Learning Proof Transformations and Its Applications in Interactive Theorem Proving

Liao Zhang[1,2][0000−0002−4574−8843], Lasse Blaauwbroek[3][0000−0003−2910−8069], Cezary Kaliszyk[1,4][0000−0002−8273−6059], and Josef Urban[2][0000−0002−1384−1613]

[1] University of Innsbruck, Innsbruck, Austria
[2] Czech Technical University in Prague, Czech Republic
[3] Institut des Hautes Etudes Scientifiques Paris, France
[4] International Neurodegenerative Disorders Research Center, Czech Republic

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.

Keywords: Interactive theorem proving · Machine learning · Neural networks.

1 Introduction

Interactive theorem provers (ITPs) [15] are sophisticated systems used for constructing machine-verified proofs. Various proof assistants, such as HOL4 [31], HOL Light [14], Lean [23], Isabelle/HOL [24], and Mizar [3], are used by formalizers. Coq [33] 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 1). To complete the proof, one can perform induction on the list `l` (with the help of the tactic `induction l as [| n l'`

```
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.
```

Fig. 1: A formal Coq proof, showing the equality property of the lengths of a list and its reverse

IHl']), splitting the proof state into a case where l is empty and a case where l is nonempty. In the first case, 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 2 presents the before and after states of the tactic `rewrite add_comm` at its position in Figure 1. In this particular case, the hypotheses remain unchanged, but in the goal, the two sides of the addition are swapped.

```
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

Fig. 2: The before and after states of `rewrite add_comm` in Figure 1, with hypotheses above the dashed line and the required goal below it.

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.

1.1 Motivation

Tactic prediction methods have so far relied solely on before states, typically to guide automated tactical proof search in systems like Tactician [6]. 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 [2], 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 [30] 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.

1.2 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 2, to capture the semantics of tactics, we design three characterizations: feature difference, anti-unification, and tree difference.

3. In Section 4, we conduct experiments to verify the strengths of our characterizations with a random forests classifier and the GPT-2 language model.
4. In Section 5, we propose and evaluate two applications of the task, namely tactic suggestion and proof shortening.

Besides the above-mentioned contributions, Section 3 introduces the preliminaries of the learning technology used in this paper. We discuss two related research fields in Section 6. The conclusions and future work are presented in Section 7.

2 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.

2.1 Feature Difference

To characterize the proof states, we start with the features used by [42]. 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 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.

2.2 Anti-unification

Anti-unification, first proposed by Plotkin [27] and Reynolds [29], aims to calculate generalizations of the given objects. Since Coq is based on the Calculus of Inductive Constructions (CIC) [25], an appropriate anti-unification algorithm for Coq should be higher-order. However, higher-order anti-unification is undecidable [26]. 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,

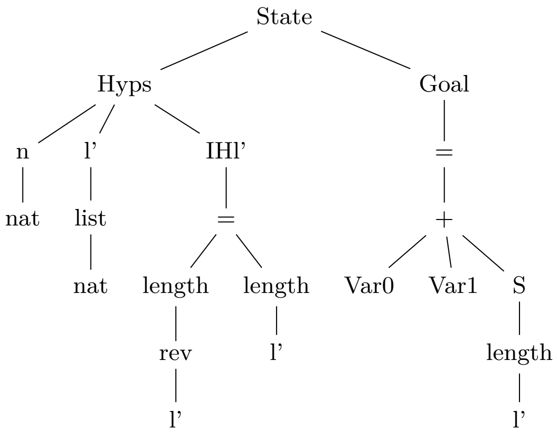


Fig. 3: The least general generalization of the before and after states in Figure 2

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 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 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 3 presents the lgg of the before and after states considered in Figure 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 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 [19].

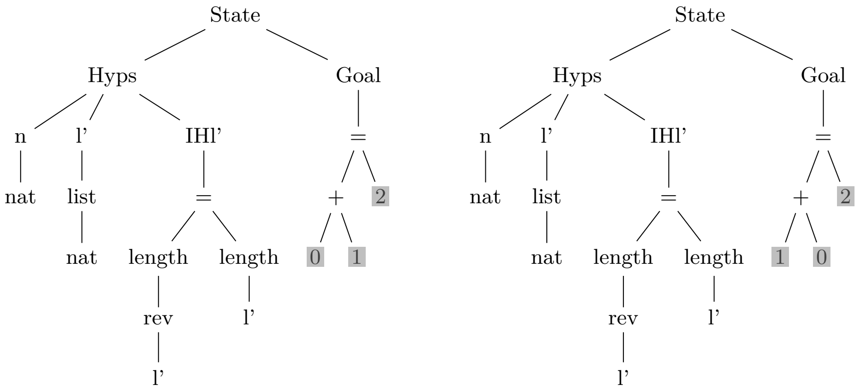
2.3 Tree Difference

