Gpass: a Goal-adaptive Neural Theorem Prover based on Coq for Automated Formal Verification

1st Yizhou Chen
Key Lab of HCST (PKU), MOE
School of Computer Science, Peking University
Beijing, China
yizhouchen@stu.pku.edu.cn

2nd Zeyu Sun
National Key Laboratory of Space Integrated Information System
Institute of Software, Chinese Academy of Sciences
Beijing, China
zeyu.zys@gmail.com

3rd Guoqing Wang
Key Lab of HCST (PKU), MOE
School of Computer Science, Peking University
Beijing, China
guoqingwang@stu.pku.edu.cn

4th Dan Hao
School of Computer Science, Peking University;
Zhongguancun Laboratory
Beijing, China
haodan@pku.edu.cn

***Abstract*—Formal verification is a crucial means to assure software quality. Regrettably, the manual composition of verification scripts proves to be both laborious and time-consuming. In response, researchers have put forth automated theorem prover approaches; however, these approaches still grapple with several limitations. These limitations encompass insufficient handling of lengthy proof steps, difficulty in aligning the various components of a Coq program with the requirements and constraints of the proof goal, and inefficiencies. To surmount these limitations, we present Gpass, a goal-adaptive neural theorem prover based on deep learning technology. Firstly, we design a unique sequence encoder for Gpass that completely scans previous proof tactics through multiple sliding windows and provides information related to the current proof step. Secondly, Gpass incorporates a goal-adaptive feature integration module to align the reasoning process with the requirements of the proof goal. Finally, we devise a parameter selection method based on loss values and loss slopes to procure parameter sets with diverse distributions, thereby facilitating the exploration of various proof tactics. Experimental results demonstrate that Gpass attains better performance on the extensive CoqGym benchmark and proves 11.03%-96.37% more theorems than the prior work most closely related to ours. We find that the orthogonality between Gpass and CoqHammer proves their complementary capabilities, and together they prove a total of 3,774 theorems, which is state-of-the-art performance. In addition, we propose an efficiency optimisation approach that allows Gpass to achieve performance beyond Diva at one-sixth of the parameter sets.**

***Index Terms*—Automated Formal Verification, Proof Synthesis, Deep Learning**

I. Introduction

Ensuring the correctness of software systems is crucial. Formal verification methods provide a rigorous approach to validating software, offering a heightened level of assurance and aiding in the early identification of potential errors and security vulnerabilities. Currently, a variety of Interactive Theorem Provers (ITPs) are widely used in the verification community, including Coq [1], Agda [2], and Isabelle/HOL [3]. These tools have been instrumental in verifying critical properties of a range of software systems. For instance, Coq has been used to verify the correctness of compilers [4], Agda has facilitated the verification of complex operating systems [5], and Isabelle/HOL has been employed in verifying the properties of database systems [6] and distributed systems [7].

However, manually crafting proof scripts for verification remains labor-intensive and often daunting, even with the aid of ITPs. For example, verifying the C compiler CompCert required six person-years and over 100,000 lines of Coq code, with the Coq proof being more than three times larger than the compiler itself [4]. Similarly, it took 11 person-years to create proof scripts for verifying a microkernel [5]. Despite the substantial costs associated with formal verification, research on compilers has demonstrated that CompCert exhibits significantly greater robustness compared to its unverified counterparts. Therefore, reducing the costs of program verification is of considerable value.

Imagine a scenario where a simple click of a button generates a proof for a given theorem, significantly reducing the cost and effort associated with formal verification. In pursuit of this vision, many researchers have developed Automated Theorem Provers (ATPs). The first line of research focuses on a collection of tools commonly referred to as ”hammer” tools, such as CoqHammer [8]. These tools leverage precomputed mathematical facts to ”hammer out” proofs. For instance, CoqHammer can automatically prove 26.6% of the theorems found in open-source Coq projects [8]. However, these hammers are limited by their reliance on precomputed facts and their inability to handle induction and other advanced proof techniques, which constrains their overall capabilities. In contrast, the second line of research explores the application of Deep Learning (DL) techniques to model existing proof scripts and acquire higher-order proof methods. This approach aims to

HCST: High Confidence Software Technologies
MOE: Ministry of Education
Corresponding author: Dan Hao (Key Lab of HCST (PKU), MOE), haodan@pku.edu.cn

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00116

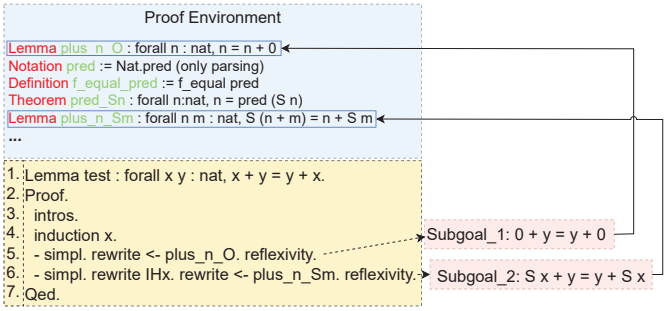

Fig. 1. An example represents the relevance of a proof goal to elements in the proof environment.

guide metaheuristic searches and facilitate the synthesis of new proof scripts [9]–[12]. Recent advancements in this area have achieved a proof success rate of 33.8% by combining DL techniques with traditional methods [13]. Despite these promising developments, several challenges remain unaddressed.

Problem of Lengthy Proofs: Coq requires that each proof step explicitly specify how the rules of inference are to be applied. Complex theorem proving necessitates many intermediate steps to reach the final proof goal. This makes proofs in Coq very lengthy. Previous work [9], [11], [13] primarily focused on the proof tactics immediately surrounding the current proof steps by using RNN-based models to encode proof scripts. These works typically consider only the last 30 tokens of the proof script due to the long dependency problem [14], which likely neglects the influence of earlier proof scripts. Consequently, this often leads to excessive reliance on the current proof state when proving more intricate theorems, resulting in circular reasoning or the generation of redundant proof steps, thus elongating and impeding the efficiency of the proof process.

Problem of Reasoning Alignment: all reasoning processes are designed to address and approximate the proof goal. Indeed, proof goals often exhibit dependencies on other components of the Coq program, such as definitions, lemmas, or theorems. Fig. 1 illustrates an example where various lemmas and theorems about natural numbers are initialized in the proof environment of Coq[1]. When proving the law of exchange for addition, *Subgoal_1* requires a dependency on the lemma *plus_n_O* (Line 5), and *Subgoal_2* requires a dependency on the lemma *plus_n_Sm* (Line 6). However, previous methods [13], [15] have primarily incorporated proof goals as part of the model inputs without adequately correlating the dependency between the current proof goal and other components. In other words, they cannot precisely identify the proof constraints, and lemmas associated with the current proof goal. This can lead to the model requiring additional time and computational resources to discover an effective proof tactic or it may deviate from the optimal proof tactic.

Problem of Integration Efficiency: the state-of-the-art ATP, Diva [13], employs the Coq prover as an oracle and integrates more than sixty models with varying hyper-parameters to maximize the generation of correct proof tactics. This means that the Diva method requires repeatedly training the model and integrating it into the ultimate ATP. Despite the optimization scheme proposed in their study, such a large model integration system still seriously affects the verification efficiency of ATP.

In these scenarios, we propose Gpass, a Coq-based goal-adaptive neural theorem prover for automated formal verification, as a solution to address these limitations. **For lengthy proofs problem**, we exploit valuable information from earlier proof states and proof tactics. We sample five key components of the Coq program (including proof environment, proof local context, proof goal, proof script, and proof term) and encode them according to corresponding data structures. For the proof states (i.e., proof environment, proof local context, and proof goal), Gpass uses a multi-layer Graph Neural Network (GNN) to learn structural and semantic information. For lengthy proof scripts and proof terms sequence, we propose a sequence encoder that overcomes the limitation on encoding length. It can encode proof scripts for thousands of tokens and provides information relevant to the current proof step. **For reasoning alignment problem**, we propose a series connection of goal-adaptive feature integration modules. It takes the proof goal as the central interaction point, integrating information from the proof state. Formally, this module assigns weights to information within the proof state via the proof goal, enabling the model to focus on the most relevant information for achieving the proof goal. **For the integration efficiency problem**, compared to Diva, Gpass acquires multiple parameter sets of diversified distributions through only a single model training, enabling the exploration of various proof tactics. This is achieved by allowing the exploration of different local optima or feature distributions through multiple time steps within a single training. In this study, we propose a parameter selection method for weighted loss values and loss slopes, which enables the accurate selection of parameter sets exhibiting different distributions. Consequently, Gpass can achieve optimal proof results by integrating only a small number of models.

