Natural Formalized Proof Language

Lihan Xie, Zhicheng Hui, and Qinxiang Cao( )

Shanghai Jiao Tong University, Shanghai, China
{sheringham laplace_demon}@sjtu.edu.cn caoqinxiang@gmail.com

Abstract Artificial intelligence assisted mathematical proof has become a highly focused area nowadays. One key problem in this field is to generate formal mathematical proofs from natural language proofs. Due to historical reasons, the formal proof languages adopted by traditional theorem provers were not intended to represent natural language proofs. Therefore, they are not well-suited for the aforementioned tasks and proof-checking work for educational purposes. In this paper, we design a proof language and its corresponding abstract syntax tree and implement a proof checking tool for it. This language can be easily converted from natural language, thus providing a rich corpus of formal proof. Additionally, it supports the handling of issues in informal proofs through static analysis, and enhances the expressive power of the language by introducing the structure of partial proofs. This design combines the expressiveness of natural language and the accuracy of formal language, resulting in an improved mathematical proof language.

Keywords: Formal proof language · Theorem proving · Static analysis

Introduction

Formal mathematical proofs are based on rigorous reasoning in formal logic, providing a completely accurate proof process that can be automatically verified by computers. Formalizing informal proofs can make them more convincing, as seen in the formal proof of the Four-Color Theorem [1,11]. Additionally, automated theorem proving relies on formal proof languages to generate rigorous proofs. The rise of large language models in recent years has spurred research in AI-assisted mathematical proof, particularly in the fields of automated theorem proving and transforming informal proofs into formal proofs [14,20]. This means that large language models can directly generate formal proofs or indirectly transform natural language proofs into formal proofs. These formal proofs can then be verified by theorem provers, ensuring their correctness.

Due to historical reasons, early versions of theorem provers were primarily focused on ensuring the correctness of proofs, rather than directly modelling natural language proofs. Subsequent developments, such as the Ltac [9] tactic language in Coq [2] and Isar [19] proof language in Isabelle [16], aimed to

L. Xie and Z. Hui—contributed equally to this work.

W.-N. Chin and Z. Xu (Eds.): TASE 2024, LNCS 14777, pp. 446–464, 2024.
https://doi.org/10.1007/978-3-031-64626-3_26

help with proof constructions. Commands in these languages can be seen as transformations that modify the current proof goal, where each proof goal comprises named premises and the conclusion that needs to be proved. However, it is still di cult to translate natural language proofs word-for-word into such formal proof languages. Consequently, for current applications like AI-assisted mathematical proofs or automated proof grading for educational purposes, these formal languages seem to be insu cient.

In the following example shown in Fig. 1 about the proof of the monotone convergence theorem through the supremum theorem, we can observe several characteristics of natural language proofs and the di culties consequent upon the task of formalization using existing formal proof languages like Coq. We will also demonstrate how our work circumvents these di culties.

Monotone Convergence Theorem: For every sequence of real numbers $(a_n)_{n\in\mathbb{N}}$, $(a_n)_{n\in\mathbb{N}}$ converges if it is monotonically increasing and bounded above.

Proof. (1) Assume $(a_n)_{n\in\mathbb{N}}$ is monotonically increasing and bounded above.
(2) By supremum theorem, there exists A such that $A = \sup\{a_n\}$.
(3) We use the definition of limit to show that $\lim_{n\to+\infty} a_n = A$.
(4) For every $\varepsilon > 0$, by the definition of least upper bound, there exists an integer N such that $a_N > A - \varepsilon$.
(5) Since $\{a_n\}$ increases, for every n, $n > N$ implies $a_n > a_N$.
(6) Since A is an upper bound of $(a_n)_{n\in\mathbb{N}}$, for every n, $n > N$ implies $a_n < A$.
(7) Consequently, for every n, $n > N$ implies $A - \varepsilon < a_n < A + \varepsilon$.
(8) By the definition of convergence, we have $\lim_{n\to+\infty} a_n = A$,
(9) which proves the theorem. □

Fig 1 Proof of monotone convergence theorem.

Partial Transformation of Proof Goal. The concept of proof goal models the task of a mathematical proof, i.e. deriving the conclusion from premises and proven results. Following the steps of a proof, we implicitly transform the proof goal by introducing new variables, proving intermediate results, or posing subgoals. For example, we pose a subgoal in line 3, thereby transforming the proof goal into two subsequent proof goals: (1) proving $\lim_{n\to\infty} a_n = A$, (2) and proving the conclusion with the intermediate result $\lim_{n\to\infty} a_n = A$ proven. However, there are circumstances where the subsequent proof goal cannot be found directly: We pose an assumption $\quad > 0$ in line 4 and derive the result $n > N, A - \ < a_n < A + \$ in line 7 under the assumption. The overall process proves the result $\quad > 0, \exists N \in \quad , \ n > N, A - \ < a_n < A + \$, which is not explicitly stated in line 4. Consequently, the transformation of the proof goal in line 4 is partial. Since each tactic in tactic languages should represent a clearly defined, complete transformation of the current proof goal, formalizing

the proof using tactic languages would involve the extra task of determining the subsequent proof goal. For example, by inferring the proposition to be filled in a Coq "assert" tactic. *In our work, a structure of partial proof is included in the design of proof language to model this pattern of proof, it eliminates the need for additional inferences or structural modifications on the proof during the task of formalization.*

Context dependent Semantics. Depending on the context, the same natural language statement could have multiple interpretations, of which the semantic differences may be subtle. For example, when we write "there exists" in the proof, at least three interpretations are possible. (1) In the context of proving an existential statement, a satisfying value has been found for one of the existential quantifiers in the conclusion to be proved, (2) or a proposition beginning with an existential quantifier is stated, (3) or similar to the second case, except that the qualified variable becomes a free variable and can be used later, as in line 2 and line 4 of the example, where variables A and N are used in line 3 and line 5, respectively. In order to use the tactic language of Coq, all those semantic variances must be explicitly formulated. Namely by an "exists" tactic for the case (1), an existentially quantified variable in the proposition for the case (2) and a free variable in the proposition for the case (3). Determining the correct semantic interpretation would require analysing the context during the process of formalization. Not only is such an hidden task of analysis generally harder to perform on an unstructured natural language proof, but it is also indirect on a tactic proof, as tactics do not explicitly contain information on proof goals. That explains the di culties faced by the tactic languages as object languages of automatic formalization. *In our work, the proof language is designed to resemble natural language in order to streamline the formalization. Moreover, the resemblance allows the proof language to temporarily preserve the context-dependent semantics, thus allowing the resolution to be postponed until we can perform static analysis on the formalized version of the proof.*