In addition to anti-unification, we propose a characterization based on a tree difference algorithm [21]. 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 [16] 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 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 [21]. 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.

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



(a) Deletion context

(b) Insertion context

Fig. 4: The deletion and insertion contexts of the before and after states in Figure 2. `Hole0`, `Hole1`, and `Hole2` denote `length l'`, `1`, and `S(length l')`, respectively.

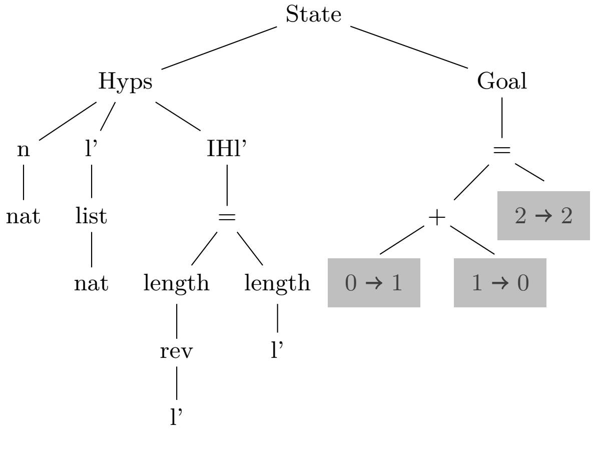


Fig. 5: The patch of the before and after states in Figure 2

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*.

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 2.2. 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 5 depicts the patch of the before and after states in Figure 2.

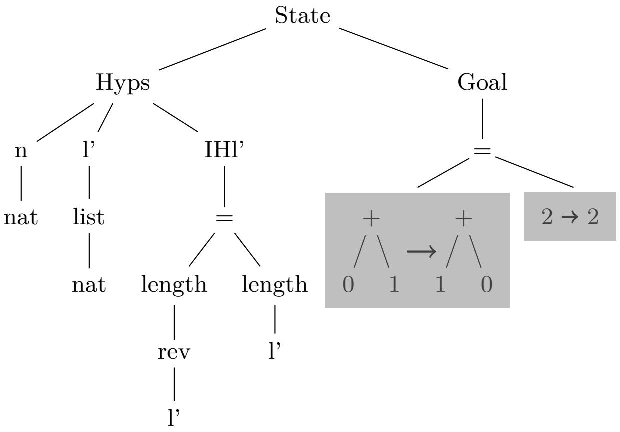


Fig. 6: The result of applying the closure function to the patch in Figure 5

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 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 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 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 2 for instance. The lgg in Figure 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 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.

2.4 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 2.1). 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.

3 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 [7]. 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 [42].

The other used machine learning technique is the pre-trained language model GPT-2 [28]. GPT-2 is based on neural networks, which consist of many artificial neurons to learn from training data. The self-attention [35] 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 [43]. 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) [8] and formal mathematics [34, 39].

4 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 [22]. 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 [38]. 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 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%

Table 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. 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 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 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%

5 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 (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.

Table 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

5.1 Tactic Suggestion

We view the experiments in Section 4 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 Forum[5]. 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 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_O` 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.

5.2 Shortening Proofs

The results presented in the previous Section 4 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 [5] 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 tactics would be unfair). For all tactic sequences of

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

Table 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

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. 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, 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 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

Table 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`

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.

6 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 [1]. 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 [19], as well as approaches that use deep learning for this [36]. 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 [32, 17] as well as in higher-order logic automated theorem proving [10]. Similarly, the meaning of the usefulness of a proof step has been considered, for example as part of the HOLStep [18], 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 [13]. TacticToe [12] and Tactician [5, 42] apply classical statistical learning techniques such k-nearest neighbors [9] and random forests [7] to generate tactic predictions based on the before states. Several systems use neural networks for the same task, e.g. HOList [4], CoqGym [41], and Lime [40]. These are all different from the current work that considers the after states as well.