We evaluate Gpass on the large-scale dataset CoqGym [9], which consists of 68,501 theorems extracted from 122 open-source software projects written in Coq. The experimental results show Gpass proves 11.03%-96.37% more theorems than the prior work most closely related to ours. In terms of efficiency, our Gpass offers great improvements in both training efficiency and detection efficiency. Specifically, Gpass only needs to be trained once to obtain an integrated system with diverse distributions (the state-of-the-art Diva requires more than 60 times). Notably, we propose a parameter selection method that requires only 10 parameter sets to be integrated for optimal performance (Diva requires more than 60 ones). It is important to note that the manual writing of proof scripts for formal verification is an arduous task. Thus, even minor enhancements in proof power can yield significant outcomes.

In summary, this paper makes the following contributions.

- **A goal-adaptive proof assistant Gpass**, which utilizes

[1]https://coq.inria.fr/doc/V8.19.2/stdlib/Coq.Init.Peano.html

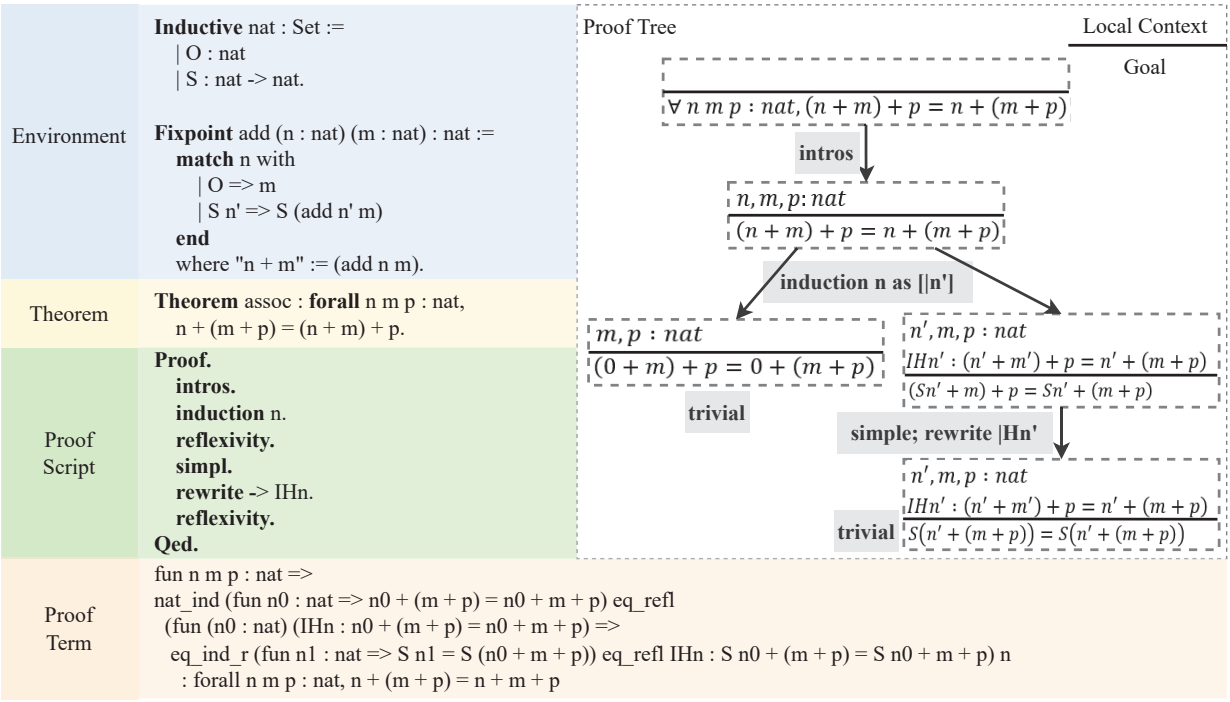

Fig. 2. A simple Coq proof for the associativity of the addition operation on natural numbers.

the interaction of proof goals and other components of the Coq program to align the requirements and constraints between the reasoning process and the goals.

- **An extensive experiment** on a large-scale dataset of 68501 theorems, which shows the effectiveness of Gpass.
- **An optimization** for improving Gpass's efficiency.
- **A reproducible package** available at https://github.com/chenpp1881/Gpass.

II. Preliminary

A. Background on Coq

Coq, an ITP introduced by Barras et al. [1], has gained significant popularity due to its active community and wide range of applications. It has been extensively utilized in the development of certified software and hardware, as well as in mathematics [16]. In Coq, the proof process resembles a task-oriented dialogue, where the user interacts with the prover to complete the proof. The user starts with a theorem as the initial goal and decomposes the goal into a list of sub-goals using a series of proof tactics. The process continues until the proof is complete, which means that all sub-goals have been proved. In Fig. 2, we attempt to prove a theorem about the associativity of addition for natural numbers by using induction. A complete proof process encompasses the environment, theorem to be proven, and the proof script. Successful Coq proofs implicitly generate an AST for the proof, as depicted on the right side of Fig. 2. All goals share a common environment, but each node has a unique local context; edges in the proof tree represent tactics. When a theorem is proven, Coq constructs a proof term using the internal language Gallina [17].

In general, the proof process requires the user to experiment with different tactics in order to decompose the current goal, carefully analyze the feedback provided by Coq, and occasionally backtrack to previous steps to explore alternative tactics. From the perspective of the DL, theorem proving in Coq can be seen as a task-oriented dialogue where the model interacts with the Coq to complete the proof. At each step, a model can utilize the feedback provided by Coq, including the goal, local context, and global environment, to predict appropriate tactics for the next proof step until the proof is eventually completed. The dialogue between the model and Coq can facilitate the DL model to continuously refine its tactics, gradually converging towards an optimal solution for the proof.

The following Coq-related terms are employed in this study for the purpose of facilitating comprehension, and their specific meanings are as delineated below:

- **Proof tactic** is a specific command or instruction used to guide the Coq on how to advance the proof. The purpose of Tactics is to change the proof from one state to another, bringing the proof closer to completion. Common proof tactics include "intros", "apply", "simpl", "rewrite", etc.
- **Proof step** is a specific operation performed each time a proof tactic is applied. Each proof step changes the current proof state. Proof steps are the basic units of constructing proofs. For example, applying the "intros" tactic to introduce premises into the context constitutes a proof step.
- **Proof environment** refers to the interactive environment in Coq where proofs are developed. This includes all the lemmas and contexts for writing and executing Coq

scripts.

- **Proof local context** consists of the assumptions and contexts currently available during the proof process. It includes all introduced variables, assumptions, and intermediate subgoals generated during the proof.
- **Proof goal** is a statement or proposition that needs to be proven. In Coq, the proof process involves transforming the current proof goal into simpler subgoals until they can be directly proven.
- **Proof state** is the collection of the proof environment, the local context, and the current goals. It represents the complete status of the proof at any given point. The proof state changes dynamically as proof steps are applied.

B. Problem Definition

Let us consider a Coq proof with a current proof goal, denoted as G. The local context, denoted as L, contains the hypotheses and assumptions available for the current proof. The global environment, denoted as E, provides information about the existing definitions, lemmas, and theorems that can be used during the proof. And a part of the proof script S and the corresponding proof term T. The goal is to train a DL model M that takes G, L, E, S, and T as inputs and generates an appropriate proof tactic, denoted as Tac, for the next proof step. The real tactic function is denoted as $f : (G, L, E, S, T) \rightarrow Tac$. Formally, we aim to learn a mapping function $M : (G, L, E, S, T) \rightarrow Tac$ to approximate f as accurately as possible. During the training process, the model learns to capture the underlying patterns and dependencies between these components and tactics. Once the training is complete, the DL model can assist Coq users in generating proof tactics, which can then be inputted back into Coq. This iterative process continues until the proof is complete or times out.