Overloading of Notation. Sometimes a notational convention may be employed though not being mathematically rigorous. For example, the appearance of $\{a_n\}$ in line 2 and line 5 does not represent a singleton but rather the set containing all elements of the sequence a. Another typical example is the extensive usage of $f\ x$) for representing the function f itself. Similar to context-dependent semantics, all mathematical formulas should be written rigorously in Coq. *By performing static analysis on the entire formal proof, the precise meaning of each expression can also be inferred.*

In summary, to address the problems mentioned above, we design a natural-language-like formal proof language for modeling mathematical proofs. We add partial proof structures to make them more similar to the natural language proofs. Furthermore, after transforming the natural formal proof into an abstract syntax tree by a parser, we can perform static analysis on it to resolve context-dependent semantics and overloading of notation.

Accordingly, we implement a framework called ProofGrader for checking mathematical proofs automatically. For each step of reasoning, corresponding solvers are chosen heuristically in order to check its correctness. Apart from the proof checking kernel, the rest of the system is designed to be highly modular so that our system may fit into different usage scenarios: It is possible to alter the mathematical objects and the usable theorems involved in the proof, the acceptable forms of proof steps, or the solvers used for checking. For example, to serve educational purposes, mild solvers fitting the level of human intuition can be plugged in instead of powerful ones, and advanced theorems can be temporarily disabled until they are proved or taught later.

In the rest of this paper, we will first present the design of our proof language in Sect. 2, with elaborations on several important elements. After that, we give the formal semantics of our proof language in Sect. 3. We then show the schema of the workflow of our checker in Sect. 4. A detailed description of our solvers is developed in Sect. 5. Section 6 gives an evaluation of the proof checker. And then we will introduce some related works in Sect. 7. Finally, Sect. 8 is devoted to a conclusion.

2 Proof Language Design

We focus on defining a natural-language-like proof language, whose structure faithfully reflects that of natural language. Thus, our natural formal proof language is provided with the hierarchy of natural language proof. At the top level are proof steps, what follows are propositions and terms. A certain amount of investigations of natural language proofs are done to incorporate common proof patterns within our proof language. Such a proof language follows predefined grammar rules but can be read directly as natural language, an explanation of its grammar can be found in the long version of this paper [21].

To perform proof checking, the natural formal proof language is further translated into an abstract syntax tree via a parser. Below shows a subset of its definitions. The definitions related to `term` and `prop` can be found in the long version of this paper [21].

⟨*proof*⟩ ::= '`Proof ction`' ⟨*action*⟩ ⟨*proof*⟩
| '`PoseWithoutProof`' ⟨*fwd*⟩ ⟨*prop*⟩ ⟨*proof*⟩
| '`Pose ndProve`' ⟨*fwd*⟩ ⟨*prop*⟩ ⟨*proof*⟩ ⟨*proof*⟩
| '`ClaimSuffice`' ⟨*bwd*⟩ ⟨*prop*⟩ ⟨*proof*⟩
| '`ProveSuffice`' ⟨*bwd*⟩ ⟨*prop*⟩ ⟨*proof*⟩ ⟨*proof*⟩
| '`ConclWithoutProof`' ⟨*fwd*⟩
| '`Concl ndProve`' ⟨*fwd*⟩ ⟨*proof*⟩
| '`PosePartialProof`' ⟨*poseAction*⟩ ⟨*proof*⟩ ⟨*proof*⟩
| '`EndPartialProof`'

⟨*fwd*⟩ ::= '`FNoHint`'
| '`FDefinition`' ⟨*definition*⟩
| '`FTheorem`' ⟨*theorem*⟩
| '`F ddEqn`' {⟨*identifier*⟩}*
| '`FDeriBothTerms`' ⟨*identifier*⟩
| ...

⟨*bwd*⟩ ::= 'BNoHint'
| 'BContra'
| ...

⟨*action*⟩ ::= ' Intros' ⟨*identifier*⟩
| ' Exists' ⟨*term*⟩
| ' Suppose' ⟨*prop*⟩
| ' Set' ⟨*identifier*⟩ ⟨*term*⟩
| ' SetProp' ⟨*prop*⟩
| ' ExistVar' ⟨*identifier*⟩

⟨*poseAction*⟩ ::= ' PoseVar' ⟨*identifier*⟩ {*prop*}*
| ' PoseProp' ⟨*prop*⟩

In the following, we will explain the meaning of each proof structure. We will refer to the names of abstract syntax tree nodes because these names correspond to components of the proof language. Some of our proof structures have similar functionalities to those of certain tactic language, while others have no counterpart, which enables us to better express natural language proof.

ProofAction. The field `action` constitutes an operation on the proof goal, and the field `proof` refers to subsequent proof steps.

The design of `Proof ction` refers to some tactic languages, such as the tactic *intro* and *exists* of Coq [4], which deal with the quantifiers in the proposition to be proved. That corresponds to our proof action `Intros` and `Exists`. However, the variety of proof actions is richer than that of tactic language. For example, `Suppose` introduces a premise in the conclusion rather than a variable, which is also expressed by the tactic *intro* in Coq. Such a difference is due to the fact that the proof language models the natural language directly, and thus is more in line with human intuition.

`Set` binds a name to a term, to which the proof can refer later, and the existence of the term will be verified by the checker to ensure mathematical rigor.

`SetProp` is a generalization of `Set`. Instead of introducing a new variable through an equation, `SetProp` introduces new variables through a proposition. Figure 2 below contrasts the usage of these two proof actions, the one-to-one correspondence between natural language and our proof language is marked in background color.

