



上海交通大学学位论文

对自然形式化证明语言的研究

姓 名：惠志成

学 号：520021910569

导 师：马叶涵

学 院：巴黎卓越工程师学院

专业名称：信息工程（中法合作办学）

申请学位层次：学士

2024 年 05 月

**A Dissertation Submitted to
Shanghai Jiao Tong University for Bachelor's Degree**

TOWARDS NATURAL FORMAL PROOF LANGUAGES

Author: Zhicheng Hui

Supervisor: Yehan Ma

School of SJTU Paris Elite Institute of Technology

Shanghai Jiao Tong University

Shanghai, P.R. China

May, 2024

上海交通大学

学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全知晓本声明的法律后果由本人承担。

学位论文作者签名：惠志成

日期：2024年 06月 20日

上海交通大学

学位论文使用授权书

本人同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。

本学位论文属于：

☒ 公开论文

☐ 内部论文，保密 ☐ 1 年 / ☐ 2 年 / ☐ 3 年，过保密期后适用本授权书。

☐ 秘密论文，保密 ____ 年（不超过 10 年），过保密期后适用本授权书。

☐ 机密论文，保密 ____ 年（不超过 20 年），过保密期后适用本授权书。

（请在以上方框内选择打“√”）

学位论文作者签名：惠志成

指导教师签名：

日期：2024年 06 月 20日

日期： 年 月 日

摘 要

自然语言数学证明中存在着歧义和省略步骤，它们给数学证明的自动检查带来了困难。现有的自动检查数学证明的方法包括在定理证明器中形式化数学证明，而形式化的工作通常由人工智能完成。然而，人工智能方法易于产生错误，并且会产生过度推理的问题。此外，定理证明器提供的形式语言不易被非专业的用户读懂和使用。在这篇文章中，我们设计了一个接近自然语言的形式证明语言，并且实现了一个证明检验系统。该证明语言可以由自然语言数学证明直接转换而来，并且允许我们使用静态分析和添加部分证明结构的方法处理自然语言证明中的问题。这种设计结合了自然语言的表达能力和自动定理证明的可靠性，是对现有形式证明语言的改进。

关键词：形式语言，定理证明，静态分析

ABSTRACT

Mathematical proof checker faces challenges due to ambiguity and omitted steps in natural language proofs. Current methods include formalizing proofs in theorem provers, usually by artificial intelligence. However, AI-based approaches are prone to unpredictable errors and excessive inference, and formal languages offered by theorem provers are not easy for non-professional users to read and use. In this paper, we design a natural-language-like proof language and implement a proof checking system. This language can be directly converted from natural language, and supports the handling of issues in informal proofs through static analysis and the structure of partial proofs. This design combines the expressive ability of natural language and the reliability of automated theorem proving, resulting in an improved mathematical proof language.

Key words: formal language, theorem proving, static analysis

Contents

摘 要	I
ABSTRACT	II
Chapter 1 Introduction	1
1.1 Foreword	1
1.2 The main content of this paper	2
Chapter 2 Analysis of Natural Language Proofs	4
2.1 Running Example	4
2.2 Arborescent Structure	4
2.3 Context-dependent Semantics	5
2.4 Overloading of Notation	6
2.5 Conclusion	6
Chapter 3 The Design and Formal Semantics of Proof Language	7
3.1 Formal Definitions of Proof Language	7
3.2 Formal Definitions of Abstract Syntax Tree	9
3.3 Language Constructs	11
3.4 Basic Definitions of Formal Semantics	15
3.5 Formal Semantics of Proof Language	16
3.6 Conclusion	19
Chapter 4 The Design of Proof Checking System	21
4.1 Proof Parser	21
4.2 Static Analyzer	21
4.3 Proof Checker	23
4.4 Solver Manager	23
4.5 Conclusion	24

Chapter 5 Evaluation and Related Works	25
5.1 Performance	25
5.2 Readability and Proof Length	26
5.3 Related Works	26
5.4 Conclusion	27
Chapter 6 Conclusion	29
6.1 Main Conclusions	29
6.2 Research Outlook	29
Bibliography	31
List of Research Achievements	32
Acknowledgements	33

Chapter 1 Introduction

1.1 Foreword

Mathematical proof can be viewed as the process of deriving certain mathematical statements under the accepted set of axioms and rules of inference. Generally, much as mathematical proofs are based on logic, the majority of them are still written in terms of natural language which itself admits some level of ambiguity. That will sometimes make people falsely believe results erroneously proven unless enough care is taken.

Consequently, formal proofs are employed to manage such challenge. They are fully written in the form of a sequence of well-formed formulas of a formal language, which renders them rigorous, unambiguous and mechanically checkable, while tedious to construct manually. It is therefore common practice to construct formal proofs by theorem provers instead of by hand. A theorem prover is a software offering their own formal language to write the proof with (in the following we call that a proof language), and equipped with a checker to mechanically check the correctness of the formal proof. Examples include the Ltac^[1] proof language for the proof assistant Coq^[2], and Isar^[3] proof language for Isabelle^[4]. The design of proof languages remains an active research area due to its relation to mathematical expressiveness.

One of its applications is the automated checking of mathematical proofs. Employed for educational purposes, it can help teachers grade students' exercises efficiently and reduce oversights caused by fatigue or other factors. Also, it can be used to help review new math discoveries. Due to these applications, automatic math proof checking has also attracted a lot of attentions in researches, leveraging new natural language processing and theorem proving techniques. In order for the proofs to be checked, they should firstly be written in a formal language. To deal with the gap between informal and formal proofs, either the proof is formalized manually, or via machine learning techniques. However, a manual transformation generally involves intellectual burden and an expertise of theorem provers. In the case of autoformalization realized by machine learning, there is no guarantee that the formalized proof generated in such way faithfully reflects the original proof, contradicting the goal of ensuring rigorousness. Therefore, a transformation more direct and loyal to the original should be

proposed.

In addition, whether the proof is transformed through machine learning techniques or written directly in a theorem prover, the final result is usually a formal proof written in tactic-based proof language. Compared to traditional mathematical proofs, these tactic-based formal proofs look more like code written in programming languages. This difference makes it difficult to formalize flexible natural language proofs as tactic-based proofs in some cases.

1.2 The main content of this paper

In this paper, we present our design of a natural formalized proof language and the implementation of a mathematical proof checker. Compared to existing formal tactic proof language, our proof language has more expressive power thanks to the incorporation of partial proof. To cope with the problems of natural language proof, such as semantic vagueness and abuse 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 better readability and an easier process of formalization.

