalovský, Jan Jakubuv, Martin Suda, and Josef Urban. ENIGMA-NG: Efficient neural and gradient-boosted inference guidance for E. In Pascal Fontaine, editor, *Proceedings of the 27th International Conference on Automated Deduction, CADE 27, Natal, Brazil, August 27-30, 2019*, volume 11716 of *LNCS*, pages 197–215. Springer, 2019.

[CJSU19b] Karel Chvalovský, Jan Jakubův, Martin Suda, and Josef Urban. Enigma-ng: efficient neural and gradient-boosted inference guidance for e. In *CADE 27: August 27–30, 2019, Proceedings 27*, pages 197–215. Springer, 2019.

[CK18a] Łukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *Journal of automated reasoning*, 61:423–453, 2018.

[CK18b] Lukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for dependent type theory. *J. Autom. Reasoning*, 61(1-4):423–453, 2018.

[CK23] David M Cerna and Temur Kutsia. Anti-unification and generalization: a survey. *arXiv preprint arXiv:2302.00277*, 2023.

[CM21] Andrew Cropper and Rolf Morel. Learning programs by learning from failures. *Machine Learning*, 110:801–856, 2021.

[cop] The cops dataset.

[CSG+24] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524*, 2024.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[Del00] David Delahaye. A tactic language for the system coq. In *Logic for Programming and Automated Reasoning: 7th International Conference, LPAR 2000 Reunion Island, France, November 6–10, 2000 Proceedings 7*, pages 85–95. Springer, 2000.

[DH00] Pedro M. Domingos and Geoff Hulten. Mining high-speed data streams. In Raghu Ramakrishnan, Salvatore J. Stolfo, Roberto J. Bayardo, and Ismail Parsa, editors, *Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 71–80. ACM, 2000.

[dMKA+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *CADE-25: 25th International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[DR08] Luc De Raedt. *Logical and relational learning*. Springer Science & Business Media, 2008.

[DRKT07] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI*, volume 7, pages 2462–2467. Hyderabad, 2007.

[DSST89] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.

[Dud76] Sahibsingh A Dudani. The distance-weighted k-nearest-neighbor rule. *IEEE Transactions on Systems, Man, and Cybernetics*, (4):325–327, 1976.

[DXYZ19] Wang-Zhou Dai, Qiuling Xu, Yang Yu, and Zhi-Hua Zhou. Bridging machine learning and logical reasoning by abductive learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[EG18] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61:1–64, 2018.

[ERSLT18] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. *Advances in neural information processing systems*, 31, 2018.

[FB16] Michael Färber and Chad E. Brown. Internal guidance for Satallax. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning - 8th International Joint Conference, IJCAR 2016*, volume 9706 of *LNCS*, pages 349–361. Springer, 2016.

[Fel15] Bertram Felgenhauer. *Confluence of Term Rewriting: Theory and Automation*. PhD thesis, PhD thesis, University of Innsbruck, 2015.

[FGMP97] Moreno Falaschi, Maurizio Gabbrielli, Kim Marriott, and Catuscia Palamidessi. Confluence in concurrent constraint programming. *Theoretical Computer Science*, 183(2):281–315, 1997.

[Fit12] Melvin Fitting. *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.

[FK15a] Michael Färber and Cezary Kaliszyk. Random forests for premise selection. In *International Symposium on Frontiers of Combining Systems*, pages 325–340. Springer, 2015.

[FK15b] Michael Färber and Cezary Kaliszyk. Random forests for premise selection. In Carsten Lutz and Silvio Ranise, editors, *Proceedings of Frontiers of Combining Systems (FroCoS'15)*, volume 9322 of *LNCS*, pages 325–340. Springer, 2015.

[FPG24] Evelina Fedorenko, Steven T Piantadosi, and Edward AF Gibson. Language is primarily a tool for communication rather than thought. *Nature*, 630(8017):575–586, 2024.

