International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2023, 13(3): 323–357, doi: 10.21655/ijsi.1673-7288.00303
©2023 by Institute of Software, Chinese Academy of Sciences. All rights reserved.

International Journal of Software and Informatics

Research Article

Coq Formalization of ZFC Set Theory for Teaching Scenarios

Xinyi Wan (万新熠), Ke Xu (徐轲), Qinxiang Cao (曹钦翔)

(School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200240, China)
Corresponding author: Qinxiang Cao, caoqinxiang@sjtu.edu.cn

Abstract Discrete mathematics is a foundation course for computer-related majors, and propositional logic, first-order logic, and the axiomatic set theory are important parts of this course. Teaching practice shows that beginners find it difficult to accurately understand abstract concepts, such as syntax, semantics, and reasoning system. In recent years, some scholars have begun introducing interactive theorem provers into teaching to help students construct formal proofs so that they can understand logic systems more thoroughly. However, directly employing the existing theorem provers will increase students' learning burden since these tools have a high threshold for getting started with them. To address this problem, we develop a prover for the Zermelo-Fraenkel set theory with the axiom of Choice (ZFC) in Coq for teaching scenarios. Specifically, the first-order logical reasoning system and the axiomatic set theory ZFC are formalized, and several automated proof tactics specific to reasoning rules are then developed. Students can utilize these automated proof tactics to construct formal proofs of theorems in a textbook-style concise proving environment. This tool has been introduced into the teaching of the course of discrete mathematics for freshmen. Students with no prior theorem-proving experience can quickly construct formal proofs of theorems including mathematical induction and Peano arithmetic with this tool, which verifies the practical effectiveness of this tool.

Keywords Coq; axiomatic set theory ZFC; first-order logic

Citation Wan XY, Xu K, Cao QX. Coq formalization of ZFC set theory for teaching scenarios, *International Journal of Software and Informatics*, 2023, 13(3): 323–357. http://www.ijsi.org/1673-7288/303.htm

The formal proof of mathematical theorems has been developed and studied extensively in recent years. Theorem proving tools, such as Coq[1], Isabelle[2], Mizar[3], and Lean[4] have gradually matured. These tools pave the way for the formalization of the proof construction for many mathematical theorems, including the famous four-color theorem[5], Gödel's incompleteness theorem[6], and Cauchy-Schwarz inequality. Gowers, a winner of the Fields Medal, believed that computers would play a key role in theorem proving and would change the study mode of theoretical mathematics in the future[7]. Hanna and Yan[8] pointed out that theorem provers could not only help mathematicians carry out theoretical study but also change the teaching method tailored to beginners of mathematical logic.

This is the English version of the Chinese article "针对教学场景的 ZFC 集合论 Coq 形式化. 软件学报, 2023, 34(8): 3549–3573. DOI: 10.13328/j.cnki.jos.006868"

Received 2022-09-05; Revised 2022-10-13; Accepted 2022-12-14; IJSI published online 2023-09-27

Discrete mathematics has been offered by most colleges and universities as a basic compulsory course in the undergraduate curriculum of computer-related majors in colleges and universities in China. Propositional logic, first-order logic, and the axiomatic set theory are important components of this course. Teaching practice reveals that undergraduates who are also beginners in mathematical logic have difficulty in learning and clarifying abstract concepts, such as grammar, semantics, and reasoning system. Moreover, students often fail to gain intuitive understanding through specific examples when they learn to describe mathematical propositions and mathematical proofs with first-order logic and the axiomatic set theory. The extension of interactive theorem provers to teaching is expected to alleviate this problem. Computer-aided automated checking can provide timely and accurate feedback on whether the propositions written by students are grammatically correct and whether they follow the reasoning rules of the logic system. In this way, it helps students construct a correct and strict proof and spare them from waiting for feedback from the teaching assistant which may take weeks. The real-time interaction feature of interactive theorem provers also allows students to dynamically see the reasoning process, thereby helping deepen their understanding of the logic systems.

In recent years, increasing scholars around the world have begun incorporating theorem provers into the courses of mathematical logic, especially courses in computer science majors, to assist teaching. For example, researchers at Oxford University have developed Jape for teaching predicate logic[9]. Cezary and Freek have developed the online teaching tool ProofWeb based on Coq[10], and Avigad has applied Lean to the teaching of first-order logic and set theory concepts[11]. However, most of the above efforts are based on the built-in logic of the theorem provers, which leads to many shortcomings in teaching:

(1) Most theorem provers, such as Coq, adopt a "bottom-up" reverse proof mode, where users continuously apply proof tactics to decompose the goal to be proven into a series of sub-goals, and the proof is ultimately completed with all the sub-goals as premises. Nevertheless, the commonly used proof modes also include forward reasoning, which means to derive new conditions from premises and ultimately obtain the goal to be proven.

(2) Students shall reason according to the reasoning rules of the logic system they are learning, instead of using the logic built into the theorem prover since it deviates from the teaching goal.

(3) Directly using a theorem prover to construct proofs is more like learning a programming language in the sense that students are required to spend a lot of extra time in familiarizing themselves with the instructions to operate a specific theorem prover. This approach not only increases learning costs but also renders the logic system easily overlooked.

(4) The proof code is usually excessively lengthy since theorem provers only provide basic proof tactics. Consequently, the large workload restrains students from constructing formal proofs of more complex theorems.

(5) The lack of a textbook-style concise symbol system is not conducive to students' understanding of the logic systems.

Briefly, the threshold for getting started with theorem provers developed for assisting the study of mathematical theories is high. Moreover, such theorem provers differ significantly from the form of the logic systems found in textbooks. Simply introducing them directly into teaching may increase learning costs and even confuse students as they are likely to mistake the logic built into a theorem prover for the logic system they are learning.

Considering these problems, some scholars, rather than using theorem provers, have instead developed simple proving tools tailored to teaching from scratch in programming languages. Breitner[12] developed Incredible Proof Machine[12], with which students could construct proofs by connecting the blocks representing propositions with those representing the reasoning rules of the first-order logic using the mouse in the visual interface. Lerner *et al.* proposed the visual interface Polymorphic block[13], with which students could connect proposition blocks with reasoning blocks in a way similar to jigsaw puzzle solving to complete the reasoning in a natural deduction system. These tools can provide a more conveniently interactive and interesting experience, but their development from scratch entails tremendous work. The existing interactive theorem provers have already provided a robust user interface involving proof interface, proof state, and error information. Developing teaching tools based on theorem provers can greatly reduce development costs. For example, the teaching tool proposed in this study only comprises 2,300 lines of Coq code, making it a more practical option for teaching. Moreover, it can also be adjusted for different teaching scenarios more conveniently.

The gap between formal proofs constructed in theorem provers as tools for assisting the learning of logic and the proofs of the logic systems in textbooks should be reduced so that students can focus on the logic system they are learning rather than the skills to use theorem provers. A qualified teaching tool based on theorem proving concepts shall have the following characteristics.

(1) The reasoning mode it supports is the same as that used in traditional logic textbooks.