In order to facilitate the proof checking process, we write a parser to convert the natural formalized proof into an abstract syntax tree in Coq. The parser is implemented by flex^[5] and bison^[6]. After that, we use a static analyzer to eliminate the problems originating from natural language proofs. Regarding to the checking process, we implement a checker which recurses on the abstract syntax tree to check each step of the proof, along with a solver manager to handle various automated proof strategies. It is responsible for selecting suitable automated proof strategies based on the form and contextual environment of the propositions. We also describe the theorem manager, which allows us to configure the available theorems during proof checking. This eases the development of the proof checking system and offers additional modularity since we can alter the set of embedded knowledge.

In the following of this thesis, we will first motivate our work through analyzing the characteristics of natural language proofs and the problems caused by them in Chapter 2. We then present the design of our proof language along with that of its abstract syntax tree in Chapter 3, the formal semantics of our proof language is also defined. After that, we describe in detail the entire workflow and the components of the proof checker in Chapter 4.

We then evaluate our proof language and proof checking system in Chapter 5 by comparing with similar tools and running on a set of examples. Finally, we draw a conclusion in Section 6.

Chapter 2 Analysis of Natural Language Proofs

2.1 Running Example

Fig 2-1 shows a proof of the monotone convergence theorem through the supremum theorem. The proof steps are labeled by line number and we can observe several characteristics of natural language proofs from it. Based on that, we also present the difficulties consequent upon the task of formalization using existing formal proof languages like Coq, as long as how our work circumvents these difficulties.

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 $\lim_{n \rightarrow +\infty} a_n = A$.
 (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 $\lim_{n \rightarrow +\infty} a_n = A$,
 (9) which proves the theorem. ■

Figure 2-1 Proof of monotone convergence theorem.

2.2 Arborescent Structure

Unlike how it is written line-by-line on paper, the structure of the proof is not linear but arborescent. When a subgoal of proof is posed, as in line 3 of the example, the whole proof is actually divided into two parts: proof of the subgoal contained in lines 3-8, and the rest of the proof. It is also possible that we set some assumptions and deduce some results under these assumptions for later use, as in lines 4-7 of the example where we suppose $\varepsilon > 0$ and deduce $\forall n, n > N \implies A - \varepsilon < a_n < A + \varepsilon$. After that, we leave the scope of the assumption

$\varepsilon > 0$ and finish the remaining part of the proof.

The correct recognition of such arborescent structure is vital to proof formalization and checking. In Coq, we can use the “assert” tactic  to pose a subgoal and prove it. However, the latter case where there are no explicitly declared subgoals is beyond the capability of tactics, since they are supposed to be transformations of the proof state. Although we can make additional efforts to find out what should be filled in the “assert” tactic, that calls for extra inference. Besides, an “assert” tactic can pose only one subgoal at a time, while we can derive several results under the assumptions.

In our work, a construct 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 transformations on the structure of the proof during the task of formalization.

2.3 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 of the goal, (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 a 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 states. That explains the difficulties 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 tem-

porarily preserve the context-dependent semantic, thus allowing the resolution to be postponed until we can perform **static analysis** on the formalized version of the proof.

2.4 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 suite a . Another typical example is the extensive usage of “ $f(x)$ ” for representing the function “ f ” itself. Similar to context-dependent semantic, 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.

2.5 Conclusion

The characteristics of natural language mathematical proofs corresponds to human intuition and way of thinking. This is in contrast with how they are dealt with in computer system, which requires a maximum of rigorousness in order to apply formal methods. Such a gap renders autoformalization indirect and nontrivial to realize.

We propose to postpone the handling of these issues after parsing the natural language proof into its internal representation. Basically, the three characteristics mentioned above are in turn carried to the abstract syntax tree, which is much easier to analyse and manipulate. In particular, we can define specific rules of proof transformation.

Chapter 3 The Design and Formal Semantics of Proof Language

3.1 Formal Definitions of Proof Language

We aim to create a proof language that mirrors the structure of natural language. This natural formal proof language is designed to reflect the hierarchy found in natural language proofs. At the highest level, it includes proof steps, followed by propositions and terms. We have conducted extensive research into natural language proofs to incorporate common proof patterns into our language. This proof language adheres to predefined grammatical rules but is still readable as natural language.

Since natural language offers various ways to express the same meaning, we choose a subset of these expressions as keywords for the definition of natural formalized proof. In the formal definition given below, we choose only one of the possible options as example. Furthermore, due to the complexity of mathematical expressions and propositions involving various mathematical concepts, we do not elaborate on them in detail here.

```

<proof> ::= <proof statement>
| <proof statement> ‘,’ <proof>
| ‘The following proves’ <math proposition> ‘{’ <proof> ‘}’ <proof>
| ‘It suffices to prove’ <math proposition> <proof>
| <proof_action_statement> ‘{’ <proof> ‘}’ <proof>

<proof statement> ::= ‘which proves the proposition.’
| ‘Use’ <definition> ‘to prove’
| ‘Use’ <theorem> ‘to prove’
| <since_clause> ‘then’ <math proposition>
| <proof_action_statement>
| ...

<since_clause> ::= ‘Obviously’
| ‘Similarly’
| ‘Since’ <knowledge>

```

| ‘Because’ $\langle \text{math proposition} \rangle$
 | ‘By’ $\langle \text{prop_action_statement} \rangle$
 | ...
 $\langle \text{prop_action_statement} \rangle ::= \text{adding } \{ \text{proposition name} \}^*$
 | $\langle \text{prop_transformation} \rangle$ ‘on the both sides of the equality’
 | ...
 $\langle \text{prop_transformation} \rangle ::= \text{‘adding’ } \langle \text{math expression} \rangle$
 | ‘squaring’
 | ‘taking the logarithm’
 | ‘taking the derivative of’ $\langle \text{variable name} \rangle$
 | ...
 $\langle \text{proof_action_statement} \rangle ::= \text{‘Let’ } \langle \text{variable name} \rangle$
 | ‘Let’ $\langle \text{variable name} \rangle \langle \text{math proposition} \rangle$
 | ‘There exists’ $\langle \text{math expression} \rangle$
 | ‘There exists’ $\langle \text{variable name} \rangle$ ‘such that’ $\langle \text{math proposition} \rangle$
 | ‘Suppose’ $\langle \text{math proposition} \rangle$
 | ‘Suppose’ $\langle \text{variable name} \rangle$ ‘such that’ $\langle \text{math proposition} \rangle$
 | ‘Set’ $\langle \text{variable name} \rangle$ ‘=’ $\langle \text{math expression} \rangle$
 | ...
 $\langle \text{knowledge} \rangle ::= \langle \text{theorem} \rangle$
 | $\langle \text{definition} \rangle$
 | $\langle \text{property} \rangle$

In the formal definitions above, we use the notation $\{\text{nonterminal}\}^*$ to denote a list of nonterminals in the curly bracket. The nonterminal symbols $\langle \text{math proposition} \rangle$ and $\langle \text{math expression} \rangle$ follows their literal meaning. The nonterminal symbols $\langle \text{theorem} \rangle$, $\langle \text{definition} \rangle$, and $\langle \text{property} \rangle$ are merely alias of mathematical proposition.