III. Our Approach

In our study, Coq is used as a verification tool for programs. In order for the model to better understand and handle programs in Coq, we divide it into five key parts, including proof environment, proof local context, proof goal, proof script, and proof term. To effectively utilize this information, we adopt a multi-input encoder-decoder architecture. Fig. 3 provides an overview of Gpass, which consists of four pivotal modules:

1. **Multi-input Encoder:** the various components of the Coq program are encoded separately using distinct modal encoders based on the data structure. The proof environment, proof local context, and proof goal are initially parsed into ASTs, which are fed into a graph encoder; the proof scripts and proof terms are fed into a sequence coder that is designed to handle lengthy sequences.
2. **Goal-adaptive Feature Integration:** after encoding each component, interacting the proof goal with each component, aligning the specific requirements and constraints imposed by the reasoning process with the proof goal.
3. **Tactic Decoder:** the integrated features are fed as input to the decoder. The decoder sequentially expands a constraint-based AST and parses it into subsequent tactics.
4. **Interactive Proof:** the generated proof tactic interacts with the Coq prover to verify its correctness. In addition, Gpass establishes multiple sets of parameters for interactive verification with the Coq prover to generate diverse proof tactics.

A. Multi-input Encoder

The multi-input encoder takes as input the different components of Coq programs, including the proof environment, proof local context, proof goal, proof script, and proof term. By capturing the rich semantic and structural information present in Coq programs, the multi-input encoder lays the foundation for subsequent tactic generation. To handle the diverse data structures, we introduce two encoders: the graph encoder and the sequence encoder.

1) Graph Encoder: The graph encoder is designed to handle the proof environment, proof local context, and proof goal, which can be parsed as ASTs. Traditionally, models such as TreeLSTM have been employed to encode these tree structures. However, it processes information strictly in a top-down manner, which limits the model's ability to effectively capture cross-level dependencies and relationships of sibling nodes. GNNs, on the other hand, offer a more flexible and powerful framework for encoding ASTs. GNNs treat ASTs as general graphs, allowing for more comprehensive information propagation and aggregation across the entire structure. This flexibility enables GNNs to capture intricate dependencies and relationships within the proof environment, proof local context, and proof goals more effectively than TreeLSTMs. Therefore, in this work, we use GNN to encode these ASTs. Its workflow is shown in the Fig. 4.

We first define $G = (V, E)$ as a graph that represents an AST generated by the proof state. Here, V denotes the set of nodes and E represents the set of edges. Each node $v \in V$ has a feature vector representation x_v, which can be combined into a feature matrix $X \in \mathbb{R}^{n \times d}$, where n is the number of nodes and d is the dimensionality of the feature vector. The first step in GNN is to compute the aggregation of the features of a node's neighbors. Assuming that the set of neighbor nodes of each node v is $N(v)$, then the aggregation of neighbor features of node v can be expressed as:

$$\tilde{h}_v = \sum_{u \in N(v)} \frac{1}{\sqrt{\deg(v) \cdot \deg(u)}} h_u \tag{1}$$

where $\deg(v)$ is the degree of node v, and h_u represents the feature vector of node u.

Next, we aggregate feature $\tilde{h}_v$ to generate a new feature representation for each node:

$$h_v = \sigma(W\tilde{h}_v + b) \tag{2}$$

where W is a learnable weight matrix, b is a bias vector, and $\sigma(\cdot)$ is a non-linear activation function (e.g., ReLU). Iterate the above steps multiple times to update the feature representation of the nodes. This allows the features to propagate to more

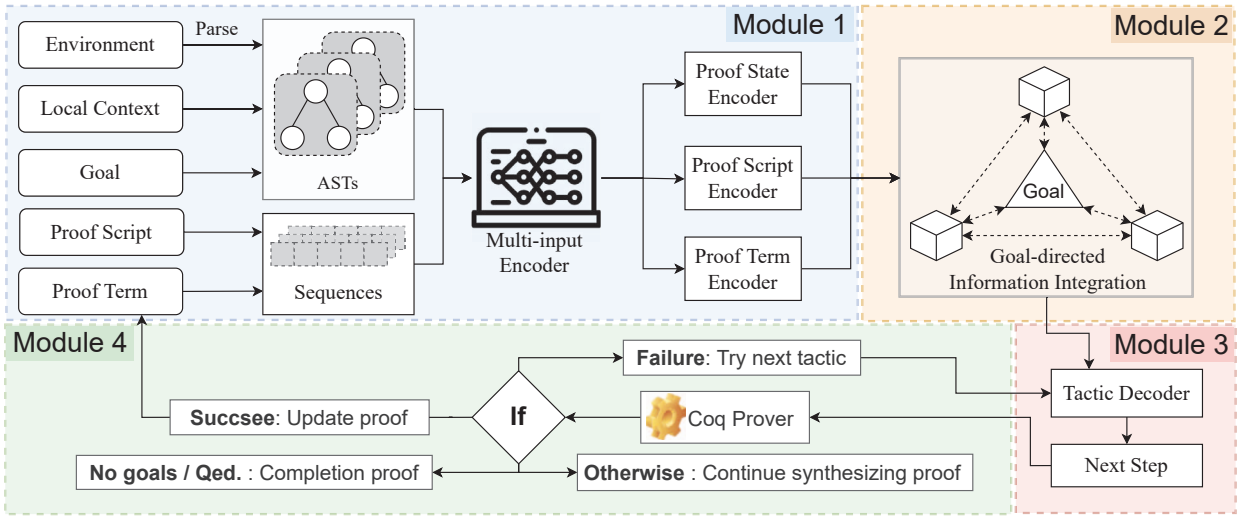

Fig. 3. An overview of our Gpass.

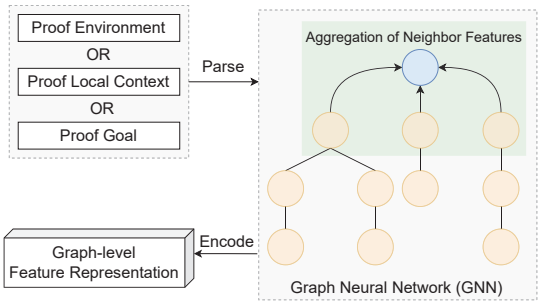

Fig. 4. An AST encoder for proof states.

distant neighbor nodes, capturing a piece of broader context information. Finally, the node features can be aggregated and subsequently inputted into a fully connected layer to obtain graph-level feature representation.

2) Sequence Encoder: Our sequence encoder is specifically designed to address the challenge of handling lengthy proof scripts and proof terms effectively. As the proof process progresses, the proof script and terms can become significantly longer, with some proof scripts or terms reaching thousands of tokens. Previous studies [9], [11], [13] often struggle to encode such extensive sequences effectively, typically focusing only on the last 30 tokens, which can overlook crucial information from earlier proof steps. In contrast, our approach incorporates a novel sequence encoder that can manage these long sequences comprehensively. By capturing a broader context, including earlier proof steps, our method addresses the limitations of existing approaches.