Finally, `ExistVar` indicates the action of instantiating the variable mentioned by an existential quantifier in the last premise in the proof goal. The usage of `ExistVar` is further demonstrated in Sect. 4.3 when we perform static analysis.

In the upper part of Fig. 2, the variable A is set equal to the limit of a sequence, the checker then verifies whether the limit exists. In the lower part, a variable k is introduced implicitly due to the introduction of a subsequence, the checker then verifies whether $a_n)_{n\in}$ admits a convergent subsequence. Both of the two proof steps will add corresponding assumptions to the proof goal.

PoseWithoutProof & PoseAndProve. These two components correspond to forward reasoning. The field `fwd` indicates the method involved in forward reasoning. The field `prop` denotes the proposition that the step proves. The last field `proof` still refers to subsequent proof steps.

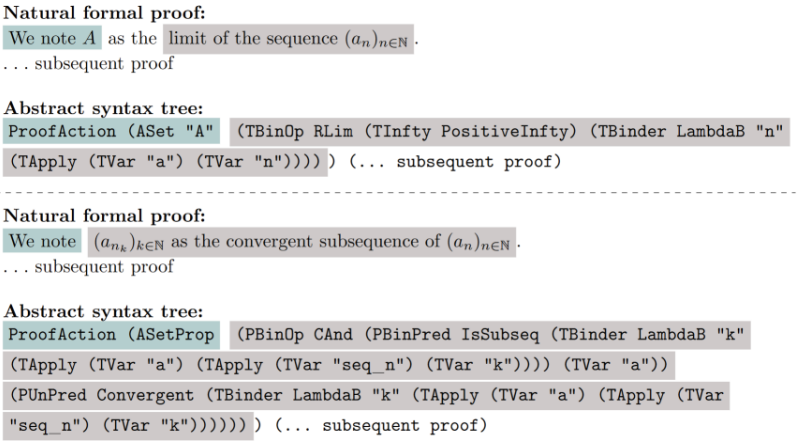


Fig 2 Example of ProofAction.

Depending on the complexity of the reasoning, one may choose to provide a proof as a justification of the reasoning or just let the checker figure it out. This makes the difference of `PoseWithoutProof` and `Pose ndProve`. `Pose ndProve` carries an extra field `prop`, which is a complete proof showing how to derive its result. This design is also widespread in tactic languages. An example is the tactic *assert* in Coq, which poses a subgoal to be proved. Figure 3 shows its usage scenario, which corresponds to the line 3 of Fig. 1.

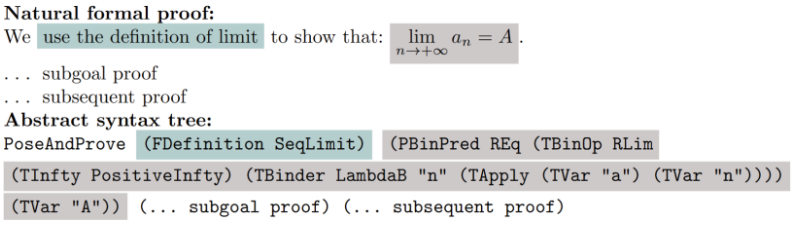


Fig 3 Example of PoseAndProve.

Since the hint of using the definition of limit is far from su cient to prove the result, an additional proof is provided to complete this task. The proven subgoal is then available in the subsequent proof.

The set of possible methods `fwd` is rather rich. To name a few, `FTheorem` denotes applying a theorem, `F ddEqn` denotes adding several equations together to get a new one, and `FDeriBothTerms` denotes taking a derivative from both sides of an equation. If no method is indicated, `FNoHint` is filled in. It is quite easy to incorporate new methods so this set is highly expandable.

ClaimSu ce & ProveSu ce. These two components correspond to backward reasoning. The first field `bwd` indicates the method involved in backward reasoning, the second field `prop` denotes the proposition the step proves and the last field `proof` refers to subsequent proofs. The distinction between `ClaimSuffice` and `ProveSuffice` follows that between `PoseWithoutProof` and `Pose ndProve`.

Backward reasoning signifies that they start from the conclusion to be proved. The result provided is supposed to imply goal, which will then become the new goal.

ConclWithoutProof & ConclAndProve. These two components correspond to the step of deriving the conclusion and terminating the proof, only a field `fwd` is presented to indicate the method involved. `Concl ndProve` allows the choice of presenting an extra explanatory proof when the derivation of the conclusion is not immediate.

Basically, `ConclWithoutProof` corresponds to the last step of the proof, such as the line 9 "which proves the theorem" in Fig. 1, and does not carry much information. But such a concluding remark imitates natural language proof, and the similarity to natural language characterizes our proof language.

PosePartialProof & EndPartialProof. In theorem provers, we always need to explicitly keep a record of the current proof goal. In most cases, this is the overall proposition to be proved. We can also pose a subgoal and prove it subsequently, a process represented by the `Pose ndProve` component in our proof language.

As discussed in Sect. 1, natural language proofs may also involve partial transformations of the proof goal, in the sense that subgoals are not explicitly stated in advance. Instead, We simply pose certain assumptions and proceed to derive the subgoals. Refer to Fig. 4 for an illustration of how the different forms of proof goal are in the example Fig. 1. The dark gray part of the proof denotes what we term as a "partial proof", aligning with the `PosePartialProof` and `EndPartialProof` components in our proof language.

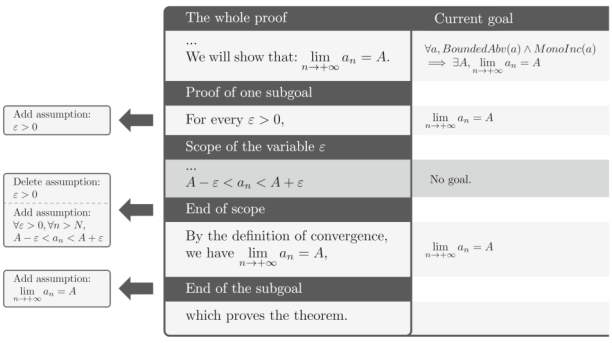


Fig 4 Illustration of how the proof goal changes.