(2) The same logic system as the one in the teaching content is used, and all operations are based on the reasoning rules of the system to the effect that the logic built into the theorem prover is exposed as little as possible. Furthermore, the logic terms and symbols in the textbooks are used instead of the newly created variable names in the theorem prover.

(3) Only a few simple instructions are needed to complete the proofs, thereby reducing the learning cost for students to use theorem provers. These proof instructions are highly automated to reduce the amount of code and help students complete strict proofs of more complex propositions.

To solve the above questions, we developed a prover with automated proof tactics in Coq for the axiomatic set theory ZFC to assist the teaching of the first-order logic and the axiomatic set theory. They formalized the axiomatic set theory ZFC based on the first-order logic in Coq using the logic system “sequent calculus” and applied the “notation” mechanism to provide concise symbols. We also developed several automated proof tactics to simplify the proof construction process, including the automated proof tactic “FOL_Tauto” for propositional logic. This tactic could transform a provable problem into a Boolean satisfiability problem in conjunctive normal form and call the Davis-Putnam-Logemann-Loveland (DPLL) satisfiability solver we implemented in Coq to automatically solve the problem. In this way, the fully automated processing of logical connectives is achieved. Tactics are developed for the reasoning rules for quantifiers, to support the simultaneous introduction and elimination of multi-level quantifiers; the tactic “peqsub_tac” is developed for the equal sign replacement rule to automatically replace the first-order logic terms in propositions. All the above tactics could automatically process α-equivalence and share the same forward reasoning mode as that in the textbooks. A simple and fast proving environment for set theory is constructed accoding to the above approach. It not only hid many details of Coq but also enabled proofs to be more in the textbook style and more automated. Students could construct proofs of the theorems in set theory more easily with this tool than by directly using existing theorem provers. Figure 1 shows an example of proof construction, and it suggests that the intersection of the two sets is bound to be a subset of one of them.

This tool achieved satisfactory results when it is introduced into the practical teaching of discrete mathematics. Students shall use the proposed prover to prove mathematical induction as one of the course projects. The practice shows that freshmen who are also new to first-order logic and the axiomatic set theory can complete the formal proof of the theorem after brief learning.

Some outstanding students can use this tool to further construct a proof of Peano arithmetic and prove the relevant properties of addition and multiplication after completing the learning of discrete mathematics. The above application results reflect the simplicity and effectiveness of the proposed tool.

```
Theorem intersect_subset1:
[[ ZF⊢∀u, ∀v, u∩v⊆u]].
Proof.
  "pose proof" Intersection_iff.
  universal instantiation H u v z.
  assert [[ZF⊢z∈u∩v→z∈u]] by FOL_Tauto.
  universal generalization H1 u v z.
  The conclusion is already proven.
Qed.
```

(a) Proof code

```
1 subgoal
H : [[ ZF⊢∀x, ∀y, ∀z, z∈x∩y↔z∈x∧ z∈y]]
H0 : [[ ZF⊢z∈u∩v ↔ z∈u∧z∈v]]
H1 : [[ ZF⊢z∈u∩v → z∈u]]
H2 : [[ ZF⊢∀u, ∀v, ∀z, z∈u∩v→z∈u]]
________________________________________(1/1)
[[ ZF⊢∀u, ∀v, u∩v⊆u]]
```

(b) Proof state bar in Coq before the last step

Figure 1 Example of proving environment in prover for set theory ZFC

Section 1 of this paper outlines the research background, including the research status of the interactive theorem prover Coq and logic teaching tools based on theorem provers. Section 2 describes the design framework of the prover for the set theory ZFC, and Section 3 presents the formalization of the axiomatic set theory ZFC. Section 4 demonstrates the implementation of automated proof tactics, and Section 5 shows the actual effectiveness of the proposed prover in the teaching of discrete mathematics. The final part summarizes the whole paper.

1 Research Background

1.1 Research status of logic teaching tools based on theorem provers

The main problem with using theorem provers to assist teaching lies in the large gap between theorem provers and the form of the logic systems found in textbooks, and how to narrow this gap has become the main line of research. One approach attempts to gradually bring the code proof of theorem provers closer to the proofs in textbooks in terms of the style of the proof. Böhne *et al*.[14] provided a proof tactic for each reasoning rule in Coq. Specifically, students are required to first complete the proof of a proposition in Coq and then write the corresponding first-order logic proposition to be proven in this case in the form of comments before and after each line of the proof code. Finally, they are required to sketch a textbook-style pen-and-paper proof that had the same structure as that of the proof constructed in Coq. Such a proof shared the same reverse reasoning mode with the one constructed in Coq. The practice indicated that students could complete the proofs and understand the correspondence between the proof constructed in Coq and the logic system. Avigad[11] used the logic built into Lean to teach first-order logic and set theory and enabled students to understand proof code and pen-and-paper proof by comparison.

Students could complete the proof as well after they systematically learned the usage of Lean. Another approach is to directly develop theorem-proving tools that are more in the textbook style. As early as the end of the 20th century, researchers at the University of Oxford developed Jape[9] and thereby provided a decent visual interface that could display proofs in the form of trees or boxes. Inspired by Jape, Kaliszyk *et al.* developed ProofWeb[10] based on Coq. It adopted a natural deduction system similar to Gentzen and provided a lightweight online Graphical User Interface (GUI). Students could also construct proofs from bottom to top by applying the tactics corresponding to each reasoning rule.

Most research on set theory in China focuses on the formalization of important theorems in set theory, such as C. T. Yang's theorem[15] and Tukey's lemma[16]. At present, the application of theorem proving in teaching has rarely been reported, and the related research available is still in the stage of discussion and exploration. Li[17] discussed the feasibility of using Isabelle to assist in the teaching of the course on mathematical logic offered to computer majors. Jiang *et al.*[18] compared the scheduling of courses related to logic and verification at the Technical University of Munich and that at universities in China. They called for more content related to theorem proving in the teaching of theoretical and practical courses for computer majors in China. To the best of our knowledge, this study takes the lead in applying interactive theorem proving to classroom teaching for beginners of mathematical logic in China.

1.2 Interactive theorem prover Coq

Coq is an interactive theorem-proving tool with powerful expressiveness and excellent scalability. As one of the mainstream proving assistants in the world, Coq has been widely used in the formal verification of mathematical logic, algorithms, and highly reliable software. The theoretical basis of Coq is the calculus of inductive constructions for providing users with inductive types that are more expressive than the inductive types in most functional programming languages. Users can define mathematical concepts, programming languages, and logic systems for formalization and construct proofs of relevant properties and theorems in Coq. During the proof construction, users start the proof construction with the instruction "Proof" and then apply a series of proof tactics to interact with Coq. The proof tactics involve decomposing the goal to be proven into several simpler sub-goals or directly proving the goal according to the current proof state and the proved theorems and axioms. The type-checking algorithm of the proof checker in Coq will mechanically check the types according to the nature of the calculus of inductive constructions. After the proof is completed, users run the instruction "Qed" to exit the proof. Table 1 lists some commonly used proof tactics in Coq and their purpose.