3.2 Formal Definitions of Abstract Syntax Tree

To perform proof checking, the natural formal proof language is further translated into an abstract syntax tree via a parser.

We first demonstrate the definitions of $\langle term \rangle$ and $\langle prop \rangle$, they represent expressions and propositions in the proof.

$$\begin{aligned}
 \langle term \rangle ::= & \text{ 'TNum' } \langle number \rangle \\
 & | \text{ 'TInfy' } \langle infinity \rangle \\
 & | \text{ 'TConst' } \langle const \rangle \\
 & | \text{ 'TUnOp' } \langle unaryTermOperator \rangle \langle term \rangle \\
 & | \text{ 'TBinOp' } \langle binaryTermOperator \rangle \langle term \rangle \langle term \rangle \\
 & | \text{ 'TApply' } \langle term \rangle \langle term \rangle \\
 & | \text{ 'TBinder' } \langle binder \rangle \langle identifier \rangle \langle term \rangle \\
 & | \text{ 'TVar' } \langle identifier \rangle \\
 & | \text{ 'TInterval' } \langle intervalType \rangle \langle term \rangle \langle term \rangle \\
 & | \text{ 'TSet' } \{ identifier \}^* \langle term \rangle \langle prop \rangle \\
 \\
 \langle prop \rangle ::= & \text{ 'PUnPred' } \langle unaryPredicate \rangle \langle term \rangle \\
 & | \text{ 'PBinPred' } \langle binaryPredicate \rangle \langle term \rangle \langle term \rangle \\
 & | \text{ 'PCBinPred' } \langle binaryPredicate \rangle \langle term \rangle \langle term \rangle \langle propContext \rangle \\
 & | \text{ 'PLongOrder' } \langle order \rangle \langle term \rangle \langle prop \rangle \\
 & | \text{ 'PUnOp' } \langle unaryPropOperator \rangle \langle prop \rangle \\
 & | \text{ 'PBinOp' } \langle binaryPropOperator \rangle \langle prop \rangle \langle prop \rangle \\
 & | \text{ 'PQuant' } \langle quantifier \rangle \langle identifier \rangle \langle prop \rangle \\
 \\
 \langle binder \rangle ::= & \text{ 'SeqLimitB' } \\
 & | \text{ 'LambdaB' }
 \end{aligned}$$

We can define $\langle proof \rangle$ based on the definitions of props and terms, a proof is composed of several proof steps of different sorts. Each proof step models a statement in a natural language proof. Below shows a subset of its definitions.

$$\begin{aligned}
\langle proof \rangle ::= & \text{'ProofAction'} \langle action \rangle \langle proof \rangle \\
& | \text{'PoseWithoutProof'} \langle fwd \rangle \langle prop \rangle \langle proof \rangle \\
& | \text{'PoseAndProve'} \langle fwd \rangle \langle prop \rangle \langle proof \rangle \langle proof \rangle \\
& | \text{'ClaimSuffice'} \langle bwd \rangle \langle prop \rangle \langle proof \rangle \\
& | \text{'ProveSuffice'} \langle bwd \rangle \langle prop \rangle \langle proof \rangle \langle proof \rangle \\
& | \text{'ConclWithoutProof'} \langle fwd \rangle \\
& | \text{'ConclAndProve'} \langle fwd \rangle \langle proof \rangle \\
& | \text{'PosePartialProof'} \langle poseAction \rangle \langle proof \rangle \langle proof \rangle \\
& | \text{'EndPartialProof'}
\end{aligned}$$

$$\begin{aligned}
\langle fwd \rangle ::= & \text{'FNoHint'} \\
& | \text{'FDefinition'} \langle definition \rangle \\
& | \text{'FTheorem'} \langle theorem \rangle \\
& | \text{'FAddEqn'} \{ identifier \}^* \\
& | \text{'FDeriBothTerms'} \langle identifier \rangle \\
& | \dots
\end{aligned}$$

$$\begin{aligned}
\langle bwd \rangle ::= & \text{'BNoHint'} \\
& | \text{'BContra'} \\
& | \dots
\end{aligned}$$

$$\begin{aligned}
\langle action \rangle ::= & \text{'AIntros'} \langle identifier \rangle \\
& | \text{'AExists'} \langle term \rangle \\
& | \text{'ASuppose'} \langle prop \rangle \\
& | \text{'ASet'} \langle identifier \rangle \langle term \rangle \\
& | \text{'ASetProp'} \langle prop \rangle \\
& | \text{'AExistVar'} \langle identifier \rangle
\end{aligned}$$

$$\begin{aligned}
\langle poseAction \rangle ::= & \text{'APoseVar'} \langle identifier \rangle \{ prop \}^* \\
& | \text{'APoseProp'} \langle prop \rangle
\end{aligned}$$

3.3 Language Constructs

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^[8].

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

The design of `ProofAction` refers to some tactic languages, such as the tactic *intro* and *exists* of Coq^[9], which deal with the quantifiers in the proposition to be proved. That corresponds to our proof action `AIntros` and `AExists`. However, the variety of proof actions is richer than that of tactic language. For example, `ASuppose` 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.

`ASet` 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.

`ASetProp` is a generalization of `ASet`. Instead of introducing a new variable through an equation, `ASetProp` introduces new variables through a proposition. Fig 3-1 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, `AExistVar` indicates the action of instantiating the variable mentioned by an existential quantifier in the last premise in the proof goal. The usage of `AExistVar` is further demonstrated in Section 5.2 when we perform static analysis.