The notion of partial proof can be considered as one innovation of our proof language. Compared with tactic languages, it offers us extra convenience in modelling natural language proofs.

A partial proof starts with a list of variables and propositions, which then become the temporary assumptions. This information is carried by the first field `pose ction` is the grammar. `PoseVar` poses a new temporary variable along with related assumptions, while `PoseProp` poses a proposition. The former field `proof` carries the partial proof, which shall be terminated by `EndPartialProof`. After the termination of partial proof, all proven propositions will be altered to its proper form - without free variables. These altered propositions will become henceforth available in the subsequent proof - the latter field `proof`.

In Fig. 4, the variable ε and the proposition $\varepsilon > 0$ are posed to start a partial proof. When reaching the result $n > N, A - \varepsilon < a_n < A + \varepsilon$ at the end of the partial proof, it is expanded to $\varepsilon > 0,\ n > N, A - \varepsilon < a_n < A + \varepsilon$, which is subsequently used to prove convergence. Figure 5 shows the how a partial proof looks like, which corresponds to line 4 of Fig. 1.

Natural formal proof:
For every $\varepsilon > 0$,
. . . partial proof in the scope of ε
. . . subsequent proof
Abstract syntax tree:

```
PosePartialProof (APoseVar "epsilon" ((PBinPred RGt (TVar "epsilon")
(TNum 0)) :: nil)) (... partial proof in the scope of ε ... EndPartialProof)
(... subsequent proof)
```

Fig 5 Example of PosePartialProof.

3 The Formal Semantics of Proof Language

In this section, we demonstrate a selected set of the formal semantics of our proof language. Some semantics of the proof language not covered here can be found in the extended version of the paper [21]. Since each proof step defines a transformation of the proof goal, we use the notation $pr, pg) \rightarrow pr', pg')$ to denote that the proof goal pg' results from the proof goal pg after one or more proof steps in pr, with pr' continuing the proof. A proof goal is defined as a pair $A, C)$, where A is a list of premises and C a conclusion that needs to be proved. In the case of partial proof, the conclusion C does not exist and we note it as .

For convenience, we represent a list of premises as a set of propositions, and we write the triple $pr, A, C)$ to represent the pair $pr,\ A, C))$. We use the notation $FV\ A)$ to denote the set of free variables in the set of propositions A, the same notation $FV\ t)$ is also used to denote the set of free variables in the term t. We also define a constant QED such that the proof is successfully completed when $pr, A, C) = QED$.

Proof Action.

$$\frac{v \notin FV\ \ A \cup \{\ \ x.C\})}{\textbf{ProofAction}\ \ \textbf{AIntros}\ v)\ pr, A,\ \ x.C) \rightarrow\ \ pr, A, C[v/x])}\ \textsc{ntros}$$

Proof actions are proof steps that directly manipulates the proof goal, usually when some conditions are met. The behavior of **AIntros** v is to remove the universal quantifier in the conclusion $x.C$ and replace the variable x in C by the variable v, when the variable v does not occur freely in the proof goal. The behavior of other **ProofAction** statements is similar. Detailed semantics and descriptions are available in the long version of the paper [21].

Partial Proof.

$$\frac{\begin{array}{c} pr, A \cup sP,\ \ ) \rightarrow\ \textbf{EndPartialProof}, A',\ \ ) \\ FV\ \ sP) \subset FV\ \ A \cup \{C\}) \cup \{s\} \end{array}}{\begin{array}{l} \textbf{PosePartialProof}\ \ \textbf{APoseVar}\ s\ sP)\ pr\ pr', A, C) \\ \rightarrow\ \ pr', A \cup\ \text{AddVarDep}\ \ A' \setminus\ \ A \cup lP))\ s\ sP), C) \end{array}}\ \textsc{PoseVar}$$

$$\frac{\begin{array}{c} pr, A \cup \{P\},\ \ ) \rightarrow\ \textbf{EndPartialProof}, A',\ \ ) \\ FV\ \ P) \subset FV\ \ A \cup \{C\}) \end{array}}{\begin{array}{l} \textbf{PosePartialProof}\ \ \textbf{APosePropP})\ pr\ pr', A, C) \\ \rightarrow\ \ pr', A \cup\ \text{AddPropDep}\ \ A' \setminus\ \ A \cup \{P\}))\ P), C) \end{array}}\ \textsc{PoseProp}$$

A partial proof first makes one or more assumptions and then deduces a series of results. After the partial proof terminates, these results are added with dependencies on the assumptions, by prefixing them with universal quantifiers and prerequisite conditions. The assumptions can be either (1) posing a new variable satisfying certain conditions, (2) or posing a hypothesis on the existing variables of the proof goal. These two cases correspond to the constructs **APoseVar** $s\ sP$ and **APoseProp** P defined above, where s refers to the name of the posed variable, sP a set of assumptions on the posed variable and P an assumption on existing variables. The two operators AddVarDep and AddPropDep add dependencies on the assumptions to the results derived during the partial proof. Similar to the subgoal proof, a **PosePartialProof** statement causes the checker to first check the partial proof. After reaching **EndPartialProof**, which marks the end of the partial proof, the checker integrates the proof goal back to the main proof by modifying all the derived premises.

The Table 1 shows how the proof goal gets transformed between lines 4–7 in Fig. 1, which represents a partial proof structure. It first makes an assumption > 0 and then deduces several results. After that, these results are added with a prefix > 0 to indicate their dependency on the assumption > 0.

Table 1 Transformation of proof goal between lines 4–7 in Fig. 1. The upper part of the table shows the proof goal before reaching line 4 of the proof. The middle part of the table shows the deduction of intermediate results during the partial proof. The lower part of the table shows the proof goal after terminating the partial proof in line 7 of the proof.