The workflow is shown in Fig. 5. In general, previous proof scripts or terms are not always valuable for subsequent proof tactics. Our focus should only be on scripts or terms that are relevant to the subsequent proof tactic. Therefore, the sliding window technique is employed to extract relevant information by Convolutional Neural Network (CNN) [18]–[20]. Specifically, we set up L filters built into the CNN, and each filter has J convolution kernels. The advantage is that filters with different window sizes can learn pieces of information of different dimensions, where multiple convolution kernels enable the learning of mutually complementary features within the script or term sequence. Finally, all the important features obtained from the 1-max pooling operation are concatenated to form a feature matrix. This feature matrix serves as the input for the subsequent layers in the neural network. The specific mathematical formulas are as follows.

$$C^{j,l} = \text{Relu}(W^{j,l} \bullet E_{i:i+h_l-1} + b) \tag{3}$$

where $W^{j,l} \in \mathbb{R}^{(h,k)}$ denotes the j-th $(j \in J)$ convolution kernel of the l-th $(l \in L)$ filter with the height h_l. The $C^{j,l} \in \mathbb{R}^{(1,n-h_l+1)}$ represents a new feature map via a convolution calculation, and b is a bias. The P elements with the highest weight $[\widehat{C_1^{j,l}}, \widehat{C_2^{j,l}}, ..., \widehat{C_P^{j,l}}]$ are concatenated into a new feature vector, which represents the set of features associated with the subsequent proof tactic, named $M \in \mathbb{R}^{(J*L,P)}$:

$$M = \begin{pmatrix} \widehat{C}_1^{1,1} & ... & \widehat{C}_p^{1,1} & ... & \widehat{C}_P^{1,1} \\ ... & ... & ... & ... & ... \\ \widehat{C}_1^{j,l} & ... & \widehat{C}_p^{j,l} & ... & \widehat{C}_P^{j,l} \\ ... & ... & ... & ... & ... \\ \widehat{C}_1^{J,L} & ... & \widehat{C}_p^{J,L} & ... & \widehat{C}_P^{J,L} \end{pmatrix}. \tag{4}$$

The attention mechanism [21] is then employed to facilitate the interaction of information among feature points in M. The detailed equation is as follows:

$$M' = \text{FFN}\left(\text{Softmax}(\frac{qk^T}{\sqrt{d}})v\right), \tag{5}$$

where queries q, keys k, and values v are the matrices generated by different linear transformations of the input M; a feed-forward network $\text{FFN}(\cdot)$ contains two transformation layers and an activation function (ReLU); M' is finally the sequence-level feature representation.

B. Goal-adaptive Feature Integration

In Coq, all proofs are purposefully designed to provide guidance towards achieving the proof goal. Our intuition

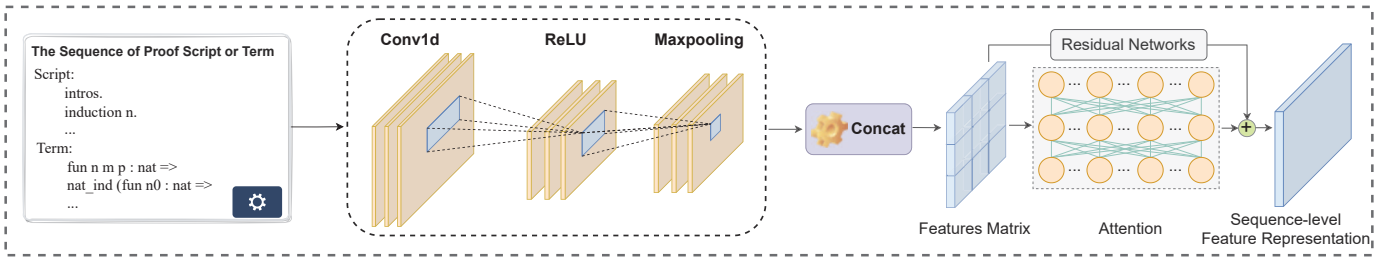

Fig. 5. A sequence encoder for proof scripts and terms.

is that by analyzing the interplay between proof goals and various components of the Coq program, Gpass can effectively align the reasoning process with the specific requirements and constraints imposed by these proof goals. Notably, proof goals often exhibit dependencies on other components within the Coq program, such as definitions, lemmas, or axioms. By leveraging these dependencies, Gpass allows the reasoning process to be guided more effectively to meet the specific conditions required by the proof goals. The previous approaches [13], [15] have merely incorporated proof goals as part of the model inputs, without delving into the dependencies between proof goals and elements within the other components.

Therefore, following the encoding of the individual components, the outputs of the encoders are integrated using a goal-adaptive feature integration module. Here, the feature representation of the proof goal is leveraged as a central interaction point. The encoder outputs of the proof environment, proof local context, proof goal, proof script, and proof term are denoted as $[A_e, A_l, A_g, A_s, A_t]$ respectively.

Here, we use the internal product of the proof goal and each of the elements in the other components as the guide score. The size of the score reflects the degree of dependency between the target element and the proof goal. The guide score is then multiplied with each element to determine the final feature representation. The feature representations of the proof goal and other components interact through the following mathematical process:

$$A_i' = A_i^T \cdot \mathrm{Softmax}\left(W_i A_i \cdot W_g A_g\right), \tag{6}$$

where $i \in \{e, l\}$. W_i and W_g are the weights of the linear transformation, respectively. $[A_e', A_l']$ are the goal-guided feature vectors. All the feature vectors are combined together to form a new feature matrix $\widetilde{A} = [A_e'^T, A_l'^T, A_g^T, A_s^T, A_t^T]^T$. Finally, we pass the matrix $\widetilde{A}$ into an attention mechanism and use its output as input for the tactic decoder.

C. Tactic Decoder

The tactic decoder aims to generate target tactics based on the input features. Fig. 6 shows an overview of the tactic decoder. Within the tactic decoder, the input corresponds to the output of the feature integration module. By sequentially expanding an AST, the decoder generates a representation of tactic [22]. At each non-terminal node, AST selects a production rule from a specified context-free grammar that defines the space of tactics. At a terminal node, a node is synthesized

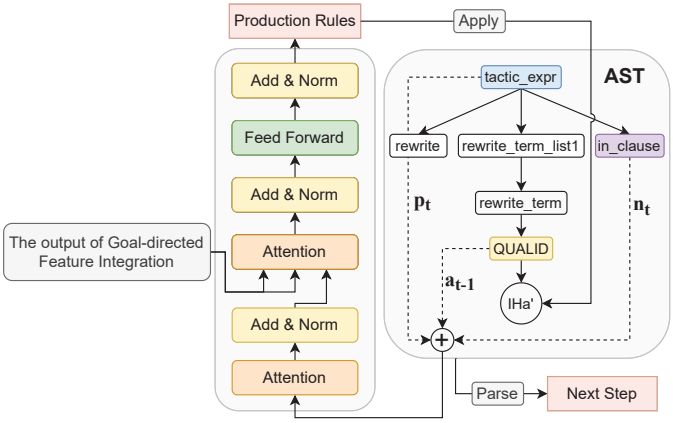

Fig. 6. The workflow of tactic decoder.

based on semantic constraints through the controlled process of tree growth by an attention module. Specifically, at time t, n_t represents the symbol of the current node, a_{t-1} denotes the production rule for expanding the prior node, and p_t indicates the state of the parent node. The n_t, a_{t-1}, and p_t are concatenated within the tactic decoder to interact with the output of the feature integration module for predicting subsequent tactics.

D. Interactive Proof

The field of formal verification presents a unique application of DL, offering the promise of correctness predictions. Unlike most DL domains where the notion of model correctness remains elusive, formal verification has a distinct advantage. Here, interactive theorem prover assume the role of oracles, guiding the synthesis of proof scripts for each theorem. A proof script is considered correct if it successfully leads the theorem prover to reach a proof termination in Qed. This property allows ATPs to discard unsuccessful attempts while retaining successful ones. Thus, First et al. [13] used ensemble learning [23]–[25] to integrate more than 60 models into an automated proof system, which was used to improve the proof capability and achieved satisfactory results. However, training these models is very costly in terms of resources and time.

Essentially, the purpose of training multiple models is to obtain diverse feature distributions thereby exploring various proof tactics. It is worth mentioning that we can observe the phenomenon of diversification of feature distributions even at different stages of a single model training. This is because the

model gradually adjusts the feature distributions as the training proceeds, and these different stages of feature distributions provide us with a window to explore diverse proof tactics. Consequently, while training the model, we save the parameter sets. At the end of the training process, we obtain multiple sets of parameters that reflect different feature distributions. Leveraging these diverse parameter sets allows us to explore various synthesis tactics. The testing of all parameter sets, however, would incur significant time costs. We need to select the parameter set purposefully. Our intuition is that the absolute values of loss values and loss slopes are critical. First, choosing parameters with lower loss values ensures that the model will fit the target better on the training data, thus ensuring model performance. Second, the absolute value of the loss slope provides information about the inconsistency of the current parameter distribution with the previous parameter distribution, which means that the model may try different synthesis tactics.