Coq allows users to combine existing proof tactics in to a new tactic in the language Ltac, thereby simplifying the proof construction process and improving the automation level of the proof. The automated proof tactics developed in this study are also implemented in the language Ltac. Students did not need to learn the proof instruction language of Coq. Instead, they are only required to use the seven automated proof tactics developed in this study. One of the proof tactics related to the separation axiom is not detailed in this paper since it is not engaged in the course project.

2 Design Framework of Prover for Axiomatic Set Theory

One thing should be clarified before introducing the implementation framework of the theorem prover for ZFC. It is that this theorem prover is developed for specific teaching scenarios of mathematical logic. The object of teaching is students who have mastered propositional logic, and the focus of teaching at the current stage is the four reasoning rules for quantifiers in the logic system "sequent calculus" and the relationship between the axiomatic set theory ZFC and

Table 1 Commonly used proof tactics in Coq

Proof tactic	Purpose
intros	Introducing the conditions in the goal to be proven
destruct	Discussing by categories according to constructors
induction	Conducting structural induction according to constructors
apply	Applying a hypothesis or proven theorem to the goal to be proven
eapply	No need to specify variables during application and delaying the instantiation of variables
pose proof	Introducing a proven theorem into the condition
assert	Declaring a new goal to be proven
assert... by...	Declaring a new goal to be proven and resolving it directly by the tactic after "by"
rewrite	Rewriting according to the equation
congruence	Automated proof tactic for equation rewriting
reflexivity	Automatically comparing whether the two sides of the equal sign in the goal to be proven are the same
tauto	Automated proof tactic by intuitionistic logic

everyday mathematics. Therefore, we used the same logic system "sequent calculus" as the one in textbooks when we formalized the axiomatic set theory ZFC. When developing the proof tactics, we integrated all the proof constructions related to propositional logic into the tactic "FOL_Tauto" completely automatically and provided a separate tactic for each inference rule for quantifiers. Furthermore, we introduced the symbols of binary intersection $\cap$, binary union $\cup$, and singleton $\{\}$ as function symbols in first-order logic. They also proposed the empty set $\varnothing$ as a constant symbol. The set theory ZFC in textbooks usually does not contain these symbols, but the introduction of these symbols will not change the reliability of the entire reasoning system. Specific proof of this statement is available in standard textbooks[19]. Although the proposed theorem-proving tool is implemented in Coq, this study is designed to provide a teaching-oriented theorem-proving tool rather than a verified prover. Therefore, the verification of the correctness of the solvers and other modules involved is not repeated in this paper.

The proposed prover is designed into two parts, namely, the formalization of the axiomatic set theory ZFC and the interactive theorem proving environment, and both parts are implemented in Coq. The specific details of the design are presented in Figure 2.

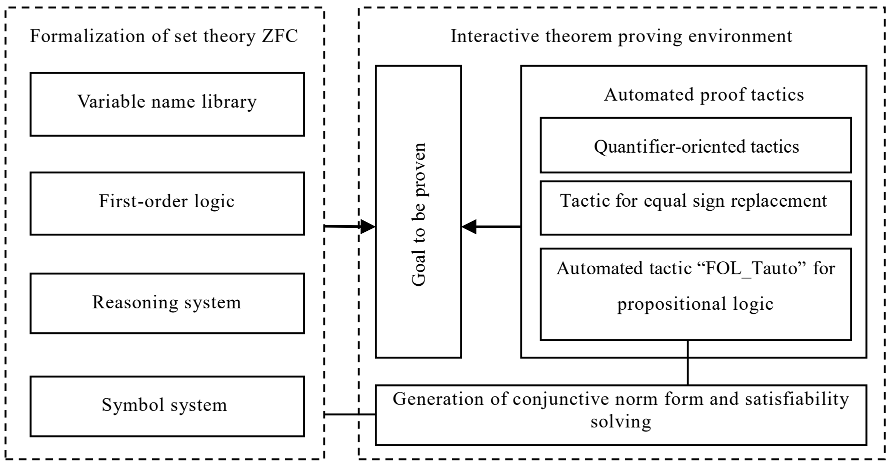

Formalization of set theory ZFC
Variable name library
First-order logic
Reasoning system
Symbol system
Interactive theorem proving environment
Goal to be proven
Automated proof tactics
Quantifier-oriented tactics
Tactic for equal sign replacement
Automated tactic "FOL_Tauto" for propositional logic
Generation of conjunctive norm form and satisfiability solving

Figure 2 Design framework of the prover for the set theory ZFC

Part 1 involves the formalization-oriented definition of the relevant concepts of the axiomatic set theory ZFC in Coq and the construction of a reliable reasoning system. This part can be described from the following four aspects.

(1) Explicit variable name library StringName

Rather than using the variable name system built into Coq, we instead developed an explicit variable name library called StringName specifically for the reasoning system based on the existing string library called String in Coq. This library provides users with better control over variable names in the sense that it allows users to define methods for introducing new variable names and comparing variable names. The control is beneficial for subsequent substitutions, α-equivalence, and other grammatical operations. Furthermore, starting the formalization of the logic system from the string-based abstract syntax tree can also avoid exposing the logic built into Coq in the proving environment. As shown in Figure 1(b), the proposed proving environment only displays a series of conditions for provable propositions, and conditions such as x:String will thus not be displayed. Moreover, users no longer need to call the proof tactic "intros" to introduce conditions for each variable at the beginning of the proof, thereby maintaining a high degree of consistency with textbook-style proofs.

(2) First-order logic proposition

Since the search object of the first-order logic is first-order logic propositions, propositions and terms shall be formalized in Coq.

(3) Reasoning system for the set theory ZFC

The logic system "sequent calculus" in textbooks is adopted as the reasoning system, and each step in the proof construction process is composed of multiple premises and one conclusion. Since the reasoning system only involves grammatical structures, we only formalized provable property "derivable" in this study, rather than formalizing the semantics of the system.

(4) Symbol system

The proof code of theorem provers is usually poorly readable and not easy for students to understand. Nevertheless, the "notation" mechanism of Coq can effectively solve this problem. For example, the formalization-oriented definition of material implication is `PImpl P Q`. If Notation `"P1→P2":=(PImpl P1 P2)` is defined, it can be abbreviated as "`P→Q`" in subsequent proofs.

Part 2 is the interactive theorem-proving environment, where the theorem to be proven consists of the previously formalized propositions. Users decide how to apply reasoning rules by the proposed automated proof tactics. The tactics developed in this study all adopted the same forward reasoning mode as in textbooks. Every time a user successfully applies a tactic, the new reasoning relationship generated by the reasoning rule will be added to the proving environment according to the current conditions. When the conclusion can be obtained by propositional logic reasoning on the existing premises, the user can call the tactic "The conclusion is already proven" (alias of "FOL_Tauto" to complete the proof) to finally complete the proof of the theorem. Users do not need any proof tactics built into Coq other than "pose proof". The specific implementation of automated proof tactics will be presented in Section 4.