[G+08] Georges Gonthier et al. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[GCJ+21] Zarathustra A Goertzel, Karel Chvalovský, Jan Jakubův, Miroslav Olšák, and Josef Urban. Fast and slow enigmas and parental guidance. In *International Symposium on Frontiers of Combining Systems*, pages 173–191. Springer, 2021.

[GCW+24] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*, 2024.

[GIM99] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *Proceedings of 25th International Conference on Very Large Data Bases, VLDB'99, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 518–529. Morgan Kaufmann, 1999.

[GJK+22] Zarathustra A Goertzel, Jan Jakubův, Cezary Kaliszyk, Miroslav Olšák, Jelle Piepenbrock, and Josef Urban. The isabelle enigma. *arXiv preprint arXiv:2205.01981*, 2022.

[GK15a] Thibault Gauthier and Cezary Kaliszyk. Premise selection and external provers for HOL4. In Xavier Leroy and Alwen Tiu, editors, *Proceedings of the 4th Conference on Certified Programs and Proofs (CPP'15)*, pages 49–57. ACM, 2015.

[GK15b] Thibault Gauthier and Cezary Kaliszyk. Premise selection and external provers for hol4. In *CPP*, pages 49–57, 2015.

[GK19] Thibault Gauthier and Cezary Kaliszyk. Aligning concepts across proof assistant libraries. *J. Symbolic Computation*, 90:89–123, 2019.

[GKU17] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. TacticToe: Learning to reason with HOL4 tactics. In Thomas Eiter and David Sands, editors, *Proceedings of the 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR-21*, volume 46 of *EPiC Series in Computing*, pages 125–143. EasyChair, 2017.

[GKU+21a] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. TacticToe: Learning to prove with tactics. *Journal of Automated Reasoning*, 65(2):257–286, 2021.

[GKU+21b] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. TacticToe: Learning to prove with tactics. *J. Autom. Reason.*, 65(2):257–286, 2021.

[GMR+15] Jürgen Giesl, Frédéric Mesnard, Albert Rubio, René Thiemann, and Johannes Waldmann. Termination competition (termcomp 2015). In *International Conference on Automated Deduction*, pages 105–108. Springer, 2015.

[Göd92] Kurt Gödel. *On formally undecidable propositions of Principia Mathematica and related systems*. Courier Corporation, 1992.

[GSKT06] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In *International Joint Conference on Automated Reasoning*, pages 281–286. Springer, 2006.

[GWR15] Thomas Gransden, Neil Walkinshaw, and Rajeev Raman. SEPIA: Search for proofs using inferred automata. In *International Conference on Automated Deduction*, pages 246–255. Springer, 2015.

[H+21] Jesse Michael Han et al. Proof artifact co-training for theorem proving with language models. *arXiv preprint arXiv:2102.06203*, 2021.

[HAB+17] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Hoang Le Truong, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. A formal proof of the Kepler conjecture. In *Forum of mathematics, Pi*, volume 5. Cambridge University Press, 2017.

[Har96] John Harrison. Hol light: A tutorial introduction. In *Formal Methods in Computer-Aided Design: First International Conference, FMCAD'96 Palo Alto, CA, USA, November 6–8, 1996 Proceedings 1*, pages 265–269. Springer, 1996.

[Har09] John Harrison. Hol light: An overview. In *International Conference on Theorem Proving in Higher Order Logics*, pages 60–66. Springer, 2009.

[HB13] Florian Haftmann and Lukas Bulwahn. Code generation from isabelle/hol theories. *Part of the Isabelle documentation: http://isabelle. in. tum. de/dist/Isabelle2017/doc/codegen. pdf*, 2013.

[HIM12] Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory Comput.*, 8(1):321–350, 2012.

[HK20] John T Hancock and Taghi M Khoshgoftaar. Catboost for big data: an interdisciplinary review. *Journal of big data*, 7(1):94, 2020.

[HKT02] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In *International Conference on Graph Transformation*, pages 161–176. Springer, 2002.

[HM76] James Wayne Hunt and M Douglas MacIlroy. *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.