Algorithm 1: Parameter Set Selection

Input: Loss values L_i, weights w_{loss}, w_{slope}, number of parameter sets P
Initialize empty lists N_L, N_S, $Scores$, $L_0 = 0$
Compute normalized loss values
for i *in range(*N*)* **do**
  $N_L[i] \leftarrow \frac{L_i - \min(L)}{\max(L) - \min(L)}$
end
Compute loss slopes S_i from L_i
for i *in range(*N*)* **do**
  $S_i \leftarrow |L_i - L_{i-1}|$
end
for i *in range(*N*)* **do**
  $N_S[i] \leftarrow \frac{S_i - \min(S)}{\max(S) - \min(S)}$
end
Compute $Scores$ **for** i *in range(*N*)* **do**
  $Score_i \leftarrow w_{\text{slope}} \cdot N_S[i] - w_{\text{loss}} \cdot N_L[i]$
  Add $Score_i$ to the $Scores$ list
end
Select parameter sets with the top P scores for testing
return Selected parameter sets

Specifically, as described in Algorithm 1, we collect the loss values across multiple time steps and compute the absolute value of their loss slopes. Subsequently, both the loss values and loss slopes are normalized to ensure a consistent scale for comparison. Then, the loss values and loss slopes are integrated using a weighted approach, enabling the determination of scores. Lastly, a top-down selection of P parameter sets is employed for tactic synthesis, based on the scores.

IV. Experimental Settings

We conduct evaluations of our proposed approach to address the following Research Questions (RQ):

RQ1: *How effective is Gpass compared with the state-of-the-art approaches in proof script synthesis?*

RQ2: *How does Gpass compare to the baseline approaches in terms of orthogonality?*

RQ3: *How does the choice of parameter set in interactive proofs affect the effectiveness of Gpass?*

RQ4: *How much do the different components of Gpass contribute to overall performance?*

A. Datasets

We utilize CoqGym [9], an state-of-the-art benchmark that has been employed in previous evaluations of ATPs. It encompasses a collection of 70,856 theorems extracted from 123 open-source software projects written in Coq. At a later date, First et al. [13] excluded some projects that would have triggered errors within the Coq. Overall, our evaluation encompasses a comprehensive total of 68,501 theorems derived from both training and test sets involving 122 projects. We keep the same set of splits as in the previous work, in which consists of a training set of 96 projects containing 57,719 manually-written proof scripts, and a test set of the 10,782 theorems from 26 projects.

B. Baselines

Our evaluation will compare Gpass to three ATPs, CoqHammer [8] and two neural-based approaches (i.e., ASTactic [9], TacTok [11], and Diva [13]). In a nutshell, Coqhammer is an automated tool used to assist Coq proof, which provides functions such as tactic search and automated inference to help users more efficiently perform interactive theorem proving. ASTactic is a search-based automated verification tools that use the DL model to predict the next tactic from the current proof state. TacTok is an improved version of ASTactic that has incorporated a bi-directional LSTM [26] to learn a series of proof scripts. Diva is a state-of-the-art automated theorem prover that effectively generates a diverse range of models by meticulously controlling input data and model parameters. It enhances its proof capability through continuous interaction with the Coq prover.

C. Evaluation Metrics

To measure the effectiveness of our Gpass, we quantify two metrics to fairly compare it with other baseline methods.

Success Rate of a tool, widely used in prior evaluations [9], [11], [13], [27], can be mathematically represented as the fraction of all theorems for which the tool generates a successful proof script:

$$\text{Success Rate} = \frac{N_S}{N_T}, \tag{7}$$

where N_S is the number of theorems with successful proof scripts; and N_T is the number of theorems with all proof scripts.

Added Value of tool A over tool B can be mathematically represented as the ratio of the number of new theorems that tool A proves but tool B does not to the total number of theorems that tool B proves:

$$\text{Added Value} = \frac{N_{a\neg b}}{N_b}, \tag{8}$$

TABLE I
QUANTITATIVE EXPERIMENTAL RESULTS INCLUDE THE ABSOLUTE VALUES, SUCCESS RATES, AND ADDED VALUES OF THEOREMS PROVED BY GPASS, DIVA, ASTACTIC, TACTOK, AND COQHAMMER, ACROSS THE THEOREMS IN THE COQGYM BENCHMARK.

Tool	Theorems Proven	Success Rate	Added Value
ASTactic	1,322	12.26%	1,340 (101.30%)
TacTok	1,388	12.87%	1,280 (92.21%)
Diva	2,338	21.68%	420 (17.96%)
CoqHammer	2,865	26.57%	909 (31.72%)
Gpass	2,596	24.07%	-
Gpass & CoqHammer	**3,774**	**35.00%**	-

where $N_{a\neg b}$ is the number of new theorems proved by tool A but not by tool B; N_b is the total number of theorems proved by tool B.

D. Implementation Details

For the graph encoder, we aggregate the 6-hop neighbor nodes of each node and stack 6 isomorphic network layers. It employs four filters with different window sizes which are mutually prime to ensure that no redundant information is learned, including filter heights of 3, 5, 7, and 11. Each filter is equipped with 64 internal convolution kernels, which are in steps of 1. The attention layer is recycled 6 times. For the interactive proof module, the weights w_{slope} and w_{loss} are set to 0.5. We save the parameter set of 100 time steps and the top P parameter sets are chosen for interactive theorem proving.

During training, the learning rate is initialized to 1 x 10^{-5} and is optimized using the AdamW optimizer. The batch size is fixed at 128. All word embeddings have the dimension 256. For all the baselines, we utilize the complete code from their provided open-source libraries and adhere to the configurations specified in their research. The experiments are conducted using hardware resources that included 4 Nvidia RTX 3090 GPUs with 96GB video memory. These GPUs are utilized in parallel for training. For the software environment, we employ Ubuntu 20.04 LTS as the operating system.

V. EXPERIMENTAL RESULTS

A. RQ1: How effective is Gpass compared with the state-of-the-art approaches in proof script synthesis?

To evaluate the effectiveness of Gpass, we compared it against five state-of-the-art baselines. Specifically, we divide the 57,719 theorems used for training into 189,824 proof steps. Each proof step consists of six components: the proof environment, local proof context, proof goal, partial proof script, and proof terms; and the next-step proof tactic, which serves as the label for supervised learning during model training. As described in Section III-D, we select the top 10 parameter sets based on their scores for testing on the remaining 10,782 theorems. The results are reported in Table I. For all added values, the calculation represents the ratio of the number of theorems proven by Gpass to the number of theorems proven by other methods.

For the table I, we observe that Gpass achieves the best performance among the neural-based approaches, proving 2,596 theorems with a success rate of 24.07%. In comparison, ASTactic, which only utilizes proof environment, local proof context, and proof goal for model training, proves 1,322 theorems with a success rate of 12.26%. TacTok, building upon ASTactic by incorporating partial proof scripts during training, successfully proves 1388 theorems with an improved success rate of 12.87%. Diva, further enhancing TacTok by incorporating Coq's internal language and proof terms for training and utilizing the Coq prover as an interactive oracle, ultimately proves 2,338 theorems with a success rate of 21.68%. From another perspective, Gpass proves 1,340 theorems that ASTactic fails to prove, demonstrating an added value of 101.30% compared to ASTactic. Similarly, it proves 1,280 theorems that TacTok fails to prove, providing an added value of 92.21%, and proves 420 theorems that Diva fails to prove, providing an added value of 17.96%.