3 Formalization of Set Theory ZFC

3.1 Explicit variable name library

We first implemented the naming system module NAME_SYSTEM_EXT by applying the module mechanism in Coq. This module accepts the parameters shown in Table 2. These parameters can be used to further define other functions of Coq and construct proofs of their related properties.A function named `list_new_name` is defined for introducing new names according to the name list, and the name next to the one with the largest name in the list is returned.

```
Definition list_new_name (xs: list t):=next_name (list_max xs).
```

In practical use, a naming system module of the type specified by the user can be obtained by instantiating the various parameters in Table 2 only and proving that the instantiated types and functions satisfy the properties in Table 2. Parameter `t` is instantiated into the type string in the library String, and the lexicographical order of strings is adopted for parameters `max` and `le`. Moreover, `next_name` is instantiated to adding the suffix "′" after the input string, and `default` is instantiated to the string "`x`". In this way, we obtained the string module `StringName`. Proving that the above types satisfy the properties in Table 2 is not the key to this paper and is thus not presented here.

Table 2 Parameters of module NAME_SYSTEM_EXT

Parameters and properties to be satisfied	Description
`t: Type`	Name type
`max: t→t→t`	Functions that compare and return the larger name
`next_name: t→t`	Returning the next name entered
`le: t→t→Prop`	Binary relation of being smaller than or equal to
`default: t`	Default name
`eq_dec: ∀v1 v2: t, {v1=v2}+{v1<>v2}`	Names are comparable to the effect that they are either identical or different
`Transitive le`	The relation of being smaller than or equal to is transitive
`∀u v, le u v→u≠next_name v`	The name next to any name is larger than itself
`∀v1 v2, le v1 (max v1 v2)`	Correctness 1 of function `max`, larger than the first parameter
`∀v1 v2, le v2 (max v1 v2)`	Correctness 2 of function `max`, larger than the second parameter
`∀v1 v2, max v1 v2=max v2 v1`	Commutative property of function `max`
`∀v1 v2 v3, max v1 (max v2 v3)=max (max v1 v2) v3`	Associative property of function `max`

In subsequent development, library `StringName` would be referenced under different names for different purposes. In the reasoning system, we let `V:=StringName` and used `V.t` as the type of variable names in the proposition. In the DPLL solver, `PV:=StringName` is assumed, and `ident:=PV.t` is used as the type of the propositional variables. Although they are both string types in essence, different names are used to avoid confusion.

Furthermore, we assigned variable names built into Coq to commonly used strings, such as

```
Definition x:=EVAL "x"% string.
Definition x0:=EVAL (V.next_name x).
```

In this way, letters, such as `x`, instead of quoted strings would be directly displayed in the proving environment, and the names next to `x` could also be displayed as `x0,x1,x2,...` in sequence, as shown in Figure 1.

3.2 Formalization of propositions and their related properties

The basic research object of first-order logic is first-order logic terms. In the axiomatic set theory ZFC, a term represents a set, and its inductive definition is as follows:

$$t ::= \varnothing \mid x \mid \{t\} \mid t_1 \cap t_2 \mid t_1 \cup t_2 \tag{1}$$

where $\varnothing$ is the empty set constant; x is a variable; $\{t\}$ is a singleton; and $\cap$ and $\cup$ are binary intersection and binary union, respectively. The keyword "Inductive" of Coq is used to obtain the formalization-oriented definition `term` of the abovementioned terms as follows:

```
Inductive term :=
| var (v:V.t) | empty_set | singleton (x:term) | union (x y: term) |
    intersection (x y:term)
```

where constructor `empty_set` corresponds to the empty set constant; constructor `var` corresponds to the variable represented by the string; `singleton` corresponds to a singleton; and `union` and `intersection` correspond to binary union and binary intersection, respectively. Afterward, the "notation" mechanism is used to obtain the following abbreviations, thereby ensuring that the symbols in Coq are consistent with those in Eq. (1). The newly defined propositional symbols are only valid within double brackets "`[[]]`" so that they would not conflict with the ones built into Coq.

```
Notation "[[e]]" := e (at level0, e custom set at level99).
Notation "∅" := empty_set (in custom set at level5, no associativity).
Notation "x ∩ y" :=(intersection x y)(in custom set at level11, left
    associativity).
```

The other abbreviations are attained in a similar manner.

The inductive definition of the proposition is as follows.

$$\begin{aligned} P ::= & t_1 = t_2 \mid t_1 \in t_2 \mid \top \mid \bot \mid \neg P \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \to P_2 \mid P_1 \leftrightarrow P_2 \mid \\ & \forall x, P \mid \exists x, P \end{aligned} \tag{2}$$

where $t_1 = t_2$ and $t_1 \in t_2$ are atomic propositions and represent the equal and belonging relationships between two sets, respectively; $\top$ and $\bot$ are true and false propositions, respectively; the other constructors correspond to negation, conjunction, disjunction, material implication, if and only if, universal quantifier, and existential quantifier, respectively. The formalization-oriented definition *prop* of the proposition is as follows.

```
Inductive prop: Type :=
| PEq(t1 t2: term) | PRel(t1 t2: term) | PFalse | PTrue
| PNot(P: prop) | PAnd(P Q: prop) | POr(P Q: prop) | PImpl(P Q: prop) | PIff(P
    Q: prop)
| PForall(x: V.t)(P: prop) | PExists(x: V.t)(P:prop).
```

In Coq, the "notation" mechanism can also be used to provide the same notations as those in Eq. (2).

Next, a method of replacing the free variables in a proposition would be provided. The substitution task σ is defined as a list composed of ordered pairs of variable names and terms, and the term by which each variable would be substituted is recorded. The result of substituting proposition P with σ could be denoted as $P[\sigma]$:

```
∅[σ]::=∅                          x[·]::=x
x[x↦t;;...]::=t                   x[y↦t;;σ′]::=x[σ′]
(t₁∩t₂)[σ]::=t₁[σ]∩t₂[σ]
⊤[σ]::=⊤                          (t₁=t₂)[σ]::=t₁[σ]=t₂[σ]
(P₁∧P₂)[σ]::=P₁[σ]∧P₂[σ]
(∀x, P)[σ]::=∀x, P[σ]             if x∉V(σ)
(∀x, P)[σ]::=∀u, P[x↦u;σ]         if x∈V(σ)
```

When terms are replaced, the empty set constant remains unchanged before and after the substitution. Variable x will be compared from the beginning with the terms in the replacement target until the term t corresponding to x is found. Otherwise, it will not be replaced. The sub-terms of the singleton and binary intersection will be replaced, separately. The proposition is replaced similarly. The replacement of quantifiers deserves special attention. Whether the variable bound by the quantifier appears in the substitution task needs to be checked. If the answer is no, the replacement of sub-propositions should be continued. Otherwise, a new variable u that is not present in the current environment needs to be introduced, and x in the sub-proposition P should be replaced by u before x↦u is added to the substitution task to avoid conflicts. The key quantifier part of the formalization-oriented definition prop_sub is listed below. Function sub_task_occur can be applied to determine the number of times a name appears in the replacement target. Moreover, function new_var is responsible for introducing new variable names, and its definition is based on function list_new_name from Section 3.1.

```
Fixpoint prop_sub (st: subst_task) (d: prop): prop :=
match d with
    | [[ ∀x,P]]   => match subst_task_occur x st with
            | 0 => PForall x (prop_sub st P)
            | _ => let x′ := new_var P st  in
                    PForall x′ (prop_sub (cons (x, var x′)  st) P)
            end
    | ...
    end
```

Another important grammatical property of propositions is α-equivalence describing the renaming of quantifier-bound variables, such as ∀x, x=x and ∀y, y=y. Propositions characterized by α-equivalence have the same semantics. The definition of α-equivalence is usually described as multiple renames of quantifier-bound variables, that is $\forall x, P =_\alpha \forall y, P[x \mapsto y]$. Nevertheless, the replacement-based definition is not conducive to automation. Therefore, we proposed an inductive definition that came with an environment and did not rely on substitution.