[HSLZ21] Steven CH Hoi, Doyen Sahoo, Jing Lu, and Peilin Zhao. Online learning: A comprehensive survey. *Neurocomputing*, 459:249–289, 2021.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[HUW14] John Harrison, Josef Urban, and Freek Wiedijk. History of interactive theorem proving. In *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 135–214. Elsevier, 2014.

[HW79] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[ISA+16] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. Deepmath-deep sequence models for premise selection. *NIPS*, 29, 2016.

[J+20] Jan Jakubův et al. Enigma anonymous: Symbol-independent inference guiding machine (system description). In *IJCAR*, pages 448–463. Springer, 2020.

[Jac01] Paul Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat*, 37:547–579, 1901.

[JKJ+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[JLT+22] Albert Q 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. *arXiv preprint arXiv:2205.10893*, 2022.

[JM24] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd edition, 2024. Online manuscript released August 20, 2024.

[Jon04] Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 60(5):493–502, 2004.

[JU17] Jan Jakubuv and Josef Urban. ENIGMA: efficient learning-based inference guiding machine. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, editors, *Intelligent Computer Mathematics - 10th International Conference, CICM 2017*, volume 10383 of *LNCS*, pages 292–302. Springer, 2017.

[Jur00] Daniel Jurafsky. Speech and language processing, 2000.

[JYX+23] Ziwei Ji, Tiezheng Yu, Yan Xu, Nayeon Lee, Etsuko Ishii, and Pascale Fung. Towards mitigating llm hallucination via self reflection. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1827–1843, 2023.

[KCS17] Cezary Kaliszyk, François Chollet, and Christian Szegedy. HolStep: A machine learning dataset for higher-order logic theorem proving. In *ICLR 2017*. OpenReview.net, 2017.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[KMF+17] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30, 2017.

[KMH+20] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[KSZM09] Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean termination tool 2. In *Rewriting Techniques and Applications: 20th International Conference, RTA 2009 Brasília, Brazil, June 29-July 1, 2009 Proceedings 20*, pages 295–304. Springer, 2009.

[KU13] Cezary Kaliszyk and Josef Urban. Stronger automation for flyspeck by feature weighting and strategy evolution. In *Third International Workshop on Proof Exchange for Theorem Proving (PxTP 2013)*, page 87, 2013.

[KUMO18a] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement learning of theorem proving. *Advances in Neural Information Processing Systems*, 31, 2018.

[KUMO18b] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement learning of theorem proving. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 8836–8847. Curran Associates, Inc., 2018.

[KUV15a] Cezary Kaliszyk, Josef Urban, and Jirí Vyskocil. Efficient semantic features for automated reasoning over large theories. In Qiang Yang and Michael J. Wooldridge, editors, *IJCAI 2015*, pages 3084–3090. AAAI Press, 2015.

[KUV15b] Cezary Kaliszyk, Josef Urban, and Jiří Vyskočil. Efficient semantic features for automated reasoning over large theories. In Qiang Yang and Michael Wooldridge, editors, *Proceedings of the 24th International Joint Conference on Artificial Intelligence, (IJCAI'15)*, pages 3084–3090. AAAI Press, 2015.

[KUV15c] Cezary Kaliszyk, Josef Urban, and Jiří Vyskočil. Efficient semantic features for automated reasoning over large theories. In Qiang Yang and Michael Wooldridge, editors, *IJCAI*, pages 3084–3090. AAAI Press, 2015.

[KUV17a] Cezary Kaliszyk, Josef Urban, and Jiří Vyskočil. Automating formalization by statistical and semantic parsing of mathematics. In *Interactive Theorem Proving: 8th International Conference, ITP 2017, Brasília, Brazil, September 26–29, 2017, Proceedings 8*, pages 12–27. Springer, 2017.

[KUV17b] Cezary Kaliszyk, Josef Urban, and Jirí Vyskocil. Automating formalization by statistical and semantic parsing of mathematics. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017*, volume 10499 of *LNCS*, pages 12–27. Springer, 2017.

[LeC23] Yann LeCun. Do large language models need sensory grounding for meaning and understanding. In *Workshop on Philosophy of Deep Learning*, 2023.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[Ler21] Xavier Leroy. *The CompCert C verified compiler: Documentation and user's manual*. PhD thesis, Inria, 2021.