In the upper part of Fig 3-1, 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 $(a_n)_{n \in \mathbb{N}}$ admits a convergent subsequence. Both of the two proof steps will add corresponding assumptions to the proof goal.

Natural formal proof:

We note A as the limit of the sequence $(a_n)_{n \in \mathbb{N}}$.
 ... subsequent proof

Abstract syntax tree:

```
ProofAction (ASet "A" (TBinOp RLim (TInfty PositiveInfty) (TBinder LambdaB "n"
(TApply (TVar "a") (TVar "n"))))) (... subsequent proof)
```

Natural formal proof:

We note $(a_{n_k})_{k \in \mathbb{N}}$ as the convergent subsequence of $(a_n)_{n \in \mathbb{N}}$.
 ... subsequent proof

Abstract syntax tree:

```
ProofAction (ASetProp (PBinOp CAnd (PBinPred IsSubseq (TBinder LambdaB "k"
(TApply (TVar "a") (TApply (TVar "seq_n") (TVar "k"))))) (TVar "a")))
(PUnPred Convergent (TBinder LambdaB "k" (TApply (TVar "a") (TApply (TVar
"seq_n") (TVar "k")))))) (... subsequent proof)
```

Figure 3–1 Example of ProofAction.

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.

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 `PoseAndProve`. `PoseAndProve` 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. Fig 3–2 shows its usage scenario, which corresponds to the line 3 of Fig 2–1.

Since the hint of using the definition of limit is far from sufficient 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, `FAddEqn` denotes adding several equations together to get a new one, and `FDeriBothTerms` denotes taking a derivative from both sides of an equation. If no

Natural formal proof:

We use the definition of limit to show that: $\lim_{n \rightarrow +\infty} a_n = A$.

... subgoal proof

... subsequent proof

Abstract syntax tree:

```
PoseAndProve (FDefinition SeqLimit) (PBinPred REq (TBinOp RLim
(TInfty PositiveInfty) (TBinder LambdaB "n" (TApply (TVar "a") (TVar "n"))))
(TVar "A")) (... subgoal proof) (... subsequent proof)
```

Figure 3–2 Example of PoseAndProve.

method is indicated, FNoHint is filled in. It is quite easy to incorporate new methods so this set is highly expandable.

ClaimSuffice & ProveSuffice 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 `PoseAndProve`.

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. `ConclAndProve` 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 2–1, 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 Generally, we always keep a record of the current goal, which is either the overall proposition to be proved, either a subgoal we posed in the hope of coming in handy later, which is the case of `PoseAndProve` in our proof language.

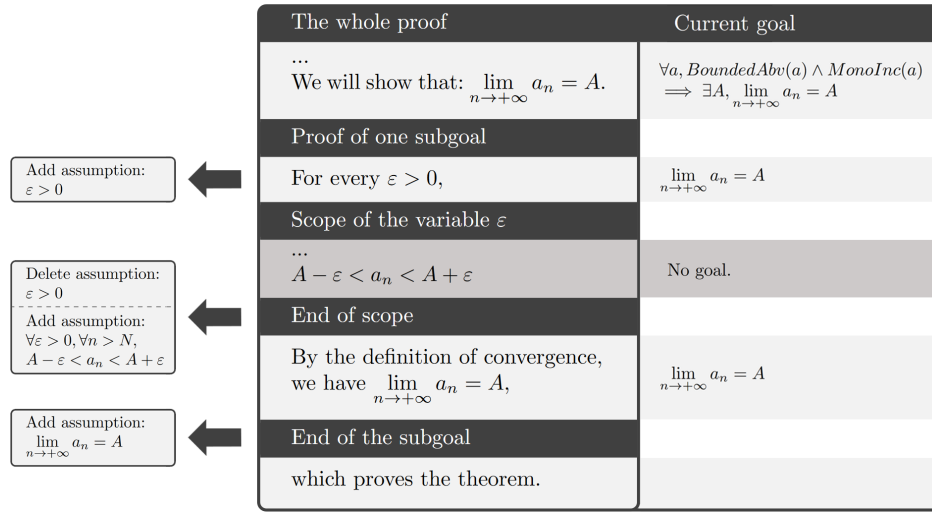


Figure 3-3 Illustration of how the proof goal changes.

This is especially true in theorem provers, whose tactic language requires knowing the most exact form of the proof goal to be able for human to use.

But posing a subgoal is not the only way to generate new available conditions. Sometimes we just pose certain assumptions along with related variables and proceed, no prediction is made in advance to indicate what is to be attained. In other words, in that circumstance we do not have a specific goal, and the exploration can be terminated freely without the risk of not passing the check. See Fig 3-3 for an illustration. We will use the name “partial proof” to emphasize the vacancy of goal. The notion of partial proof is one innovation of our proof language, which can not be found in tactic-based proof languages, thus offering additional expressive ability.

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 `poseAction` is the grammar. `APoseVar` poses a new temporary variable along with related assumptions, while `APoseProp` 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`.