An environment θ is a list of 3-tuples:

$$\theta ::= [\cdot] \mid (u, v, b);;\theta' \tag{3}$$

where u and v are variable names that record the correspondence between the variables bound by quantifiers in two propositions and b is a Boolean value where true indicates that the correspondence is currently valid while false indicates that the correspondence is invalid since it has been overwritten by a new correspondence.

Environment θ records the path by which two propositions are checked from top to bottom for quantifier correspondence. The environment-bearing α-equivalence of the first-order logic terms is denoted as $=_\theta$, and some of its inductive definitions are presented as follows:

$$A_\varnothing\frac{}{\varnothing =_\theta \varnothing} \qquad \mathit{ABind}\frac{(u, v, \texttt{true}) \in \theta}{u =_\theta v} \qquad \mathit{AFree}\frac{s \notin V(\theta)}{s =_\theta s} \qquad A_\cap\frac{t_1 =_\theta t_2 \quad t_3 =_\theta t_4}{t_1 \cap t_3 =_\theta t_2 \cap t_4}$$

Specifically, rule $A_\varnothing$ specifies that empty sets are only equivalent to themselves in any environment θ. Rule *ABind* deals with bound variables, and u and v are equivalent if they have valid correspondence in the environment. Rule *AFree* deals with free variables. If variable s is not in the environment, it appears freely and can only be equivalent to itself. The binary intersection determines whether the sub-terms are equivalent, and the rule also applies to the case of singleton and binary union.

The α-equivalence property of terms can then be used to obtain the environment-bearing α-equivalence of the proposition. The following are some typical rules:

$$A_{\in}\,\frac{t_1 =_\theta t_2 \quad t_3 =_\theta t_4}{t_1 \in t_3 =_\theta t_2 \in t_4} \qquad A_{\forall}\,\frac{P =_{(x,y,\texttt{true});\theta\backslash(x,y)} Q}{\forall x, P =_\theta \forall y, Q}$$

$$A_{\top}\,\frac{}{\top =_\theta \top} \qquad A_{\wedge}\,\frac{P_1 =_\theta Q_1 \quad P_2 =_\theta Q_2}{P_1 \wedge P_2 =_\theta Q_1 \wedge Q_2}$$

The rule corresponding to quantifiers is the only one that modifies the environment. During the top-down checking, x in P corresponds to y in Q if $\forall x$, P and $\forall y$, Q are present. Therefore, $(x, y, \texttt{true})$ needs to be recorded into the environment before the sub-propositions are checked, The record with x on the left or y on the right in θ (equivalent to the term corresponding to the outer quantifier) is no longer valid in the sub-proposition P and thus needs to be set to `false`. The above processing is denoted as $\theta\backslash(x, y)$.

When the outermost environment is empty, the definition of the original α-equivalence can be naturally obtained as $P =_\alpha Q := P =_{[\cdot]} Q$. This process is decidable, and the process of determining α-equivalence is formalized into a recursive function `alpha_eq` that returns Boolean values for automated calculation in Coq.

```
Fixpoint alpha_eq(l:binder_list)(P Q:prop):bool:=
    match P,Q with
        | [[ t1∈t2]], [[ t3∈t4]] => term_alpha_eq l t1 t3 && term_alpha_eq l
        t2 t4
        | [[ P1∧P2]], [[ Q1∧Q2]] => alpha_eq l P1 Q1 && alpha_eq l P2 Q2
        | [[ ∀x, P1]],  [[ ∀y, Q1]]  => alpha_eq ((x,y,true)::(update x y l))  P1 Q1
        | ...
        | _,_ => false
    end
Definition aeq(P Q:prop):bool:= alpha_eq nil P Q
```

3.3 Formalization of reasoning systems

The logic system "sequent calculus" is adopted to serve as the reasoning system. Each step in the proof construction is a sequence of propositions $\varphi_1\varphi_2\cdots\varphi_n\ \varphi$, with the first n propositions as the premises, and the last proposition φ as the conclusion. The sequence is recorded as $\varphi_1\varphi_2\cdots\varphi_n \vdash \varphi$. Γ would be used to represent the sequence of propositions, and φ would be utilized to represent a proposition hereinafter. $\Gamma\varphi$ represents a new sequence of propositions obtained by adding proposition φ after the sequence of propositions Γ.. The reasoning rule describes a step in the process of proof construction that accepts multiple conditions and obtains a new sequence of formulas as its conclusion. The reasoning rules adopted in this study are all from logic textbooks used in actual teaching[19]. Some of them are selectively listed as follows, including the one related to logical connectives.

$$\wedge_{\textit{Intro}}\,\frac{\Gamma\vdash P\,\Gamma\vdash Q}{\Gamma\vdash P\wedge Q} \qquad \neg_{\textit{Contra}}\,\frac{\Gamma\neg P\vdash Q\,\Gamma\neg P\vdash\neg Q}{\Gamma\vdash P} \qquad \textit{Modus}\,\frac{\Gamma\vdash P\,\Gamma\vdash P\to Q}{\Gamma\vdash Q}$$

$$\textit{Assu}\,\frac{}{\Gamma P\vdash P} \qquad \textit{Weaken}\,\frac{\forall\varphi,\varphi\in\Gamma\to\varphi\in\Gamma'\,\Gamma\vdash P}{\Gamma'\vdash P} \qquad \alpha_{\textit{Congruence}}\,\frac{P =_\alpha Q\,\Gamma\vdash P}{\Gamma\vdash Q}$$

In the four reasoning rules for quantifiers, $FV(\varphi)$ is the set constituted of all the freely occurring variables in proposition φ:

$$\forall_{\textit{Elim}}\,\frac{\Gamma\vdash\forall x, P}{\Gamma\vdash P[x\mapsto t]} \qquad \exists_{\textit{Intro}}\,\frac{\Gamma\vdash P[x\mapsto t]}{\Gamma\vdash\exists x, P}$$

$$\forall_{Intro}\frac{\forall\varphi\in\Gamma,x\notin FV(\varphi)\,\Gamma\vdash P}{\Gamma\vdash\forall x,P}\qquad \exists_{Elim}\frac{\forall\varphi\in\Gamma,x\notin FV(\varphi)x\notin FV(Q)\,\Gamma\,P\vdash Q}{\Gamma\exists x,P\vdash Q}$$