[LK18] Josh Levy-Kramer. k-means-constrained, April 2018.

[MAT+23] Maciej Mikuła, Szymon Antoniak, Szymon Tworkowski, Albert Qiaochu Jiang, Jin Peng Zhou, Christian Szegedy, Łukasz Kuciński, Piotr Miłoś, and Yuhuai Wu. Magnushammer: A transformer-based approach to premise selection. *arXiv preprint arXiv:2303.04488*, 2023.

[MGK+19] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. *arXiv preprint arXiv:1904.12584*, 2019.

[MIB+24] Kyle Mahowald, Anna A Ivanova, Idan A Blank, Nancy Kanwisher, Joshua B Tenenbaum, and Evelina Fedorenko. Dissociating language and thought in large language models. *Trends in Cognitive Sciences*, 2024.

[Mit97] Tom M. Mitchell. *Machine learning, International Edition*. McGraw-Hill Series in Computer Science. McGraw-Hill, 1997.

[MLM23] Aart Middeldorp, Alexander Lochmann, and Fabian Mitterwallner. First-order theory of rewriting for linear variable-separated rewrite systems: Automation, formalization, certification. *Journal of Automated Reasoning*, 67(2):14, 2023.

[MM97] Tom M Mitchell and Tom M Mitchell. *Machine learning*, volume 1. McGraw-hill New York, 1997.

[MO99] Heiko Mantel and Jens Otten. lintap: A tableau prover for linear logic. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 217–231. Springer, 1999.

[Moc12a] Shinichi Mochizuki. Inter-universal teichmuller theory i: Construction of hodge theaters. Under review, 2012.

[Moc12b] Shinichi Mochizuki. Inter-universal teichmuller theory ii: Hodge-arakelov-theoretic evaluation. Under review, 2012.

[Moc12c] Shinichi Mochizuki. Inter-universal teichmuller theory iii: Canonical splittings of the log-theta-lattice. Under review, 2012.

[Moc12d] Shinichi Mochizuki. Inter-universal teichmuller theory iv: Log-volume computations and set-theoretic foundations. Under review, 2012.

[MS19] Victor Cacciari Miraldo and Wouter Swierstra. An efficient algorithm for type-safe structural diffing. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.

[Mug95] Stephen Muggleton. Inverse entailment and progol. *New generation computing*, 13:245–286, 1995.

[Nag19] Yutaka Nagashima. Lifter: language to encode induction heuristics for isabelle/hol. In *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, December 1–4, 2019, Proceedings 17*, pages 266–287. Springer, 2019.

[NH18] Yutaka Nagashima and Yilun He. Pamper: proof method recommendation system for isabelle/hol. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 362–372. ACM, 2018.

[NHX+23] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages. *ICLR*, 2023.

[NK17] Yutaka Nagashima and Ramana Kumar. A proof strategy language and proof script generation for Isabelle/HOL. In Leonardo de Moura, editor, *Proceedings of the 26th International Conference on Automated Deduction, CADE 26*, volume 10395 of *LNCS*, pages 528–545. Springer, 2017.

[NLA19] Max S New, Daniel R Licata, and Amal Ahmed. Gradual type theory. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–31, 2019.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[OKU20] Miroslav Olšák, Cezary Kaliszyk, and Josef Urban. Property invariant embedding for automated reasoning. In *ECAI 2020*, pages 1395–1402. IOS Press, 2020.

[Ope24] OpenAI. Openai o1 system card, September 2024.

[P+11] F. Pedregosa et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[Pau94] Lawrence C Paulson. *Isabelle: A generic theorem prover*. Springer, 1994.

[Ped19] Pierre-Marie Pedrot. Ltac2: tactical warfare. In *The Fifth International Workshop on Coq for Programming Languages, CoqPL*, volume 2019, 2019.

[Pet09] Leif E Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009.

[Pfe91] Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *LICS*, volume 91, pages 74–85, 1991.