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  $\varepsilon$  ... EndPartialProof)
(... subsequent proof)
```

Figure 3–4 Example of PosePartialProof.

In Fig 3–3, the variable ε and the proposition $\varepsilon > 0$ are posed to start a partial proof. When reaching the result $\forall n > N, A - \varepsilon < a_n < A + \varepsilon$ at the end of the partial proof, it is expanded to $\forall \varepsilon > 0, \forall n > N, A - \varepsilon < a_n < A + \varepsilon$, which is subsequently used to prove convergence. Fig 3–4 shows the how a partial proof looks like, which corresponds to line 4 of Fig 2–1.

3.4 Basic Definitions of Formal Semantics

In the following, We will demonstrate a selected set of the formal semantics of our proof language. Before that, we give the interpretation of the notations that appear in the formal semantics, as well as the concept of proof goal transformation.

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.^[8]

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 `PoseAndProve` component in our proof language.

As discussed in Section 2, there are cases where subgoals are not explicitly stated in advance, instead, We simply pose certain assumptions and proceed to derive the subgoals. This can be viewed as a partial transformation of the proof goal. The signification of partial can either be that of partial function or incomplete proof goal.

Notations 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 \square .

In order to cope with variable naming, we define the operation of variable substitution in a proposition. The notation $P[t/x]$ signifies the proposition resulting from substituting all the occurrence of free variable x in P by the term t . 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 .

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 also define a constant QED such that the proof is successfully completed when $(pr, A, C) = QED$.

3.5 Formal Semantics of Proof Language

Proof Action

$$\begin{array}{c}
 \frac{v \notin FV(A \cup \{\forall x.C\})}{(\mathbf{ProofAction} \ (\mathbf{AIntros} \ v) \ pr, A, \forall x.C \rightarrow (pr, A, C[v/x]))} \\
 \\
 \frac{FV(t) \subset FV(A \cup \{\exists x.C\})}{(\mathbf{ProofAction} \ (\mathbf{AExists} \ t) \ pr, A, \exists x.C \rightarrow (pr, A, C[t/x]))} \\
 \\
 \frac{\text{PropEquivalent } P \ Q}{(\mathbf{ProofAction} \ (\mathbf{ASuppose} \ P) \ pr, A, Q \Rightarrow C) \rightarrow (pr, A \cup \{P\}, C)} \\
 \\
 \frac{\text{TermWellDefined } t \ A \quad v \notin FV(A \cup \{C\})}{(\mathbf{ProofAction} \ (\mathbf{ASet} \ v \ t) \ pr, A, C) \rightarrow (pr, A \cup \{v = t\}, C)} \\
 \\
 \frac{\text{PropWellDefined } P \ A}{(\mathbf{ProofAction} \ (\mathbf{ASetProp} \ P) \ pr, A, C) \rightarrow (pr, A \cup \{P\}, C)}
 \end{array}$$

$$v \notin FV(A \cup \{\exists x.P\} \cup \{C\})$$

$$\text{(ProofAction (AExistVar } v) \text{ } pr, A \cup \{\exists x.P\}, C) \rightarrow (pr, A \cup \{\exists x.P\} \cup \{P[v/x]\}, C)$$

Proof actions are proof steps that directly manipulates the proof goal, usually when some conditions are met. In the semantic definitions above, $\text{PropEquivalent } P \ Q$ means the proposition P and Q are equivalent, $\text{TermWellDefined } t \ A$ means the term t is well-defined under the set of premises A , and $\text{PropWellDefined } P \ A$ means the proposition P is well-defined and satisfied under the set of premises A . These predicates are implemented as functions within our proof checking system, however, they are too cumbersome to define formally.

The behavior of **AIntros** v is to remove the universal quantifier in the conclusion $\forall x.C$ and replace the variable x in C by the variable v , when the variable v does not occur freely in the proof goal. Similarly, the behavior of **AExists** t is to remove the existential quantifier in the conclusion $\exists x.C$ and replace the variable x in C by the term t , when the term t does not contain free variables other than those in the proof goal. And the behavior of **ASuppose** P is to remove the implication in the conclusion $Q \Rightarrow C$ and add P to the premises when P and Q are equivalent propositions.

ASet $v \ t$ adds the equation $v = t$ to the premises, where v does not occur freely and the term t must be well-defined in the proof goal. The requirement for well-definiteness ensures that a term does not appear before its existence has been proved in the proof goal. The behavior of **ASetProp** P is similar, it incorporates the proposition P possibly containing new variables into the premises. This can be useful when P is not an equation.

Finally, when the last premise in the proof goal is an existential proposition, the statement **AExistVar** v instantiates the existential quantifier with the variable v , by adding the properties verified by v to the premises. It is required that the variable v does not occur freely in the proof goal.

Direct Forward Proof

$$\text{PropDeducible } P \ A$$

$$\text{(PoseWithoutProof FNoHint } P \text{ } pr, A, C) \rightarrow (pr, A \cup \{P\}, C)$$

$$\text{TheoremApplicable } thm \ P \ A$$

$$\text{(PoseWithoutProof (FTheorem } thm) \text{ } P \text{ } pr, A, C) \rightarrow (pr, A \cup \{P\}, C)$$

$$\frac{}{(\mathbf{ConclWithoutProof} \ \mathbf{FNoHint}, A \cup \{C\}, C) \rightarrow QED}$$

$$\text{TheoremApplicable } thm \ C \ A$$

$$\frac{}{(\mathbf{ConclWithoutProof} \ (\mathbf{FTheorem} \ thm), A, C) \rightarrow QED}$$

The predicate $\text{PropDeducible } P \ A$ indicates that the proposition P can be deduced from the set of premises A . Similarly, $\text{TheoremApplicable } thm \ P \ A$ signifies that the theorem thm can be applied to the set of premises A to yield the result P . It is important to note that the notion of deducibility here is algorithmic rather than theoretical: a proposition is considered deducible if it can be derived using our predefined set of solvers within a specified number of steps. The applicability of theorem is also defined by the theorem checker in our implementation. Again, we do not provide formal definitions but give descriptions in the following.

The behavior of $\mathbf{PoseWithoutProof} \ \mathbf{FNoHint}$ depends largely on the solvers. In the process of proof checking, the checker will send the proposition P and the current proof goal (A, C) to the solver manager, which checks whether the proposition P can be derived. Then, the checker adds P to the premises of the proof goal, and checks the rest of the proof pr .

Similarly, the behavior of $\mathbf{PoseWithoutProof} \ (\mathbf{FTheorem} \ thm)$ depends on the theorem checker in our implementation. The theorem checker checks two aspects: (1) whether the prerequisites for applying the theorem thm are satisfied, (2) and whether P can be derived as a conclusion from the theorem under those prerequisites.