The reasoning rule related to the equal sign is expressed as follows.

$$=_{Refl}\frac{}{\vdash t=t}\qquad =_{Subst}\frac{}{\vdash t=t'\to P[x\mapsto t]\to P[x\mapsto t']}$$

The theorems in the set theory ZFC are denoted as follows.

$$Empty\frac{}{\vdash\forall x,x\notin\varnothing}\qquad Extensionality\frac{}{\vdash\forall xy,(\forall z,z\in x\leftrightarrow z\in y)\leftrightarrow x=y}$$

$$Infinity\frac{}{\vdash\exists x,\varnothing\in x\wedge\forall y,(y\in x\to y\cup\{y\}\in x)}$$

$$Separation\frac{x\notin FV(P)\quad y\notin FV(P)}{\vdash\forall x,\exists y,\forall z,z\in y\leftrightarrow z\in x\wedge P}$$

In addition, the axioms of set-related symbols are expressed as follows.

$$Union\frac{}{\vdash\forall xyz,z\in x\cup y\leftrightarrow z\in x\vee z\in y}$$

$$Singleton\frac{}{\vdash\forall xy,y\in\{x\}\leftrightarrow y=x}$$

$$Intersection\frac{}{\vdash\forall xyz,z\in x\cap y\leftrightarrow z\in x\wedge z\in y}$$

The sequences of formulas composed of premises and conclusions are formalized separately. The conclusion proposition is of the type *prop* mentioned in the previous section, and the premises are defined to be of the type `context:=prop→Prop`, namely, a set of propositions. Then, empty premises without any propositions could be further defined and abbreviated as ZF, and the method of adding new propositions after the premises can be defined and abbreviated as $\Gamma;;\varphi$.

```
Definition empty_context: context:=fun _=>False
Notation "`ZF'":=empty_context                          (in custom set at level20)
Notation "Phi;;x":=(Union _ Phi (Singleton_x))      (in custom set at level31, left
     associativity)
```

The keyword "Inductive" can be used to inductively define the above reasoning rules in a formalization fashion to obtain provable relationship `derivable` of type `context → prop → Prop`. Moreover, "`derivable` Γ φ" formally represents that first-order logic proposition φ can be derived from premise Γ according to the reasoning rules. `Derivable` is abbreviated as $\vdash$ in Coq.

```
Notation "Phi ⊢ P":=(derivable Phi P)    (in custom set at level41, no associativity)
```

Next, the formalization-oriented definition of *derivable* would be presented with rule $\forall_{\mathrm{Intro}}$, the axiom of extensionality, and that of infinity.

(1) Rule $\forall_{\mathrm{Intro}}$ can be denoted in the manner of formalization as follows.

```
| PAnd_intros: forall Phi P Q,
    derivable Phi P→
    derivable Phi Q→
    derivable Phi [[P∧Q]]
```

If one of the premises in Coq is that P and Q can be derived from the same premise Phi during the proof construction, a claim can be further made that [[P∧Q]] can be derived from premise Phi as well.

(2) The axiom of extensionality can be expressed in the manner of formalization as follows.

```
| Extensionality: derivable empty_context[[∀x, ∀y, (∀z, z∈x ↔z∈y)↔x = y]]
```

The formalization of the axiom of extensionality has the same form as that of the axioms in textbooks. We directly took the proposition in the axiom as the conclusion, where variable names such as x and y are strings defined in Section 3.1. Therefore, we no longer needed to introduce the variables in Coq by calling "forall" during formalization. During proof construction, the axiom of extensionality is introduced by tactic "pose proof Extensibility", and the new condition [[ZF⊢∀x, ∀y, (∀z, z∈x↔z∈y) ↔x=y]] would be directly added into the proof state in Coq. Afterward, the proof tactic for eliminating universal quantifiers can be applied to instantiate x and y into practically needed terms for further proof.

(3) When the axiom of infinity is formalized, we expected that the proposition that the target set is an inductive set could be simplified into the form of a predicate.

The proposition "is_inductive_def" with parameter t is defined as follows:

```
Definition is_inductive_def(t:term) :=[[(∅∈x∧∀y, (y∈x→y∪{y}∈x)) [x↦t]]]
```

This definition represents that the empty set is in set t and the successor of any element in t is also in set t. The substitution operation x↦t is introduced to rename variable y bound by the universal quantifier in the original proposition to another variable in the manner of substitution when parameter t is variable y, thereby avoiding the incorrect proposition "∀ y, y∈y → y∪{y}∈ y".

Subsequently, the "notation" mechanism is employed to abbreviate is_inductive_def t as is_inductive t in propositions:

```
Notation "`is_inductive′ t":= (is_inductive_def t)(in a customset at
    level20, t1 at level15, no associativity).
```

The axiom of infinity can be expressed in the manner of formalization as follows:

```
| Infinity: derivable empty_context [[∃x, is_inductive x]]
```

When a proof is constructed, the tactic "pose proof Infinity" can be called to introduce the condition [[ZF⊢∃x, is_inductive x]].

4 Automated Proof Tactics

This section presents the automated proof tactics we provide for users. In the proving environment, the proof tactics we developed and the strategy "pose proof" built into Coq will be sufficient for users to construct proofs. Instead of modifying the goal to be proven, these tactics only add new conditions to the proving environment and are thus in line with the forward reasoning mode in textbooks. Table 3 lists all the proof tactics students need to construct proofs and their purpose. Section 5.1 presents the specific application effects of those tactics, with a detailed comparison among proofs constructed in Coq and the corresponding formal proofs in textbooks.

4.1 Automated proof tactic "FOL_Tauto" for propositional logic

First-order logic is usually undecidable, which means that finding an effective algorithm to decide whether a proposition can be derived from a group of propositions is impossible.

Table 3 All proof tactics students need and their purpose

Tactic	Purpose
`assert... by FOL_Tauto`	Judging whether the derivation relationship claimed by the user can be obtained by applying propositional logic on the existing conditions
`The conclusion is already proven.`	An alias for "FOL_Tauto" used to prove the last step
`universal instantiation`	Applying rule $\forall_{Elim}$ for eliminating universal quantifiers to instantiate multiple universal quantifiers before specifying a conclusion
`universal generalization`	Applying rule $\forall_{Intro}$ for introducing universal quantifiers to introduce multiple universal quantifiers before specifying a conclusion
`existential instantiation`	Applying rule $\exists_{Elim}$ for existential quantifiers to introduce multiple existential quantifiers before specifying premises
`existential generalization`	Applying rule $\exists_{Intro}$ for introducing existential quantifiers to introduce multiple existential quantifiers before specifying a conclusion
`peq_sub_tac`	Introducing a new equal sign substitution relationship
`pose proof` (built in Coq)	Introducing a proven theorem

Nevertheless, first-order logic can be converted into decidable propositional logic by abstracting first-order logic propositions containing quantifiers into propositional variables. Then, the problem can be transformed into a satisfiability-solving problem by leveraging the equivalence between the validity of a proposition and the unsatisfiability of its negative proposition. A solver can then be used to automatically process logical connectives. Drawing on this idea, we developed the automated proof tactic "FOL_Tauto" for propositional logic. This section will explain the implementation of this tactic in detail. Remarkably, the validity of the solver has not been verified although it is implemented in Coq.