Premises	**Conclusion**
$\{a_n\}$ has an upper bound $\{a_n\}$ is monotonically increasing there exists A such that $A = \sup\{a_n\}$ $A = \sup\{a_n\}$	$A = \lim_{n \to \infty} a_n$
+ $\quad > 0$ **partial proof start)** + there exists N such that $a \quad > A -$ + for all n, if $n > N$ then $a_n > a$ + for all n, if $n > N$ then $a_n < A$ + for all n, if $n > N$ then $A - \quad < a_n < A$ **partial proof end)**	
- Remove all the premises added above + for all $\quad > 0$, there exists N such that $a \quad > A -$ + for all $\quad > 0$, there exists N such that for all n, if $n > N$ then $a_n > a$ + for all n, there exists N such that if $n > N$ then $a_n < A$ + for all n, there exists N such that if $n > N$ then $A - \quad < a_n < A$	$A = \lim_{n \to \infty} a_n$

4 The Workflow of ProofGrader

4.1 Overall Architecture

In this subsection, we will primarily introduce the overall working framework of ProofGrader. Figure 6 illustrates how the natural formal proof is processed step by step in our proof checking system to obtain the final result. A larger figure can be found in the long version of this paper [21] for greater clarity.

The modules described in the squares of the diagram are the main components of ProofGrader, which include the parser, static analyser, and proof checker. These three parts will be discussed in detail in the following. The other process boxes represent different forms of mathematical proof during the whole workflow. We utilize a parser to convert the natural formalized proof into an abstract syntax tree, and then perform static analysis on the abstract syntax tree to eliminate ambiguities and add omitted steps. The checker takes in the proof goal generated by the static analyser along with the complete abstract syntax tree, and checks the proof step by step by predefined rules. Finally, it confirms whether the proof goal has been proven and generates the final result.

4.2 Proof Parsing

Proof parsing is the first step of the entire workflow of ProofGrader. Since the design of our proof language also takes a large part into account readability factors, it may not be the most convenient for the subsequent checking process. Therefore, we will first convert the natural formal proof into an abstract syntax tree described in Sect. 2. In our implementation, the lexer and parser are realized by flex and bison [10,15].

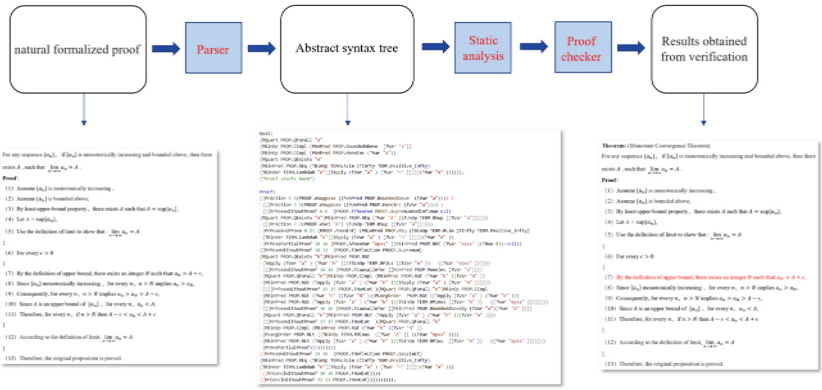


Fig 6 Overall workflow of ProofGrader.

The parser simply performs a plain translation based on the grammar rules, without making any further modification on the proof structure or the formulation of proposition. These are the subjects of the next section where we will discuss the static analysis on the proof.

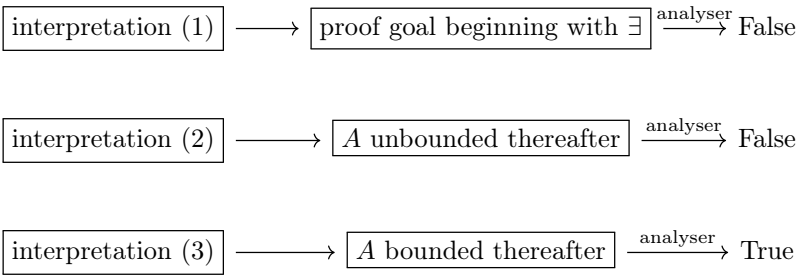
4.3 Static Analysis

As discussed in Sect. 1, natural language proofs exhibit complexities such as context-dependent semantics and overloading of notation. Given the close resemblance of our proof language to natural language and the direct translation performed by the parser, these properties will be carried into the natural formal proof, and then the abstract syntax tree. Performing static analysis on the abstract syntax tree allows us to eliminate these problems by reorganizing the proof into a more rigorous form, ready to be checked by the proof checker.

The static analyser tackles each kind of problem separately with ad hoc method. For one problematic proof step, proposition or expression, the task is to choose the right semantic between all the possible interpretations. The static analyser works by first inferring how the current proof step transforms the proof goal, based on the previous and subsequent proof goals. It then selects the semantic corresponding to this transformation, by elaborating the proof step, proposition or expression into a proper form. Judging from the problems we are currently solving by static analysis, most of them are accompanied by the appearance of free variables. So it is often the case to perform a lexical scope analysis.

Handling Context-dependent Semantics. The context-dependent semantics we currently address are stated in the descriptions of Fig. 1. In this example, the static analyser infers the correct semantic from the context, as shown in the inference steps below. As a reminder, the three possible interpretations of "there exists A such that $A = \sup\{a_n\}$" are respectively: (1) a value $\sup\{a_n\}$ has

been found for an existential quantifier in the conclusion to be proved, (2) an intermediate result $\exists A = \sup\{a_n\}$ is stated, where A is qualified by a quantifier, (3) a variable A is given the value $\sup\{a_n\}$ after proving the existence of this supremum.



The three potential semantic interpretations appear syntactically identical in the proof. Therefore, the static analyser takes the responsibility for reflecting the result of the above analysis on the abstract syntax tree through modification. This is where the proof action `ExistVar` comes into play. Placed immediately after a proposition starting with an existential quantifier, it indicates the instantiation of the variable mentioned by the quantifier, thus bringing about the semantics corresponding to case (3), namely binding variable A of the value.

Algorithm 1 illustrates the procedure of handling context-dependent semantics in the cases of **PoseWithoutProof** and **PoseAndProve**, the processing of other cases follows a similar approach.