The working process of the theorem checker begins with a pattern match between P and the conclusion of the theorem thm . This pattern match instantiates the variables and identifies the prerequisites of the theorem. Then, the theorem checker searches in the proof goal to see if all the prerequisites are satisfied. If the theorem checker is unable to match the conclusion or find all the prerequisites, then this step of transformation cannot be realized.

$\mathbf{ConclWithoutProof} \ \mathbf{FNoHint}$ transforms the proof goal into QED when the conclusion appears in the premises. $\mathbf{ConclWithoutProof} \ (\mathbf{FTheorem} \ thm)$ does the same transformation when the conclusion follows from the theorem thm by the theorem checker.

Subgoal Forward Proof

$$\frac{(pr, A, P) \rightarrow QED}{(\text{PoseAndProve FNoHint } P \text{ } pr \text{ } pr', A, C) \rightarrow (pr', A \cup \{P\}, C)}$$

The subgoal proof statement **PoseAndProveFNoHint** adds the proposition P to the premises on the condition that the proof of the subgoal is successfully completed. In the process of proof checking, the checker first checks the subgoal proof, and then returns to the main proof with the proven result.

Partial Proof

$$\frac{(pr, A \cup sP, \Box) \rightarrow (\text{EndPartialProof}, A', \Box) \quad FV(sP) \subset FV(A \cup \{C\}) \cup \{s\}}{(\text{PosePartialProof } (\text{APoseVar } s \text{ } sP) \text{ } pr \text{ } pr', A, C) \rightarrow (pr', A \cup (\text{AddVarDep } (A' \setminus (A \cup \{P\})) \text{ } s \text{ } sP), C)}$$

$$\frac{(pr, A \cup \{P\}, \Box) \rightarrow (\text{EndPartialProof}, A', \Box) \quad FV(\{P\}) \subset FV(A \cup \{C\})}{(\text{PosePartialProof } (\text{APoseProp } P) \text{ } pr \text{ } pr', A, C) \rightarrow (pr', A \cup (\text{AddPropDep } (A' \setminus (A \cup \{P\})) \text{ } P), C)}$$

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** $s \text{ } sP$ 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.

3.6 Conclusion

The resemblance of our proof language to natural language streamlines the task of formalization. We have included a rich set of proof constructs to be able to model common

proof patterns. The definitions of grammar rules allow us to make use of a parser generator to produce a parser of our proof language. The variation of expressions bearing the same meaning makes the natural formal language versatile and increase its similarities to real natural language.

The definitions of semantics allow us to write algorithms of proof checking. The checking algorithm first matches different proof steps, it then checks whether the intended transformation of proof goal corresponds to what is defined by the formal semantics. Note that some predicates in the. In the current implementation, we use a SMT solver with basic arithmetic theory for solver and pattern match for theorem matching.

Chapter 4 The Design of Proof Checking System

In this section, we primarily introduce the architecture of our proof checking system. The main components include the proof parser, static analyzer, proof checker, and solver manager.

For the workflow of our proof checking system. 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 analyzer 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. During the process of checking, the solver manager control how a proposition is checked.

4.1 Proof Parser

Proof parsing is the first step of the entire workflow. 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 Section 2. In our implementation, the lexer and parser are realized by flex and bison[5-6].

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.

4.2 Static Analyzer

As discussed in Section 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 analyzer 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 analyzer 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 2-1. In this example, the static analyzer 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.

interpretation (1) \longrightarrow proof goal beginning with \exists $\xrightarrow{\text{analyzer}}$ False

interpretation (2) \longrightarrow A unbounded thereafter $\xrightarrow{\text{analyzer}}$ False

interpretation (3) \longrightarrow A bounded thereafter $\xrightarrow{\text{analyzer}}$ True

The three potential semantic interpretations appear syntactically identical in the proof. Therefore, the static analyzer takes the responsibility for reflecting the result of the above analysis on the abstract syntax tree through modification. This is where the proof action `AExistVar` 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.

Handling Overloading of Notation. If there is only one possible interpretation for notation overloading, there is no need for analysis and the static analyzer 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 analyzer examines the appearance of free variables. As an example, the $\{a_n\}$ in line 2 of Fig. 2.1 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 .

4.3 Proof Checker

Proof checking is the final step of the entire workflow. 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 Section 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.

4.4 Solver Manager

In order to help with proposition checking, we also develop several solvers and a solver manager system. They are 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.

When users write proofs, they must adhere to the steps supported by the solvers. For example, AI-generated formal proofs sometimes use solvers like Z3 [10] 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.

4.5 Conclusion

The design and implementation of the proof checking system proves to be a robust and systematic approach to check formal proofs. The proof parser initiates the whole checking process. Given the difficulty of formalization, we simply let the parser to perform direct translation. The static analyzer then resolves ambiguities and ensures rigorosity by restructuring the proof, which is indispensable for handling context-dependent semantics and notation overloading.

In addition, the solver manager, with its specialized solvers, makes it possible to check the validity of a single proposition in the context of a proof goal. By limiting the solvers to relatively straightforward reasoning and avoiding overly powerful reasoning, our system maintains a balance between power and practicability.

Chapter 5 Evaluation and Related Works

In this section, we evaluate the time and memory overhead of our system. Considering that the form of natural language proofs may need to be elaborated manually to fit with the parser, we evaluate 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 proof checker and the dataset are available at: <https://github.com/Laplace-Demon/ProofGrader>. We will also discuss about the readability and proof length by contrasting with an example of Coq proof.

5.1 Performance