(1) Preprocessing of propositional variables

Before solving, the atomic propositions and propositions containing quantifiers in the first-order logic shall be abstracted into propositional variables. Moreover, they shall be abstracted into the same propositional variable since propositions characterized by α-equivalence have the same semantics. Specifically, we defined a simple key-value pair structure `prop_table` in Coq to record the variable corresponding to a proposition. They then used a list to store pair of propositions and corresponding `ident` (i.e., the type of the propositional variable, which is already mentioned in Section 3.1). Comparisons would be performed from the start to determine whether the input proposition and the current key are α-equivalent when function `prop_look_up` is called for searching. The corresponding variable would be returned if the answer is yes. Otherwise, `None` would be returned.

Then, we defined the type `sprop` of propositions of propositional logic.

```
Inductive sprop: Type :=
| SId (x: ident) | SFalse | STrue | SNot (P: sprop) | SAnd (P Q: sprop) | SOr
    (P Q: sprop) | SImpl (P Q: sprop)
```

where constructor `SId` is the propositional variable of the atom, and the other constructors correspond to true and false propositions and conventional logical connectives whose semantics are the same as those of the connectives in the first-order logic proposition `prop`. For the convenience of writing, `sprop` would be abbreviated into the same notations as those in Eq. (2). Remarkably, it would be expanded into the conjunction of two material implications during the conversion due to the absence of connective $\leftrightarrow$ in `sprop`.

The recursive function `sprop_gen` for conversion from proposition `prop` of the first-order logic to proposition sprop_gen of propositional logic is easy to define. This function takes three parameters, namely, proposition P to be converted, prop_table KV recording assigned propositions and variables, and available propositional variable string s. In addition, it returns three values, i.e., the conversion result P' of P, the modified prop_table KV', and the new available propositional variable s'. The definition of sprop_gen (abbreviated as sprop) is as follows, with parameter s omitted in non-critical cases and similar operations in unlisted cases.

```
sprop(P, KV) ::= KV(P), KV   if prop_look_up (P, KV)≠None
sprop(⊤, KV) ::= STrue, KV
sprop(t₁∈t₂, KV, s) ::= SId s, (t₁∈t₂↦s;;KV), next_name(s)
sprop(∀x.P, KV, s) ::= SId s, (∀x, P↦s;;KV), next_name(s)
sprop(P∧Q, KV, s) ::= let P′, KV′, s′ := sprop(P, KV, s) in
                        let Q′, KV″, s″ := sprop(Q, KV′, s′) in
                          P′∧Q′, KV″
```

If a propositional variable has already been assigned to the proposition equivalent to P in KV, sprop_gen directly returns this propositional variable. Otherwise, classification should be performed in categories according to the structure of the proposition to be converted. A true proposition $\top$ will be directly converted to Strue accordingly, saving the need to modify KV and s. Regarding atomic propositions or first-order logic propositions starting with quantifiers, the current proposition will be abstracted into the propositional variable s, and this correspondence will be recorded in KV. Then, the name next to s will be taken as a new available propositional variable. The processing of binary logical connectives starts with the conversion of the left sub-proposition. Then, the right sub-proposition will be converted based on the converted KV' and available variable s'. Finally, the corresponding logical connectives are used to connect the respective conversion results of the two sub-propositions.

(2) Generation of conjunctive normal forms

Modern satisfiability solvers convert propositions into conjunctive normal form and then solve them for higher efficiency. A conjunctive normal form can be seen as the conjunction of a series of disjunctive clauses, and each disjunctive clause is the conjunction of a series of propositional variables or their negative forms (called “literal”). The following formulas present the formalization-oriented definitions of clause type clause and type CNF of conjunctive normal forms. The disjunctive clauses are represented by a list of pairs constituted of Boolean values and propositional variables. A Boolean value of true represents the propositional variable, while that of false represents the negative form of the propositional variable. The CNF of conjunctive normal forms is represented by a list of type clause of disjunctive clauses.

```
Definition clause:=list (bool*ident).
Definition CNF:=list clause.
```

Any proposition can be converted into a logically equivalent conjunctive normal form when new propositional variables are not introduced. However, such conversion will lead to an exponential growth in the number of logical connectives. When new variables are introduced, a conjunctive normal form with linear growth of logic connectives can be generated under the condition of unchanged satisfiability rather than logical equivalence. The simplest method of this kind is the Tseitin algorithm[20], which introduces a new variable for each sub-proposition. In this study, we present cnf_gen (abbreviated as cnf), i.e., a method of generating the CNF of conjunctive normal forms without introducing excessive new variables. This function adopts a successor style and took three parameters, namely, the sprop-type proposition P to be generated, available propositional variable s, and the generated conjunctive normal form φ.

It returns two values; the new conjunctive normal form obtained by the conjunction of the conversion result of P and φ and the new available propositional variable s$'$. The second parameter s is omitted in non-critical cases.

```
cnf(SIdx, s, $\varphi$)::=(x)$\wedge\varphi$, s
cnf($\top$, s, $\varphi$)::=$\varphi$, s
cnf($\bot$, s, $\varphi$)::=(impossible)$\wedge$($\neg$impossible), s
cnf(P$\wedge$Q, s, $\varphi$)::=let $\varphi'$, s:=cnf(P, s, $\varphi$) in cnf(Q, s, $\varphi'$)
cnf(P$\vee$Q, s, $\varphi$)::=let $\tau$, $\varphi'$, s$'$:=clause(P, s, nil, $\varphi$) in
                    let $\tau'$, $\varphi''$, s$''$:=clause(Q, s$'$, $\tau$, $\varphi'$) in $\tau\wedge\varphi'$, s
cnf_gen(P$\rightarrow$Q, s, $\varphi$) ::= let $\tau$, $\varphi'$, s$'$:=neg_clause(P, s, nil, $\varphi$) in
                       let $\tau'$, $\varphi''$, s$''$:=clause(Q, s$'$, $\tau$, $\varphi'$) in $\tau\wedge\varphi''$, s
cnf_gen($\neg$P)::=let $\tau$, $\varphi'$, s$'$:=neg_clause(P, next_name(s), s, $\neg$n$\wedge\varphi$) in $\tau\wedge\varphi'$, s$'$
```

If the proposition P to be generated is a single propositional variable, the propositional variable is directly treated as a complete disjunctive clause for its further conjunction with the original φ. If the proposition is true, the conjunction of any proposition and the true proposition is logically equivalent to the proposition itself, and the original conjunctive normal form is retained. If it is a false proposition, its conjunction with φ is unsatisfiable. In this case, an unsatisfiable conjunctive normal form is directly constructed as the generated result, and *impossible* is the name of the propositional variable. In the case of P$\wedge$Q, it could be decomposed into two sub-conjunctive normal forms. The CNF of P is generated before that of Q is obtained. Then, the conjunctions of the two conjunctive normal forms with φ are obtained. In comparison, disjunction, material implication, and negation are more complicated. In such cases, simple recursion according to the structure of the proposition no longer worked. Instead, the clause method shall be used to generate disjunctive clauses and then add them to the conjunctive normal forms.