Algorithm 1 HCDS: Handling Context-dependent Semantics

Input: the original proof Pr before static analysis
Output: the modi ed proof with ontext-dependent Semantics eliminated
When Pr = PoseWithoutProof fwd $P0$ $Pr0$
$P0$ = PQuant exists "x" $P1$
"x" occur freely in $Pr0$
Then HCDS(Pr) = PoseWithoutProof fwd $P0$ (ProofAction (AExistVar "x") HCDS($Pr0$))

When Pr = PoseAndProve fwd $P0$ $Pr0$ $Pr1$
$P0$ = PQuant exists "x" $P1$
"x" occur freely in $Pr1$
Then HCDS(Pr) = PoseAndProve fwd $P0$ (HCDS($Pr0$) (ProofAction (AExistVar "x") HCDS($Pr1$))
...

Handling Overloading of Notation. If there is only one possible interpretation for notation overloading, there is no need for analysis and the static analyser simply restores its rigorous form. So far, the cases we have encountered with multiple possible interpretations for notation overloading all involve the use of formal variables. In such cases, the static analyser examines the appearance of free variables. As an example, the $\{a_n\}$ in line 2 of Fig. 1 admits two possible interpretations: (1) the singleton $\{a_n\}$, (2) or a set containing all elements of the sequence a. The correct interpretation depends on whether the variable n is bound to a value in the current proof goal. Similarly, if x is identified as a

free variable in the proof goal, $f\ x$) will be transformed into either f or $\lambda x.f\ x$), rather than the value of x applied to the function f.

Algorithm 2 illustrates the procedure for handling the notation overloading of $f\ x$) in the cases of **PoseWithoutProof** and **ProofAction ASet** $x\ t$), the processing of other cases follows a similar approach.

Algorithm 2 HON: Handling Overloading of Notation

Input: the original proof pr before static analysis
list of binded variables $binded\_varlist$ from the beginning to the current point of the proof
Output: the modi ed proof with overloading of notation eliminated
When Pr = PoseWithoutProof fwd $P0$ $Pr0$
$P0$ = Eq (TApply ("f") ("x")) $t1$
if "x" is not in $binded\_varlist$
Then HON(Pr, $binded\_varlist$) = PoseWithoutProof fwd (Eq ("f") (TBinder Lambda "x" $t1$))
HON($Pr0$, $binded\_varlist$)

When pr = ProofAction (ASet "x" t) $Pr0$
Then HON(Pr, $binded\_varlist$) = ProofAction (ASet "x" t) HON($Pr0$, "x" :: $binded\_varlist$)
...

4.4 Proof Checking

Proof checking is the final step of the entire workflow of ProofGrader. The checker takes the proof and the proof goal elaborated by the static analyser as input. For each proof step, the checker takes the current proof goal and checks the step according to the formal semantics presented in Sect. 3, it computes the subsequent proof goal along with a boolean value indicating whether the step is accepted. By iteratively repeating this process, the proof checker finally generates a list of boolean values indicating the correctness of each proof step. In order to help with proposition checking, we also develop several solvers and a solver manager system. We will introduce them in Sect. 5. The entire proof checking process is detailed in Fig. 7.

5 Solver Manager

The solver manager is an important component of the proof checker, and therefore, we will introduce it separately here. It is primarily used to verify the correctness of mathematical propositions under certain conditions. When it comes to proving and simplifying mathematical expressions, we combine different solvers to check if the current proposition can be derived from known results using specific rules.

Each solver corresponds to a deduction rule based on specific mathematical concepts, and implements the corresponding algorithm internally in the proof checker. Different solvers are responsible for handling different types of mathematical expressions. For example, when dealing with expressions involving algebraic operators, ProofGrader utilizes algebra-related solvers for simplification. Similarly, solvers for trigonometric functions, exponential and logarithm, sequential limit, and other mathematical concepts are employed to handle corresponding cases.

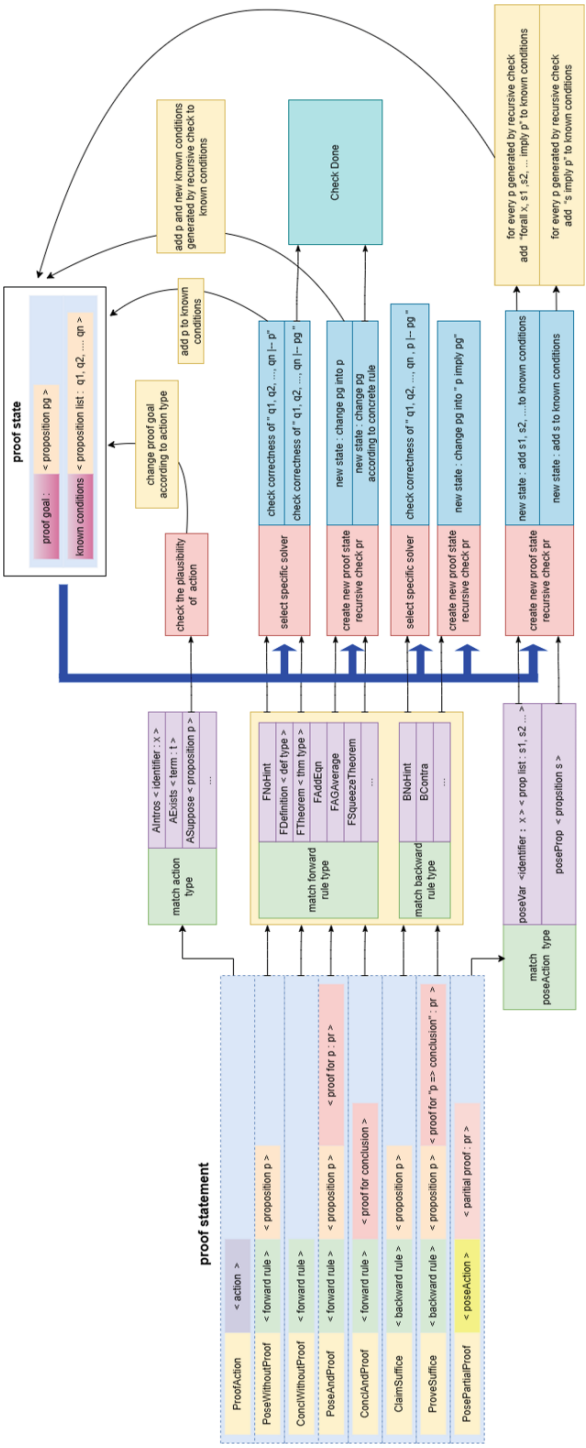


Fig 7 Process of proof checking.