We test our system on a dataset of 52 mathematical proofs, Table 5-1 shows the time and memory overhead of the proof parser and the checker. We can observe that our proof parser and checker perform their task in a reasonable amount of time.

We note that the runtime and memory usage remain stable with respect to the topic of proofs, except for the time of checking function continuity proofs. This may be due to the use of inequality scaling solvers. Basically, this solver recurses on expressions using a searching algorithm. At each iteration it substitutes a part of the expression according to the direction of the inequality.

Examples		Parser		Checker	
Topic	Number	Time(ms)	Memory(KB)	Time(ms)	Memory(KB)
arithmetic	6	34	3584	477	3456
trigonometric functions	8	53	3712	1409	3456
exponential and logarithm	3	40	3840	506	3456
inequality	10	58	3712	528	3456
derivative	3	75	3328	760	3328
sequential limit evaluation	10	58	3840	701	3328
sequential limit proof	10	78	3968	670	3328
function continuity proof	2	68	3712	7758	3456

Table 5-1 Runtime and memory usage of parser and checker on different topics.

5.2 Readability and Proof Length

We compare in Table 5–2 the proof of the monotone convergence theorem formulated in our proof language with another formal Coq proof^[11].

Monotone convergence theorem	Natural language proof	Natural formal proof	Coq formal proof
Number of proof steps	9	12	48

Table 5–2 Number of proof steps and characters (space included) of the three proofs. For the natural language proof and the natural formal proof, each sentence is considered as one proof step. For the Coq proof, each tactic is considered as one proof step.

We note that the number of proof steps is similar between the natural language proof and our natural formal proof. In fact, there is almost a one-to-one correspondence between each of the proof steps, and they share largely the same structure. In contrast, the Coq proof, though being manually written, contains significantly more proof steps despite of using previously proven results and Coq’s automation functionalities.

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 each of the constructs bears a mnemonic name. The main difficulty of reading the abstract syntax tree lies in recognizing the lengthy expressions, whereas for the tactic-based Coq proof, the main difficulty lies in following the current proof state through the process of tactic application.

5.3 Related Works

Theorem provers and proof languages. Recent decades have seen the emergence of various proof languages. A well-known work is Ltac^[11] for the theorem prover Coq, which provides convenience for constructing proofs and facilitates better proof automation. Another famous one is that of Isar^[8] 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^[12] based on this to assist students in practicing proofs. Since they still choose to rely on the tactic library extended with the Ltac2 tactic lan-

guage, Waterproof has not actually gained stronger expressive power than these tactic-based proof language. Additionally, the proof assistant Agda^[13] 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 such as Diproche and Lurch^{[14][15]}, although they support controlled natural language input, have stricter requirements on the notation and structure of proofs being written^[8].

Machine learning for formalization. Machine learning techniques have been used in the formalization of informal proofs. The work of Yuhuai Wu et al.^[16] based on large language models can correctly transform 25.3% of the solution for mathematical competition problems in MATH dataset^[17] into formal specifications in Isabelle/HOL. In the work of Jiang et al.^[18], 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^[17]. And the automatic theorem prover implemented by GPT-f^[19] 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].

5.4 Conclusion

As mentioned in previous sections, the power of solver is elaborated to avoid excessive reasoning. The limited capability of solver ensures that checking terminates within a reasonable amount of time. The relatively long time spent on inequality scaling solver also suggests that we be cautious as to the potential problem of exponential explosion, taking into account

that we may encounter longer proofs.

Recent decades have seen the development of new proof languages and proof checking tools. But they are still not well-suited for direct translation of natural language proofs, in that they do not adequately capture the nuances and expressiveness of natural language proofs. Besides, the readability remains limited, resulting in steep learning curves.

For machine-learning assisted formalization, there is no guarantee that the formalized proof faithfully restored the original proof. Besides, our work complements theirs in that our proof language can be used as a target language of machine-learning-based formalization, which we believe will lead to better results.

Chapter 6 Conclusion

6.1 Main Conclusions

In this paper, we present our design of a natural formalized proof language and the implementation of a mathematical proof checker. Compared to existing formal tactic proof language, our proof language has more expressive power thanks to the incorporation of partial proof. To cope with the problems of natural language proof, such as semantic vagueness and abuse 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 better readability and a easier formalization process.

Regarding the proof checking process, we have implemented a solver manager to handle various automated proof strategies. It is responsible for selecting suitable automated proof strategies based on the form and contextual environment of the propositions to determine their correctness. Building upon this, the proof checker takes the current proof goal and the proof statements to be checked, applies corresponding checking methods based on the statement types, updates the proof goal, and ultimately completes the checking process.

6.2 Research Outlook

In the future, we plan to work on several aspects.

Firstly, as we have developed an overall framework for the proof language and the proof checking system, it will be of interest to further expand it to a wider range of mathematics, in order to validate and ameliorate the current solution and identify potential flaws. For example, we plan to systematically formalize the content of a textbook of mathematical analysis. This may include proposing proper ways of modeling different strategies of mathematical reasoning, such as proof by contradiction or proof by induction, and different mathematical objects, such as set, summation or integration. During this process, one of our main concerns will be to discover more usage scenarios of performing static analysis.

Secondly, in order to better evaluate the proof language and the proof checking system,