Autoformalization [20] 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 [37], 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.

7 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 [11].

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.

References

1. Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., Urban, J.: Premise selection for mathematics by corpus analysis and kernel methods. J. Autom. Reason. **52**(2), 191–213 (2014). https://doi.org/10.1007/s10817-013-9286-5
2. Bancerek, G., Byliński, C., Grabowski, A., Korniłowicz, A., Matuszewski, R., Naumowicz, A., Pąk, K.: The role of the mizar mathematical library for interactive proof development in mizar. Journal of Automated Reasoning **61**, 9–32 (2018)
3. Bancerek, G., Byliński, C., Grabowski, A., Korniłowicz, A., Matuszewski, R., Naumowicz, A., Pak, K., Urban, J.: Mizar: State-of-the-art and beyond. In: Intelligent Computer Mathematics: International Conference, CICM 2015. pp. 261–279. Springer (2015)
4. Bansal, K., Loos, S., Rabe, M., Szegedy, C., Wilcox, S.: HOList: An environment for machine learning of higher order logic theorem proving. In: International Conference on Machine Learning. pp. 454–463. PMLR (2019)
5. Blaauwbroek, L., Urban, J., Geuvers, H.: Tactic learning and proving for the Coq proof assistant. In: Albert, E., Kovács, L. (eds.) LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning. EPiC, vol. 73, pp. 138–150. EasyChair (2020). https://doi.org/10.29007/wg1q
6. Blaauwbroek, L., Urban, J., Geuvers, H.: The Tactician – A seamless, interactive tactic learner and prover for Coq. In: Benzmüller, C., Miller, B.R. (eds.) Intelligent Computer Mathematics - 13th International Conference, CICM 2020. LNCS, vol. 12236, pp. 271–277. Springer (2020). https://doi.org/10.1007/978-3-030-53518-6_17
7. Breiman, L.: Random forests. Machine learning **45**(1), 5–32 (2001)
8. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. Advances in neural information processing systems **33**, 1877–1901 (2020)
9. Dudani, S.A.: The distance-weighted k-nearest-neighbor rule. IEEE Transactions on Systems, Man, and Cybernetics (4), 325–327 (1976)
10. Färber, M., Brown, C.E.: Internal guidance for Satallax. In: Olivetti, N., Tiwari, A. (eds.) Automated Reasoning - 8th International Joint Conference, IJCAR 2016. LNCS, vol. 9706, pp. 349–361. Springer (2016). https://doi.org/10.1007/978-3-319-40229-1_24
11. Gauthier, T., Kaliszyk, C.: Aligning concepts across proof assistant libraries. J. Symbolic Computation **90**, 89–123 (2019). https://doi.org/10.1016/j.jsc.2018.04.005
12. Gauthier, T., Kaliszyk, C., Urban, J., Kumar, R., Norrish, M.: TacticToe: Learning to prove with tactics. Journal of Automated Reasoning **65**(2), 257–286 (2021)
13. Gransden, T., Walkinshaw, N., Raman, R.: SEPIA: Search for proofs using inferred automata. In: International Conference on Automated Deduction. pp. 246–255. Springer (2015)
14. Harrison, J.: 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. pp. 265–269. Springer (1996)
15. Harrison, J., Urban, J., Wiedijk, F.: History of interactive theorem proving. In: Computational Logic, Handbook of the History of Logic, vol. 9, pp. 135–214. Elsevier (2014)
16. Hunt, J.W., MacIlroy, M.D.: An algorithm for differential file comparison. Bell Laboratories Murray Hill (1976)

17. Jakubuv, J., Urban, J.: ENIGMA: efficient learning-based inference guiding machine. In: Geuvers, H., England, M., Hasan, O., Rabe, F., Teschke, O. (eds.) Intelligent Computer Mathematics - 10th International Conference, CICM 2017. LNCS, vol. 10383, pp. 292–302. Springer (2017). https://doi.org/10.1007/978-3-319-62075-6_20
18. Kaliszyk, C., Chollet, F., Szegedy, C.: HolStep: A machine learning dataset for higher-order logic theorem proving. In: ICLR 2017. OpenReview.net (2017)