When users write proofs, they must adhere to the steps supported by the solvers. In Sect. 1, it was noted that AI-generated formal proofs sometimes use solvers like Z3 [8] to check propositions. However, Z3 possesses strong reasoning capabilities and can sometimes prove the correctness of conclusions even if the original deduction process contains error. On the other hand, our solver does not engage in excessively powerful reasoning. It only supports relatively obvious deduction methods in proofs and does not allow excessive omission of steps. This approach aims to reflect the correctness of the original proof more accurately. Additionally, since we can provide explicit deduction rules supported by each solver, users can clearly know which steps can be omitted and which steps cannot be omitted in their proof process.

Before presenting the design of the solver manager, it is necessary to first define the structure of the solver, as these two are closely related. Each solver consists of the following four components:

- *solve(s,p)* : the function used to verify the correctness of a proposition takes two parameters: the current proof goal and the proposition itself. It can either return false when the proposition does not pass the solver, or a list of propositions, indicating that after being processed by the current solver, the correctness of the original proposition can be determined by individually checking the propositions in the list. This design allows us to combine multiple solvers to simplify a proposition. The list can contain only one proposition, which means the solver simplifies the original proposition and passes it to the next solver. An empty list indicates that the current proposition is accepted by the solver.
- *fee* : the cost of this solver. While we want to combine multiple solvers, we cannot endlessly repeat using various solvers, as it may result in non-termination. Therefore, we need to set an upper limit on the total cost within the solver manager and define the cost of each solver.
- *default-priority* : the default priority of solvers. When dealing with mathematical propositions that do not exhibit any prominent characteristics, solvers can be selected based on their default priority levels.
- *priority(p)* : the function used to compute the dynamic priority. Given the current proposition as input, this function returns the dynamic priority of the solver. For example, when handling a mathematical proposition without limits, solvers related to sequential limit may have a low priority or even be unavailable. However, when dealing with limit-related propositions, these solvers would have a high priority. Therefore, it is necessary to dynamically adjust the priority of solvers based on the specific proposition. This prepares us for the subsequent development of the solver manager.

Based on the structure of the solver, we implemente a solver manager capable of dynamically scheduling various solvers by combining dynamic priorities. The specific algorithms and design can be found in the long version of this paper [21].

6 Evaluation

In this section, we give an evaluation of our proof checking system as well as our proof language. We first run our system on a set of sample mathematical proofs, covering the topic of arithmetic, trigonometric functions, exponential and logarithm, inequality, derivative, sequential limit and function continuity. The source code of our system and the dataset are available at: https://github.com/Laplace-Demon/ProofGrader. We then compare the features of our proof language with other proof languages and proof checking tools.

6.1 Performance

We test our system on a dataset of 52 mathematical proofs, Table 2 shows the average time and memory overhead of the proof parser and the checker on each of the topics, along with the average file size of the proof. We can observe that our proof parser and checker perform their task in a reasonable amount of time.

Table 2 Runtime and memory usage of the parser and checker on di erent topics.

Examples			Parser		Checker	
Topic	Number	File size bytes)	Time ms)	Memory kb)	Time ms)	Memory kb)
arithmetic	6	271	34	3584	477	3456
trigonometric functions	8	599	53	3712	1409	3456
exponential and logarithm	3	371	40	3840	506	3456
inequality	10	395	58	3712	528	3456
derivative	3	385	75	3328	760	3328
sequential limit evaluation	10	550	58	3840	701	3328
sequential limit proof	10	1027	78	3968	670	3328
function continuity proof	2	923	68	3712	7758	3456

6.2 Comparison of Features

In the Table3 below, we compare the features of our proof language with some other proof languages and proof checking tools. The term "transformation chain" refers to the ability to handle and perform automated reasoning on a series of consecutive transformation derivations, such as:

$$\lim_{x\to 0}\frac{\sqrt[3]{x+1}-1}{\sqrt{x+1}-1}=\lim_{x\to 0}\frac{x}{\sqrt[3]{x+1})^2+\sqrt[3]{x+1}+1}\cdot\frac{\sqrt{x+1}+1}{x}=\frac{2}{3}$$

Our proof checker aims to provide automated deduction capabilities that align with human judgment, while excessive step omissions are not allowed.

In terms of readability, our natural formal proof can be read with the same effort of reading a natural language mathematical proof. Even the abstract syntax tree generated by the parser remains readable since the components bear a mnemonic name.

In terms of static analysis, most theorem provers like Coq and Isabelle primarily support propositional-level static analysis. They utilize type inference to fill in missing information in user-written proofs. However, they do not perform static analysis on the entire proof. Regarding partial proofs, while it is sometimes possible to reorganize a proof to avoid partial proofs, we aim to preserve the characteristics of natural language proofs. Users should be able to write mathematical proofs without additional effort.

The work of Waterproof [18] aims to help students understand mathematical proofs. Therefore, its language readability falls between natural language and theorem provers. The proof checking functionality of Waterproof relies on Coq implementation, so it inherits some limitations of Coq as well. Diproche [6] utilizes natural language fragments as its input language. Due to its lack of support for static analysis and partial proofs, it imposes stricter requirements on the types and structure of proofs being written. Lurch [7] is limited to checking proofs in propositional logic and naive set theory. It does not have automatic reasoning capabilities and relies on users to provide detailed proofs for checking.

Table 3 Comparison of features of di erent mathematical proof checking tools.