we plan to perform surveys amongst students on the readability of the proof language compared to existing ones. In addition, we plan to run experiments to measure the time and memory consumption of solvers and checkers, and to attempt optimization if this turns out to be unsatisfactory.

Thirdly, we will continue working on improving our algorithm of theorem application. Together with that is the development of a saturation-based theorem prover, both of which will be necessary if we want to enable more sophisticated theorem application, where identifying equivalent mathematical expressions and automatically deducing obvious conditions can be a common pattern.

Lastly, we will work on the development of theorem solvers. Since our goal is to check steps in mathematical proofs, these solvers are supposed to have moderate capability of reasoning to fit in the level of human intuition. Consequently, it is not feasible to directly adopt existing solvers because they are intended for complex reasoning. And we should elaborate on the level of automation when customizing theorem solvers.

Bibliography

- [1] DELAHAYE D. A Tactic Language for the System Coq[C]//PARIGOT M, VORONKOV A. Logic for Programming and Automated Reasoning. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000: 85-95.
- [2] TEAM. T C D. The Coq Proof Assistant.[EB/OL]. <http://coq.inria.fr>.
- [3] WENZEL M. Isar - A Generic Interpretative Approach to Readable Formal Proof Documents[C]//International Conference on Theorem Proving in Higher Order Logics. 1999.
- [4] PAULSON L C. Isabelle: A generic theorem prover[M]. Springer, 1994.
- [5] LEVINE J. Flex & Bison: Text Processing Tools[M]. "O'Reilly Media, Inc.", 2009.
- [6] DONNELLY C. BISON the YACC-compatible parser generator[J]. Technical report, Free Software Foundation, 1988.
- [7] BARRAS B, BOUTIN S, CORNES C, et al. The Coq proof assistant reference manual[J]. INRIA, version, 1999, 6(11).
- [8] XIE L, HUI Z, CAO Q. A Natural Formalized Proof Language (long version)[J]. arXiv preprint arXiv:2405.07973, 2024.
- [9] BERTOT Y, CASTRAN P. Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions[M]. 1st. Springer Publishing Company, Incorporated, 2010.
- [10] DE MOURA L, BJØRNER N. Z3: An efficient SMT solver[C]//International conference on Tools and Algorithms for the Construction and Analysis of Systems. 2008: 337-340.
- [11] FU Y, YU W. Formalization of the Equivalence among Completeness Theorems of Real Number in Coq[J/OL]. Mathematics, 2021, 9(1). <https://www.mdpi.com/2227-7390/9/1/38>. DOI: [10.3390/math9010038](https://doi.org/10.3390/math9010038).
- [12] WEMMENHOVE J, BEURSKENS T, MCCARREN S, et al. Waterproof: educational software for learning how to write mathematical proofs[J]. arXiv preprint arXiv:2211.13513, 2022.
- [13] BOVE A, DYBJER P, NORELL U. A brief overview of Agda—a functional language with dependent types[C]//Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22. 2009: 73-78.
- [14] CARL M. Number Theory and Axiomatic Geometry in the Diproche System[J/OL]. Electronic Proceedings in Theoretical Computer Science, 2020, 328: 56-78. <http://dx.doi.org/10.4204/EPTC.S.328.4>. DOI: [10.4204/eptcs.328.4](https://doi.org/10.4204/eptcs.328.4).
- [15] CARTER N, MONKS K. Lurch: a word processor that can grade students' proofs[C]//. 2013.
- [16] WU Y, JIANG A Q, LI W, et al. Autoformalization with large language models[J]. Advances in Neural Information Processing Systems, 2022, 35: 32353-32368.
- [17] HENDRYCKS D, BURNS C, KADAVATH S, et al. Measuring Mathematical Problem Solving With the MATH Dataset[Z]. 2021. arXiv: [2103.03874](https://arxiv.org/abs/2103.03874) [cs.LG].
- [18] JIANG A Q, WELLECK S, ZHOU J P, et al. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs[J]. arXiv preprint arXiv:2210.12283, 2022.
- [19] POLU S, SUTSKEVER I. Generative language modeling for automated theorem proving[J]. arXiv preprint arXiv:2009.03393, 2020.

List of Research Achievements

- [1] Xie, L., Hui, Z., Cao, Q.: A natural formalized proof language. In Proceedings of the The 18th International Symposium on Theoretical Aspects of Software Engineering (TASE'24), accepted, 2024

Acknowledgements

This thesis is composed during my exchange at Ecole Polytechnique. It is about a research project I have been doing with Lihan Xie since 2022, under the instruction of Qinxiang Cao.

I am indebted to my supervisor Qinxiang Cao, for introducing me to the field of programming language theory. And also for his comprehension, comments, suggestions as the project progressed.

It has been a pleasure to work with Lihan Xie. Not only did we share the tasks together, we also became good friends.

I appreciate Xiaochen Wang for coordinating between SJTU and me. It would have been much harder without her assistance.

I wish to thank Shijie Shen, Yingwei Tang, and Zihan Tang. We play League of Legends routinely and had great fun.

It has been gratifying to meet Ruoyu Su. Being with her is always filled with her own subtle wit, which has motivated me a lot.

Finally, my love goes to my family. Time is fleeting, and thanks for putting all this.

A NATURAL FORMALIZED PROOF LANGUAGE

本论文提出了一种自然形式化证明语言，并实现了一个数学证明检验系统，旨在解决自然语言数学证明中的歧义和省略步骤问题。现有的自动化数学证明检查方法主要依赖于人工智能形式化数学证明，但这些方法易于产生错误和过度推理，且现有的形式语言难以被非专业用户理解和使用。本文通过设计一种接近自然语言的形式证明语言，并结合静态分析和部分证明结构的方式，将自动定理证明的可靠性和自然语言的表达能力相结合。

自动化数学证明检查在教育和数学研究中具有一定的应用价值，它能够帮助教师高效批改作业，减少疏漏，同时辅助审查新的数学发现。为了实现对数学证明的检查，首先需要将证明书写成形式化语言。然而，现有的方法要么通过人工形式化，这需要具备使用定理证明器的专门知识，要么通过人工智能技术进行自动形式化，但这种方法无法保证忠实于原始证明，从而影响严谨性。

自然语言证明中的主要问题包括语义模糊、记号重载和上下文依赖语义等。例如，某些符号在不同上下文中具有不同含义，导致形式化时需要进行复杂的语义解析。此外，自然语言中的某些表达方式在目前流行的策略形式语言中难以直接表达，需进行适当的结构转换和信息补充。

自然语言证明的结构通常不是线性的，而是树状的。当提出一个子目标时，整个证明实际上分为两个部分：子目标的证明和其余部分的证明。正确识别这种树状结构对于证明形式化和检查至关重要。然而，现有的形式语言和策略难以直接表达这些复杂的结构，需要进行额外的推理和转换。

此外，根据上下文，相同的自然语言证明步骤可能有多种解释，相同的符号也可能表示不同的数学对象，即使它们之间的语义差异可能很小。在形式化过程中，我们需要明确这些语义差异，这使得使用现有的策略语言进行自动形式化十分困难。

本论文设计的自然形式化证明语言在结构上与自然语言相似，分为证明步骤、命题和表达式等层级。该语言既基于一套形式语法规则，又保持了自然语言的可读性。它提高了形式化语言的表达能力和灵活性，使其更接近自然语言。为了便于检查过程，我们设计了相应的抽象语法树，并通过解析器将该形式化证明转换为 Coq 中的抽象语法树进行后续的验证工作。证明的解析器使用 flex 和 bison 工具实现，它仅进行基于语法规则的简单翻译，而不对证明结构或命题进行进一步修改。

本论文通过具体的 BNF 规范，形式化地定义了该证明语言的语法和抽象语法树

结构。抽象语法树的设计则便于将自然形式化证明转换为机器可处理的形式，从而为后续的静态分析和证明检查奠定基础。

除语法外，本论文也通过描述各类证明语句对应证明目标的转化规则，形式化地定义了该证明语言的语义。例如，引入变量操作可以消除全称量词，并将其替换为新的变量；存在量词操作则用于处理存在量词的命题。每一步证明操作都定义了证明目标的转化。这些形式语义规则通过具体的语法和语义分析，确保了证明过程的正确性和严谨性。

本文描述的证明检验系统包括解析器、静态分析器、证明检查器和求解器管理器等主要组件。解析器首先将自然形式化证明转换为抽象语法树，静态分析器则负责消除语义模糊和补充省略的步骤。证明检查器根据证明语言的语义规则逐步检查证明步骤，并通过求解器管理器选择合适的检查策略。

静态分析器处理语义模糊和记号重载问题。它通过推理前后证明目标的转化关系，从而选择正确的语义，并在抽象语法树中进行修改体现。证明检查器则逐步检查每一步证明，生成一个包含布尔值的列表以表示每一步证明的正确性。在这个过程中，求解器管理器通过结合不同求解器验证各类数学命题，以确保推理过程的可靠性和准确性。用户在编写证明时，也相应地需要遵循求解器支持的步骤，从而确保原始证明过程的正确性。

本论文通过一组数学证明样例对系统进行了评估，涵盖算术、三角函数、指数对数、不等式、导数、序列极限和函数连续性等主题。评估结果表明，证明解析器和检查器均在合理的时间和内存开销下完成了任务。此外，本论文还对比了自然语言证明、自然形式化证明和 Coq 形式证明的可读性和证明步骤数量。结果显示，自然形式化证明与自然语言证明在步骤数量上相似，但比 Coq 形式证明更简洁、可读。

总而言之，本论文为自然语言数学证明形式化过程的问题提供了一种新的解决方案，并且通过实现证明检验系统来佐证该方案的可行性。未来的工作将致力于扩展和优化系统，以应对更广泛的数学领域和更复杂的证明任务。