[Pie02] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[Plo71] Gordon D Plotkin. A further note on inductive generalization. *Machine intelligence*, 6:101–124, 1971.

[PM15] Christine Paulin-Mohring. Introduction to the calculus of inductive constructions, 2015.

[PMA23] Bartosz Piotrowski, Ramon Fernández Mir, and Edward Ayers. Machine-learned premise selection for lean. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 175–186. Springer, 2023.

[Pro13] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. *arXiv preprint arXiv:1308.0729*, 2013.

[PU18] Bartosz Piotrowski and Josef Urban. ATPboost: Learning premise selection in binary setting with ATP feedback. In *IJCAR*, volume 10900 of *LNCS*, pages 566–574, 2018.

[PU20] Bartosz Piotrowski and Josef Urban. Guiding inferences in connection tableau by recurrent neural networks. In *International Conference on Intelligent Computer Mathematics*, pages 309–314. Springer, 2020.

[Qui90] J. Ross Quinlan. Learning logical definitions from relations. *Machine learning*, 5:239–266, 1990.

[R+24] Jason Rute et al. Graph2tac: Learning hierarchical representations of math concepts in theorem proving. *arXiv preprint arXiv:2401.02949*, 2024.

[Ray09] Oliver Ray. Nonmonotonic abductive inductive learning. *Journal of Applied Logic*, 7(3):329–340, 2009.

[Rey70] John C Reynolds. Transformational systems and algebraic structure of atomic formulas. *Machine intelligence*, 5:135–151, 1970.

[RV99] Alexandre Riazanov and Andrei Voronkov. Vampire. In Harald Ganzinger, editor, *CADE-16 – 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, volume 1632 of *LNCS*, pages 292–296. Springer, 1999.

[RWC+19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[S+07] Yutaka Sasaki et al. The truth of the f-measure. *Teach tutor mater*, 1(5):1–5, 2007.

[SAH+20] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[Sch13] Stephan Schulz. System description: E 1.8. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *LPAR-19 – 19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

[SLS+09] Amir Saffari, Christian Leistner, Jakob Santner, Martin Godec, and Horst Bischof. On-line random forests. In *12th IEEE International Conference on Computer Vision Workshops, ICCV Workshops 2009, Kyoto, Japan, September 27 - October 4, 2009*, pages 1393–1400. IEEE Computer Society, 2009.

[Sma21] Nicholas Smallbone. Twee: An equational theorem prover. In *CADE*, pages 602–613, 2021.

[SN08a] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2008, Montreal, Canada, August 18-21, 2008*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008.

[SN08b] Konrad Slind and Michael Norrish. A brief overview of hol4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.

[SPDK23] Hikaru Shindo, Viktor Pfanschilling, Devendra Singh Dhami, and Kristian Kersting. α ilp: thinking visual scenes as differentiable logic programs. *Machine Learning*, 112(5):1465–1497, 2023.

[Sri01] Ashwin Srinivasan. The aleph manual, 2001.

[ST14] Christian Sternagel and René Thiemann. The certification problem format. *arXiv preprint arXiv:1410.8220*, 2014.

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006.

[Sud21] Martin Suda. Vampire with a brain is a good ITP hammer. In Boris Konev and Giles Reger, editors, *Frontiers of Combining Systems - 13th International Symposium, FroCoS 2021*, volume 12941 of *LNCS*, pages 192–209. Springer, 2021.

[Swi08] Wouter Swierstra. Data types à la carte. *Journal of functional programming*, 18(4):423–436, 2008.

[TAB+23] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[Tae23] Victor Taelin. Agda to typescript compilation with sonnet-3.5, 2023. Accessed: September 29, 2024.

[The19] The Coq Development Team. The Coq proof assistant, version 8.11.0, Oct 2019.

[The20] The Coq Development Team. Coq reference manual 8.11.1, 2020.

[TS09] René Thiemann and Christian Sternagel. Certification of termination proofs using ceta. In *International Conference on Theorem Proving in Higher Order Logics*, pages 452–468. Springer, 2009.

[TWL+24] 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.