Additionally, we do not expect Gpass to prove more theorems than CoqHammer, which can prove some complementary theorems compared to neural-based approaches. However, we find that both tools can complete proofs that the other cannot, allowing users to combine the two tools to generate more solutions than using either tool alone. This is demonstrated in the experiments shown in Table I. Specifically, CoqHammer proves 2865 theorems with a success rate of 26.57%. Among the theorems proved by Gpass, 909 theorems are unprovable by CoqHammer, resulting in an added value of 31.72% for CoqHammer. By combining both methods, a total of 3774 theorems are jointly proved with a success rate of 35.00%.

Finally, we meticulously observe the actual performance of Gpass across multiple projects. We find that Gpass handles logical automation and reasoning tasks effectively, especially in structured programming environments like miniML. The performance of Gpass, however, is found to be subpar when dealing with intricate mathematical structures such as hypermaps and combinatorial topology, as well as when handling high-level abstractions in topology.

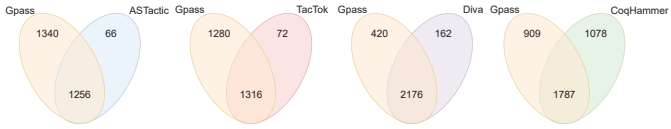

Fig. 7. The orthogonality of Gpass and other APTs in proving theorems.

Answer to RQ1: Gpass achieves the best performance among neural-based ATPs, outperforming the current state-of-the-art ATP, Diva, by proving an additional 420 theorems and achieving an added value of 17.96%. Furthermore, Gpass can be combined with CoqHammer. Experimental results demonstrate that combining Gpass with CoqHammer leads to an improvement in proof capabilities compared to using either method alone. In the end, a total of 3,774 theorems are jointly proven.

B. RQ2: How does Gpass compare to the baseline approaches in terms of orthogonality?

Orthogonality refers to the ability of two methods to independently prove different theorems. Measuring the orthogonality between Gpass and other ATP methods is crucial. It provides insights into the complementarity of their theorem-proving capabilities, guiding researchers in making informed decisions on method selection, integration, and optimization. Fig. 7 illustrates the orthogonality of Gpass with other ATPs.

From the data, we find that Gpass and ASTactic have jointly proven 1,256 theorems. Notably, among the theorems successfully proven by Gpass, 1,340 are not proven by ASTactic. Conversely, ASTactic has only 66 theorems that Gpass has failed to prove. Similarly, Gpass and TacTok have jointly proven 1,316 theorems. However, Gpass has successfully proven 1,280 theorems that TacTok has not. TacTok, on the other hand, has only 72 theorems that Gpass cannot prove. Lastly, Gpass and Diva have jointly proven 2,176 theorems. Nevertheless, there are 420 theorems proven by Gpass that Diva has not proven. Conversely, Diva has 162 theorems that Gpass cannot prove. Overall, our Gpass can cover most of the theorems proved by these ATPs, showing its strong theorem-proving ability. It not only provides effective theorem proving but also complements other ATPs in terms of proving ability.

Remarkably, our analysis reveals a strong orthogonality between Gpass and CoqHammer, manifesting in their complementary theorem-proving capabilities. Specifically, Gpass and CoqHammer jointly verify 1,787 theorems. Gpass succeeds in proving 809 theorems that CoqHammer is unable to address, while CoqHammer proves 1,078 theorems that Gpass cannot handle. Together, the two approaches achieve a remarkable total of 3,774 theorems proved, the highest tally ever recorded. This orthogonality not only reveals the unique theorem-proving abilities of each of Gpass and CoqHammer but also provides us with the possibility of using the two in combination. By fully utilizing their complementarities, we achieve better effectiveness in theorem proving. For example, in some complex theorem proving, we can first try to prove it using Gpass, and if Gpass fails to prove it, then we can turn to CoqHammer to try it. Such a combined use strategy can fully utilize the advantages of both and improve the success rate of theorem proving.

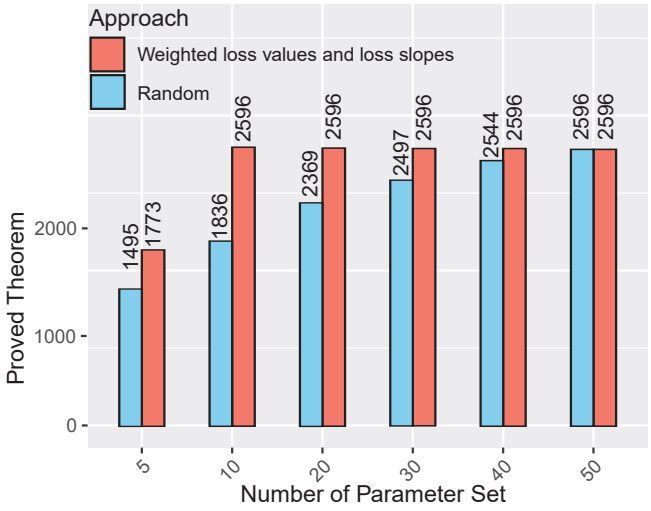

Fig. 8. The effectiveness of parameter selection method in Gpass.

Answer to RQ2: There is some orthogonality between our Gpass and other ATPs. Gpass can cover most of the theorems proved successfully by ASTactic, TacTok, and Diva, and can prove many theorems that they cannot. In addition, the remarkable orthogonality between Gpass and CoqHammer demonstrates their complementary capabilities and combining them to improve the effectiveness of theorem proving.

C. RQ3: How does the choice of parameter set in interactive proofs affect the effectiveness of Gpass?

The impact of model parameter variations on the search-based synthesis of proof scripts cannot be overlooked. The models of different parameters have the potential to generate distinct scripts for the very same theorem. However, testing all parameter sets would be costly in terms of time, so we propose a selection method of parameter set selection method. Here, we verify the effectiveness of the method.

First, in order to provide a baseline, we conduct our tests using a random selection of parameter sets. This approach is able to simulate the process of parameter selection without explicit guidance and provide us with a reference standard. Second, we select parameter groups of different sizes for testing in a top-down manner based on the parameter selection methodology presented in Section III-D. Specifically, we choose five experimental configurations containing 5, 10, 20, 30, 40, and 50 parameter sets. This design aims to comprehensively evaluate the performance of our parameter selection method under different parameter combinations. By gradually increasing the number of parameter groups, we are able to observe the trend of the parameter selection method

TABLE II
THE RESULTS OF ABLATION STUDY.

Methods	Theorems Proven	Success Rate
$\textbf{Gpass}^1_{w/oGoal}$	1,542	14.30%
$\textbf{Gpass}_{w/oGoal}$	2,377	22.05%
$\textbf{Gpass}^1_{w/oSeq}$	1,604	14.87%
$\textbf{Gpass}_{w/oSeq}$	2,458	22.79%
$\textbf{Gpass}^1_{w/oGNN}$	1,644	15.24%
$\textbf{Gpass}_{w/oGNN}$	2502	23.20%
$\textbf{Gpass}^1$	1,692	15.69%
Gpass	**2,596**	**24.07%**

on the experimental results and analyse the applicability of the method accordingly. We strictly control other variables to ensure the accuracy of the experimental results.

From the Fig. 8, we can see that our weighted method achieves the peak results of the proof theorem by integrating only 10 parameter sets. In contrast, a randomly chosen of parameter sets requires integrating of 50 parameter sets to achieve the same result.

Answer to RQ3: Gpass requires only one training session to obtain parameter sets. And our parameter selection method can effectively select those parameter sets with diverse distributions. Satisfactory proof results can only be obtained by integrating only 10 parameter sets.

D. RQ4: How much do the different components of Gpass contribute to overall performance?

In this experiment, we perform an ablation study on Gpass to evaluate the impact of its individual modules on performance. The metrics used for this analysis were Theorems Proven and Success Rate. We start by disintegrating the goal-adaptive feature integration module (the following is referred to as the goal module) in Gpass, denoted as $\text{Gpass}_{w/oGoal}$. Next, we ablate the sequence module and label it as $\text{Gpass}_{w/oSeq}$. Finally, we ablate the GNN model and replace it with the original TreeLSTM, creating the variant $\text{Gpass}_{w/oGNN}$. As described in Section IV-D, the above variants are tested on 10 sets of parameters. To more intuitively reflect the impact of each module on the overall performance, we also report the results of a single set of parameter tests. These variants are labeled Gpass^1_*. The results of the ablation study are presented in Table II.