19. Kaliszyk, C., Urban, J., Vyskocil, J.: Efficient semantic features for automated reasoning over large theories. In: Yang, Q., Wooldridge, M.J. (eds.) IJCAI 2015. pp. 3084–3090. AAAI Press (2015)
20. Kaliszyk, C., Urban, J., Vyskocil, J.: Automating formalization by statistical and semantic parsing of mathematics. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) Interactive Theorem Proving - 8th International Conference, ITP 2017. LNCS, vol. 10499, pp. 12–27. Springer (2017). https://doi.org/10.1007/978-3-319-66107-0_2
21. Miraldo, V.C., Swierstra, W.: An efficient algorithm for type-safe structural diffing. Proceedings of the ACM on Programming Languages **3**(ICFP), 1–29 (2019)
22. Mitchell, T.M., Mitchell, T.M.: Machine learning, vol. 1. McGraw-hill New York (1997)
23. de Moura, L., Kong, S., Avigad, J., Van Doorn, F., von Raumer, J.: The Lean theorem prover (system description). In: CADE-25: 25th International Conference on Automated Deduction. pp. 378–388. Springer (2015)
24. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: a proof assistant for higher-order logic. Springer (2002)
25. Paulin-Mohring, C.: Introduction to the calculus of inductive constructions (2015)
26. Pfenning, F.: Unification and anti-unification in the calculus of constructions. In: LICS. vol. 91, pp. 74–85 (1991)
27. Plotkin, G.D.: A further note on inductive generalization. Machine intelligence **6**, 101–124 (1971)
28. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al.: Language models are unsupervised multitask learners. OpenAI blog **1**(8), 9 (2019)
29. Reynolds, J.C.: Transformational systems and algebraic structure of atomic formulas. Machine intelligence **5**, 135–151 (1970)
30. Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., et al.: Mastering atari, go, chess and shogi by planning with a learned model. Nature **588**(7839), 604–609 (2020)
31. Slind, K., Norrish, M.: A brief overview of hol4. In: International Conference on Theorem Proving in Higher Order Logics. pp. 28–32. Springer (2008)
32. Suda, M.: Vampire with a brain is a good ITP hammer. In: Konev, B., Reger, G. (eds.) Frontiers of Combining Systems - 13th International Symposium, FroCoS 2021. LNCS, vol. 12941, pp. 192–209. Springer (2021). https://doi.org/10.1007/978-3-030-86205-3_11
33. The Coq Development Team: Coq reference manual 8.11.1 (2020), https://coq.github.io/doc/v8.11/refman/index.html
34. Urban, J., Jakubův, J.: First neural conjecturing datasets and experiments. In: International Conference on Intelligent Computer Mathematics. pp. 315–323. Springer (2020)
35. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in neural information processing systems **30** (2017)

36. Wang, M., Tang, Y., Wang, J., Deng, J.: Premise selection for theorem proving by deep graph embedding. Advances in neural information processing systems **30** (2017)
37. Wang, Q., Brown, C.E., Kaliszyk, C., Urban, J.: Exploration of neural machine translation in autoformalization of mathematics in mizar. In: Blanchette, J., Hritcu, C. (eds.) Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020. pp. 85–98. ACM (2020). https://doi.org/10.1145/3372885.3373827
38. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al.: Huggingface's transformers: State-of-the-art natural language processing. arXiv preprint arXiv:1910.03771 (2019)
39. Wu, Y., Jiang, A.Q., Li, W., Rabe, M.N., Staats, C., Jamnik, M., Szegedy, C.: Autoformalization with large language models. arXiv preprint arXiv:2205.12615 (2022)
40. Wu, Y., Rabe, M.N., Li, W., Ba, J., Grosse, R.B., Szegedy, C.: Lime: Learning inductive bias for primitives of mathematical reasoning. In: International Conference on Machine Learning. pp. 11251–11262. PMLR (2021)
41. Yang, K., Deng, J.: Learning to prove theorems via interacting with proof assistants. In: International Conference on Machine Learning. pp. 6984–6994. PMLR (2019)
42. Zhang, L., Blaauwbroek, L., Piotrowski, B., Černý, P., Kaliszyk, C., Urban, J.: Online machine learning techniques for Coq: A comparison. In: International Conference on Intelligent Computer Mathematics. pp. 67–83. Springer (2021)
43. Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., Fidler, S.: 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. pp. 19–27 (2015)