Feature	ProofGrader	WaterProof	Diproche	Lurch	Isabelle [16]	Coq [2]
Natural language fragment	✓	×	✓	✓	×	×
Static analysis	proof-level	×	×	×	prop-level	prop-level
Partial proof	✓	×	×	×	×	×
Transformation chains	✓	×	✓	×	✓ [3]	×

7 Related Work

Proof assistants and other proof checkers. Recent decades have seen the emergence of various proof languages. A well-known work is Ltac [9] for the theorem prover Coq, which provides convenience for constructing proofs and facilitates better proof automation. Another famous one is that of Isar [19] for the system Isabelle. One objective of Isar is to provide a more human-readable proof language than before. Despite their efforts to improve readability, the learning curve for using these tools remains high. Wemmenhove et al. developed an educational software called Waterproof [18] based on this to assist students in practicing proofs. Since they still choose to rely on the tactic library extended with the Ltac2 tactic language, Waterproof has not actually gained stronger expressive power than these tactic-based proof language. Additionally, the proof assistant Agda [5] also employs some proof notations that resemble natural language, such as using equation chain instead of the "rewrite" tactic used in Coq. However, despite these advancements, they still struggle to effectively support the structure of partial proofs. While they can perform static analysis at the propositional level, such as type inference, they are unable to perform static analysis on

the entire proof. In addition, tools mentioned in Sect. 6.2, such as Diproche and Lurch, although they support controlled natural language input, have stricter requirements on the notation and structure of proofs being written.

Machine learning for formalization. Machine learning techniques have been used in the formalization of informal proofs. The work of Yuhuai Wu et al. [20] based on large language models can correctly transform 25.3% of the solution for mathematical competition problems in MATH dataset [12] into formal specifications in Isabelle/HOL. In the work of Jiang et al. [13], they introduce Draft, Sketch, and Prove (DSP), a method that maps informal proofs to formal proof sketches, The accuracy was improved to 39.3% on the same dataset [12]. And the automatic theorem prover implemented by GPT-f [17] achieved a completion rate of 56.22% on their test set. The purpose of formalizing proofs with AI-based methods is to ensure that the proven proposition is correct, especially when proving previously unproven propositions in mathematical research. Since the target language of the transformation is often tactic-based proof languages that differ greatly from natural language, these works mainly use the original proof to guide theorem provers to complete the proof, rather than truly transforming the original proof into a formalized proof. In this case, because the language we designed combines formal rigor with similarity to natural language, if AI automatic translation is set to use our language as the target, it will complement our work well and lead to better results.

8 Conclusion

In this paper, we present our design of a natural-formalized proof language and the implementation of a mathematical proof checker. Compared to existing tactic-based proof language, our proof language has more expressive power thanks to the incorporation of partial proof. To cope with the characteristics of natural language proof, such as context-dependent semantics and overloading of notation, tactic language provides a fine-grained but cumbersome formal specification in the hope that someone can correctly reproduce the proof, while our proof language can automatically fix these problems through static analysis. All these factors result in better readability and an easier formalization process.

Regarding the process of proof checking, we implement a solver manager responsible for managing various automated proof checking strategies. It selects the most suitable strategies based on the form and contextual environment of the proposition to assess its correctness. Building upon this, the proof checker takes the proof and the proof goal provided by the static analyser, applies the appropriate checking methods, updates the proof goal iteratively, and ultimately completes the checking process.

Acknowledgements This material is based upon work supported by NSF China 92370201.

References

1. Appel, K.I., Haken, W.: Every Planar Map is Four Colorable, vol. 98. American Mathematical Soc. (1989)
2. Barras, B., et al.: The coq proof assistant reference manual. INRIA, version 6(11) (1999)
3. Bauer, G., Wenzel, M.: Calculational reasoning revisited an Isabelle/Isar experience. In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 75–90. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44755-5_7
4. Bertot, Y., Castran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions, 1st (edn.) Springer, Incorporated (2010)
5. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda – a functional language with dependent types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 73–78. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_6
6. Carl, M., Krapf, R.: Diproche–ein automatisierter tutor für den einstieg ins beweisen. Digitale Kompetenzen und Curriculare Konsequenzen, p. 43 (2020)
7. Carter, N.C., Monks, K.G.: Using the proof-checking word processor lurch to teach proof-writing (2014)
8. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
9. Delahaye, D.: A tactic language for the system Coq. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNAI, vol. 1955, pp. 85–95. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44404-1_7
10. Donnelly, C.: Bison the YACC-compatible parser generator. Technical report, Free Software Foundation (1988)
11. Gonthier, G., et al.: Formal proof-the four-color theorem. Notices AMS **55**(11), 1382–1393 (2008)
12. Hendrycks, D., et al.: Measuring mathematical problem solving with the math dataset (2021)
13. Jiang, A.Q., et al.: Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. arXiv preprint arXiv:2210.12283 (2022)
14. Lample, G., et al.: Hypertree proof search for neural theorem proving. In: Advances in Neural Information Processing Systems, vol. 35, pp. 26337–26349 (2022)
15. Levine, J.: Flex & Bison: Text Processing Tools. O'Reilly Media, Inc. (2009)
16. Paulson, L.C.: Isabelle: A Generic Theorem Prover. Springer, Berlin (1994)
17. Polu, S., Sutskever, I.: Generative language modeling for automated theorem proving. arXiv preprint arXiv:2009.03393 (2020)
18. Wemmenhove, J., Beurskens, T., McCarren, S., Moraal, J., Tuin, D., Portegies, J.: Waterproof: educational software for learning how to write mathematical proofs. arXiv preprint arXiv:2211.13513 (2022)
19. Wenzel, M.: Isar - a generic interpretative approach to readable formal proof documents. In: International Conference on Theorem Proving in Higher Order Logics (1999)
20. Wu, Y., et al.: Autoformalization with large language models. Advances in Neural Information Processing Systems, vol. 35, pp. 32353–32368 (2022)
21. Xie, L., Hui, Z., Cao, Q.: A natural formalized proof language (long version). arXiv preprint arXiv:2405.07973 (2024)