First, the goal module improves the performance of Gpass. After ablation, the number of proved theorems is reduced by 219, and on a single parameter set, by 150. Second, the sequence encoder also contributes to Gpass's performance. Ablating it reduces the number of proved theorems by 138, and on a single parameter set, by 88. Third, GNN outperforms TreeLSTM. Ablating GNN leads to a reduction of 94 in the number of proved theorems and 48 in the individual parameter sets. As described in Section III-A1, GNN enables more comprehensive information propagation and aggregation throughout the AST, achieving superior performance. For Gpass^1, the observations are similar to those of Gpass, with each module contributing to the performance of Gpass^1.

Answer to RQ4: The goal module, sequence encoder, and GNN all contribute positively to the overall performance of Gpass, while their synergistic effect enhances its effectiveness.

VI. THREATS TO VALIDITY

Threats to external validity are in the dataset used for the experiment. In the experiment, we used a state-of-the-art benchmark, which has been used in the evaluation of existing ATPs. To reduce this threat, we plan to construct a larger dataset for ATP in the future.

Threats to internal validity are in implementing the proposed Gpass and baselines. To reduce these threats, we used PyTorch to implement the DL component of Gpass, and used the reproducible package of each baseline.

Threats to construct validity are in the measurement used in the experiment. The performance of Gpass was evaluated based on the success rate and added values, without considering factors such as readability or simplicity, following prior research [9], [11], [13]. Additionally, similar to previous work, our study is a research prototype that lacks complete integration into Coq. These limitations may be addressed in future endeavors.

VII. RELATED WORK

A. Interactive Theorem Provers

ITPs are a set of tools, which allow users to verify formal proofs. They offer a precise language for expressing mathematical statements and proofs, along with interactive mechanisms that empower users to guide the proof process effectively. Some of the widely used IPTs include Coq [1], Agda [2], Dafny [28], F* [29], Liquid Haskell [30], Mizar [31], Isabelle [3], HOL4 [32], and HOL Light [33] all representing semi-automated systems dedicated to formal theorem proving. While our research primarily centers on theorem proving in Coq, the proposed methodology can be readily applied to a diverse range of other ITPs.

B. Automated Theorem Provers

The ATPs assist in the construction or simplification of proofs, thereby reducing the manual effort required from users. For example, heuristic-based search methods are partially automated ITPs [34]–[36], while tools like Hammers utilize external ATPs to automatically discover proofs [8]. In systems like HOL4 [37], [37], classical search algorithms such as A* can also be employed alongside reinforcement-learning-based approaches to search for proofs [38]. In contrast, Gpass is the novel ATP that models existing proof scripts and synthesizes proofs within the ITP framework.

C. Software Engineering for Interactive Proof Assistants

Pumpkin Patch [39] generates proof patches by learning predefined fix cases during software evolution. Pumpkin Pi [17] repairs the proof term, which is then decompiled to generate the proof script. iCoq [40] focuses on identifying failing proof scripts in software evolution by prioritizing those affected after a revision. QuickChick [41] searches for counterexamples to executable theorems and and gives feedback to the programmer. Our Gpass employs a DL model to accurately predict tactics. It has the ability to generate proof scripts from scratch, rather than having to rely on existing partial scripts.

D. Machine Learning in Formal Verification

The utilization of machine learning techniques is increasingly prevalent in automating and enhancing various aspects of formal verification processes. For example, ML4PG assists Coq users in constructing proof scripts by showcasing proof scripts from similar theorems [42], [43]. NeuroTactic [44] employs GNN to represent theorems and premises for prediction script. Diva [13], ASTactic [9], and TacTok [11] use TreeLSTM [45] to model the mapping relationship between proof states and proof tactics, that evaluate performance in the GoqGym benchmark [9]. Cunningham et al. [46] proposed a semantic parsing approach for auto-formalizing elementary mathematical proofs and simple imperative code into Coq-verifiable representations. Sanchez-Stern et al. [47] presented the Passport approach, which uses three new encoding mechanisms for identifiers to improve the automation of formal verification by proof-synthesis tools. Gpass builds upon concepts from Diva, with evaluations conducted on the CoqGym benchmark. The Gpass, in contrast to Diva, not only learns proof states but also incorporates a more comprehensive modeling of distant dependencies between proof tactics. It also aligns constraints and requirements between proof goals and proof states.

Recent research has increasingly leveraged the Large Language Models (LLMs) to conduct formal verification. Baldur [48] used large language models trained on natural language and code to generate entire proofs at once, supplemented by a fine-tuned repair model for proof correction. COPRA [49] utilized GPT-4 to propose tactic applications within a stateful backtracking search, incorporating execution feedback, search history, and lemmas from an external database to build prompts for subsequent model queries. DSP [50] mapped informal proofs to formal proof sketches, guided an automated prover, and improved its performance by focusing on easier sub-problems. PACT [51] used self-supervised data along with prediction goals for joint training to cope with data scarcity when learning theorem proving by imitation in large-scale formal mathematical libraries. Lyra [52] employed two correction mechanisms, tool correction and conjecture correction, to mitigate hallucinations of LLMs. Yang et al. [53] presented a Lean-based toolkit, LeanDojo, containing data, models, and benchmarks for removing barriers to existing machine learning theorem proving research. Based on this toolkit, the authors develop ReProver, an LLM theorem prover incorporating retrieval capabilities, and demonstrate its effectiveness on a newly constructed benchmark containing 98,734 theorems and proofs. Zhou et al. [54] utilized LLMs to automatically formalize informal mathematical statements into Isabelle code, enabling automatic verification for internal consistency and reducing unjustified logical and computational errors. AutoSpec [55] utilized static analysis, program verification, and LLMs to generate potential specifications. Subsequently, LLMs were employed to validate these specifications. These methods are not designed for Coq and fail to engage with the theorem prover in a step-by-step manner, thereby limiting their ability to fully exploit the dynamic execution information generated during interaction with the prover. In contrast, Gpass can interact with the Coq kernel to obtain information for determining the validity of the resulting proof script and iteratively predicts subsequent proof steps.

VIII. Conclusions and Future Work

The present paper introduces Gpass, a novel technique for ATP. The Gpass introduces specialized encoders for different components of the Coq program, employing GNNs for AST-based encodings and innovative sequence encoders for handling proof scripts and proof terms. In addition, it aligns the requirements and constraints between proof goals and proof states through a goal-adaptive feature integration module. By leveraging these modules, Gpass enhances the understanding and verification capabilities of Coq programs, facilitating the automated generation of proof programs. In future research, we aim to further validate the performance of Gpass by increasing the sample size of theorem proving and exploring its application in generating other ITP programs. Gpass is a research prototype. Ideally, given the context of a theorem and the object of the proof, it can generate a complete proof process. However, there are still some gaps from the ideal state, such as the need for tools to automatically extract context. Our method runs in a configured environment, but lacks the tools to automatically extract the context. Further, we propose several strategic developments: the creation of an intuitive graphical user interface, the integration of seamless workflows, and the provision of comprehensive documentation. Following up, we are committed to transitioning Gpass from a research prototype to a fully integrated tool for automated theorem proving within Coq.

Acknowledgment

This work was supported by the National Natural Science Foundation of China under Grant No. 62372005 and No. 62402482.

References

[1] D. L. Bottorff, "Coq systems: the right stuff," *Quality progress*, vol. 30, no. 3, p. 33, 1997.

[2] P. Wadler, "Programming language foundations in agda," in *Formal Methods: Foundations and Applications: 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26–30, 2018, Proceedings 21*. Springer, 2018, pp. 56–73.

[3] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[4] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[5] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.

[6] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky, "Toward a verified relational database management system," in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2010, pp. 237–248.

[7] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: a framework for implementing and formally verifying distributed systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 357–368.

[8] Ł. Czajka and C. Kaliszyk, "Hammer for coq: Automation for dependent type theory," *Journal of automated reasoning*, vol. 61, pp. 423–453, 2018.