[UB07] Josef Urban and Grzegorz Bancerek. Presenting and explaining mizar. *Electronic Notes in Theoretical Computer Science*, 174(2):63–74, 2007.

[UJ20] Josef Urban and Jan Jakubův. First neural conjecturing datasets and experiments. In *International Conference on Intelligent Computer Mathematics*, pages 315–323. Springer, 2020.

[VdBHKO02] Mark GJ Van den Brand, Jan Heering, Paul Klint, and Pieter A Olivier. Compiling language definitions: the asf+ sdf compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):334–368, 2002.

[VSP+17] 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.

[WBKU20a] Qingxiang Wang, Chad Brown, Cezary Kaliszyk, and Josef Urban. Exploration of neural machine translation in autoformalization of mathematics in mizar. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 85–98, 2020.

[WBKU20b] Qingxiang Wang, Chad E. Brown, Cezary Kaliszyk, and Josef Urban. Exploration of neural machine translation in autoformalization of mathematics in mizar. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020*, pages 85–98. ACM, 2020.

[WDS+19] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[Wen99] Markus Wenzel. Isar—a generic interpretative approach to readable formal proof documents. In *International Conference on Theorem Proving in Higher Order Logics*, pages 167–183. Springer, 1999.

[WJL+22] Yuhuai Wu, Albert Q Jiang, Wenda Li, Markus N Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *arXiv preprint arXiv:2205.12615*, 2022.

[WKU18] 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.

[WRL+21] Yuhuai Wu, Markus N Rabe, Wenda Li, Jimmy Ba, Roger B Grosse, and Christian Szegedy. Lime: Learning inductive bias for primitives of mathematical reasoning. In *International Conference on Machine Learning*, pages 11251–11262. PMLR, 2021.

[WTWD17] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. *Advances in neural information processing systems*, 30, 2017.

[X+23] Huajian Xin et al. Lego-prover: Neural theorem proving with growing libraries. *arXiv preprint arXiv:2310.00656*, 2023.

[XGS+24] Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*, 2024.

[Y+23] Kaiyu Yang et al. Leandojo: Theorem proving with retrieval-augmented language models. *arXiv preprint arXiv:2306.15626*, 2023.

[YD19] 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.

[YD21] Kaiyu Yang and Jia Deng. Learning symbolic rules for reasoning in quasi-natural language. *arXiv preprint arXiv:2111.12038*, 2021.

[ZBKU23] Liao Zhang, Lasse Blaauwbroek, Cezary Kaliszyk, and Josef Urban. Learning proof transformations and its applications in interactive theorem proving. In *International Symposium on Frontiers of Combining Systems*, pages 236–254. Springer Nature Switzerland Cham, 2023.

[ZBP+21] Liao Zhang, Lasse Blaauwbroek, Bartosz Piotrowski, Prokop Černý, Cezary Kaliszyk, and Josef Urban. Online machine learning techniques for Coq: A comparison. In *International Conference on Intelligent Computer Mathematics*, pages 67–83. Springer, 2021.

[ZCK24] Liao Zhang, David M Cerna, and Cezary Kaliszyk. Learning rules explaining interactive theorem proving tactic prediction. *arXiv preprint arXiv:2411.01188*, 2024.

[ZFM11] Harald Zankl, Bertram Felgenhauer, and Aart Middeldorp. Csi–a confluence tool. In *Automated Deduction–CADE-23: 23rd International Conference on Automated Deduction, Wrocław, Poland, July 31-August 5, 2011. Proceedings 23*, pages 499–505. Springer, 2011.

[ZHS+24] Eric Zelikman, Georges Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah D Goodman. Quiet-star: Language models can teach themselves to think before speaking. *arXiv preprint arXiv:2403.09629*, 2024.

[ZKZ+15] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.

[ZWMG22] Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

[ZZS+19] Chongsheng Zhang, Yuan Zhang, Xianjin Shi, George Almpanidis, Gaojuan Fan, and Xiajiong Shen. On incremental learning for gradient boosting decision trees. *Neural Process. Lett.*, 50(1):957–987, 2019.