[9] K. Yang and J. Deng, "Learning to prove theorems via interacting with proof assistants," in *International Conference on Machine Learning*. PMLR, 2019, pp. 6984–6994.

[10] M. Harman, "The current state and future of search based software engineering," in *Future of Software Engineering (FOSE'07)*. IEEE, 2007, pp. 342–357.

[11] E. First, Y. Brun, and A. Guha, "Tactok: Semantics-aware proof synthesis," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–31, 2020.

[12] A. Sanchez-Stern, Y. Alhessi, L. Saul, and S. Lerner, "Generating correctness proofs with neural networks," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2020, pp. 1–10.

[13] E. First and Y. Brun, "Diversity-driven automated formal verification," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 749–761.

[14] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[15] A. Sanchez-Stern, Y. Alhessi, L. Saul, and S. Lerner, "Generating correctness proofs with neural networks," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2020, pp. 1–10.

[16] P. Nie, K. Palmskog, J. J. Li, and M. Gligoric, "Deep generation of coq lemma names using elaborated terms," in *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part II 10*. Springer, 2020, pp. 97–118.

[17] T. Ringer, R. Porter, N. Yazdani, J. Leo, and D. Grossman, "Proof repair across type equivalences," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 112–127.

[18] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent convolutional neural networks for text classification," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 29, no. 1, 2015.

[19] T. He, W. Huang, Y. Qiao, and J. Yao, "Text-attentional convolutional neural network for scene text detection," *IEEE transactions on image processing*, vol. 25, no. 6, pp. 2529–2541, 2016.

[20] M. Jaderberg, K. Simonyan, A. Vedaldi, and A. Zisserman, "Reading text in the wild with convolutional neural networks," *International journal of computer vision*, vol. 116, pp. 1–20, 2016.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[22] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 440–450.

[23] Y. Yang, H. Lv, and N. Chen, "A survey on ensemble learning under the era of deep learning," *Artificial Intelligence Review*, vol. 56, no. 6, pp. 5545–5589, 2023.

[24] T. N. Rincy and R. Gupta, "Ensemble learning techniques and its efficiency in machine learning: A survey," in *2nd international conference on data, engineering and applications (IDEA)*. IEEE, 2020, pp. 1–6.

[25] M. A. Ganaie, M. Hu, A. K. Malik, M. Tanveer, and P. N. Suganthan, "Ensemble deep learning: A review," *Engineering Applications of Artificial Intelligence*, vol. 115, p. 105151, 2022.

[26] R. Ghanem and H. Erbay, "Spam detection on social networks using deep contextualized word representation," *Multimedia Tools and Applications*, vol. 82, no. 3, pp. 3697–3712, 2023.

[27] D. Huang, P. Dhariwal, D. Song, and I. Sutskever, "Gamepad: A learning environment for theorem proving," in *International Conference on Learning Representations*, 2018.

[28] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *International conference on logic for programming artificial intelligence and reasoning*. Springer, 2010, pp. 348–370.

[29] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss *et al.*, "Dependent types and multi-monadic effects in f," in *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 256–270.

[30] N. Vazou, *Liquid Haskell: Haskell as a theorem prover*. University of California, San Diego, 2016.

[31] A. Trybulec and H. A. Blair, "Computer assisted reasoning with mizar." in *IJCAI*, vol. 85. Citeseer, 1985, pp. 26–28.

[32] K. Slind and M. Norrish, "A brief overview of hol4," in *International Conference on Theorem Proving in Higher Order Logics*. Springer, 2008, pp. 28–32.

[33] J. Harrison, "Hol light: A tutorial introduction," in *International Conference on Formal Methods in Computer-Aided Design*. Springer, 1996, pp. 265–269.

[34] P. B. Andrews and C. E. Brown, "Tps: A hybrid automatic-interactive system for developing proofs," *Journal of Applied Logic*, vol. 4, no. 4, pp. 367–395, 2006.

[35] J. C. Blanchette, L. Bulwahn, and T. Nipkow, "Automatic proof and disproof in isabelle/hol," in *Frontiers of Combining Systems: 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings 8*. Springer, 2011, pp. 12–27.

[36] A. Bundy, "A science of reasoning," in *International conference on automated reasoning with analytic tableaux and related methods*. Springer, 1998, pp. 10–17.

[37] T. Gauthier, C. Kaliszyk, J. Urban, R. Kumar, and M. Norrish, "Tactictoe: learning to prove with tactics," *Journal of Automated Reasoning*, vol. 65, no. 2, pp. 257–286, 2021.

[38] M. Wu, M. Norrish, C. Walder, and A. Dezfouli, "Tacticzero: Learning to prove theorems from scratch with deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 9330–9342, 2021.

[39] T. Ringer, N. Yazdani, J. Leo, and D. Grossman, "Adapting proof automation to adapt proofs," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2018, pp. 115–129.

[40] A. Celik, K. Palmskog, and M. Gligoric, "icoq: Regression proof selection for large-scale verification projects," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 171–182.

[41] L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce, "Generating good generators for inductive relations," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–30, 2017.

[42] J. Heras and E. Komendantskaya, "Recycling proof patterns in coq: Case studies," *Mathematics in Computer Science*, vol. 8, no. 1, pp. 99–116, 2014.

[43] E. Komendantskaya, J. Heras, and G. Grov, "Machine learning in proof general: interfacing interfaces," *Electronic Proceedings in Theoretical Computer Science*, vol. 118, pp. 15–41, 2013.

[44] Z. Li, B. Chen, and X. Si, "Graph contrastive pre-training for effective theorem reasoning," *In International Conference on Machine Learning (ICML)*, 2021.

[45] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2015, pp. 1556–1566.

[46] G. Cunningham, R. C. Bunescu, and D. Juedes, "Towards autoformalization of mathematics and code correctness: Experiments with elementary proofs," *arXiv preprint arXiv:2301.02195*, 2023.

[47] A. Sanchez-Stern, E. First, T. Zhou, Z. Kaufman, Y. Brun, and T. Ringer, "Passport: Improving automated formal verification using identifiers," *ACM Transactions on Programming Languages and Systems*, vol. 45, no. 2, pp. 1–30, 2023.

[48] E. First, M. N. Rabe, T. Ringer, and Y. Brun, "Baldur: Whole-proof generation and repair with large language models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1229–1241.

[49] A. Thakur, Y. Wen, and S. Chaudhuri, "A language-agent approach to formal theorem-proving," *arXiv preprint arXiv:2310.04353*, 2023.

[50] A. Q. Jiang, S. Welleck, J. P. Zhou, W. Li, J. Liu, M. Jamnik, T. Lacroix, Y. Wu, and G. Lample, "Draft, sketch, and prove: Guiding formal theorem provers with informal proofs," *arXiv preprint arXiv:2210.12283*, 2022.

[51] J. M. Han, J. Rute, Y. Wu, E. W. Ayers, and S. Polu, "Proof artifact co-training for theorem proving with language models," *arXiv preprint arXiv:2102.06203*, 2021.

[52] C. Zheng, H. Wang, E. Xie, Z. Liu, J. Sun, H. Xin, J. Shen, Z. Li, and Y. Li, "Lyra: Orchestrating dual correction in automated theorem proving," *arXiv preprint arXiv:2309.15806*, 2023.

[53] K. Yang, A. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. J. Prenger, and A. Anandkumar, "Leandojo: Theorem proving with retrieval-augmented language models," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[54] J. P. Zhou, C. Staats, W. Li, C. Szegedy, K. Q. Weinberger, and Y. Wu, "Don't trust: Verify–grounding llm quantitative reasoning with autoformalization," *arXiv preprint arXiv:2403.18120*, 2024.

[55] C. Wen, J. Cao, J. Su, Z. Xu, S. Qin, M. He, H. Li, S.-C. Cheung, and C. Tian, "Enchanting program specification synthesis by large language models using static analysis and program verification," *arXiv preprint arXiv:2404.00762*, 2024.