Function clause takes four parameters, namely, the sprop-type proposition P to be generated, the available proposition variable s, the generated clause, and the generated conjunctive normal form φ. It returns three values, namely, the disjunctive clause τ containing the conversion result of P, the modified conjunctive normal form φ', and the new available propositional variable s$'$. The definition of clause is given below, with parameter s omitted in non-critical cases.

```
clause(SIdx, $\tau$, $\varphi$)::=x$\vee\tau$, $\varphi$
clause($\bot$, $\tau$, $\varphi$)::=$\tau$, $\varphi$
clause($\top$, $\tau$, $\varphi$)::=(tauto$\vee\neg$tauto, $\varphi$)
clause(P$\wedge$Q, s, $\tau$, $\varphi$)::=let $\tau'$, $\varphi'$, s$'$:=clause(P,next_name(s), $\neg$s, $\varphi$) in
                         let $\tau''$, $\varphi''$, s$''$:=clause(Q, s$'$, $\neg$s, $\tau'\wedge\varphi'$) in
                            s$\vee\tau$, $\tau''\wedge\varphi''$
clause(P$\vee$Q, $\tau$, $\varphi$)::=let $\tau'$, $\varphi'$, s$'$:=clause(P, s, $\tau$, $\varphi$) in
                      let $\tau''$, $\varphi''$, s$''$:=clause(Q, s$'$, $\tau'$, $\varphi'$) in $\tau''$, $\varphi''$
clause(P$\rightarrow$Q, $\tau$, $\varphi$)::=let $\tau'$, $\varphi'$, s$'$:=neg_clause(P, s, $\tau$, $\varphi$) in
                      let $\tau''$, $\varphi''$, s$''$:=clause(Q, s$'$, $\tau'$, $\varphi'$) in $\tau''$, $\varphi''$
clause($\neg$P, $\tau$, $\varphi$)::=neg_clause(P, $\tau$, $\varphi$)
```

Propositional variables are directly added into existing disjunctive clauses in the manner of disjunction. The disjunction of a false proposition and any disjunctive clause is equivalent to the proposition itself, so it do not change the disjunctive clause. The disjunction of a true proposition and any disjunctive clause is logically equivalent to the true proposition. Therefore, a disjunctive clause that is satisfiable in any case is constructed to replace τ. For disjunctive proposition P∨Q, disjunctive clauses could be generated for the two sub-propositions separately. Then, further disjunction could be performed to obtain a large clause. In contrast, material implications can be converted into an equivalent form of ¬P∨Q, and all the other operations are the same as those conducted in the case of disjunction, except for generating the negative proposition (neg_clause) for P.

More complex is the case for the conjunction P∧Q. We would introduce a new proposition variable s to represent P∧Q, similarly to the Tseitin algorithm. The basic idea is to add the propositional variable s only into the previously generated disjunctive clause τ and to add the clause of P and Q into the outer conjunctive normal form by direct conjunction. Finally, the following CNF form is obtained:

```
(s∨τ)∧(¬s∨clause(P))∧(¬s∨clause(Q))∧φ
```

The above equation is equivalent to the following form:

```
(s∨τ)∧(s→clause(P))∧(s→clause(Q))∧φ
```

The structural induction of the first parameter of clause is performed, suggesting that the above form is equally satisfiable as $((P\wedge Q)\vee\tau)\wedge\varphi$.

The Neg_clause method could be employed to generate a clause for the negative proposition of P, which has a form dual to clause. For example, ¬(P∧Q) and ¬P∨¬Q are equivalent. Therefore, when the negative proposition of the conjunctive clause P∧Q is generated, only separate conversion is needed before disjunction, and similar operations are applied in other cases.

```
neg_clause(P∧Q, τ, φ)::=let τ', φ', s':=neg_clause(P, s, τ, φ) in
                        let τ'', φ'', s'':=neg_clause(Q, s', τ', φ') in
                         τ'', φ''
```

For example, proposition P→(Q∧R) is equivalent to ¬P∨(Q∧R). Therefore, a disjunctive clause is generated for the negative proposition of P to obtain ¬P, i.e., the first part of the disjunctive clause. Then, clause(Q∧R,¬P,nil) is calculated according to the definition of *cnf*. Seen from the above discussion, a new variable s would then be introduced to replace Q∧R. The conjunction of s and ¬P is obtained, and separate disjunctive clauses would be generated for Q and R to finally obtain the following conjunctive normal form:

```
(¬P∨s)∧(¬s∨Q)∧(¬s∨R)
```

The satisfiability of this equation is the same as that of the original proposition. If the original proposition is satisfiable when the true value of Q∧R is false, s could be set to false. Otherwise, it could be set to true.

(3) DPLL solver

In this study, we implemented the DPLL satisfiability solver [21] in Coq. Compared with the latest achievements in the field of satisfiability solving, the DPLL algorithm is not efficient. However, the problem to be solved is usually not complex in the verification scenario in this study. Moreover, the DPLL algorithm can be implemented recursively and is thus easier to develop in the theorem prover Coq.

The function type of the DPLL algorithm is CNF→partial_asgn→nat→bool. The first parameter is the conjunctive normal form to be solved and the second one is a partial assignment denoted as partial_asgn:= list (ident * bool). As the type of the list is composed of ordered pairs of propositional variables and Boolean values, the second parameter records the assignment of the current part of propositional variables. The third parameter is recursion depth recording the number of propositional variables that can still be selected. The DPLL solver implemented in this study enables depth-based recursion, which can be divided into three steps, namely, assignment derivation, normal form simplification, and variable selection. The solver is defined as follows, where φ is the conjunctive normal form; J is the variable assignment; and n is the recursion depth.

```
DPLL ($\varphi$, J, n)
 if n=0 return T
 else let J′:=UnitPro(J)
   if J′ is conflict, return F
   else let $\varphi'$:=filter($\varphi$, J′)
     then pick one new variable x
        return DPLL($\varphi'$, x↦T;;J′, n−1)||DPLL($\varphi'$, x↦F;;J′, n−1)
```

Note worthily, T would be directly returned when the remaining recursion depth is 0. This is because the solving goal is to check whether the negative proposition of the proposition to be proven is unsatisfiable. Therefore, the solver shall ensure that it could correctly provide feedback on unsatisfiability. When the proposition is excessively complex, the solving goes beyond the recursion depth. In this case, returning T indicates that the solving operation has failed and that whether the normal form is unsatisfiable cannot be determined.

The corresponding DPLL function in Coq is defined as follows.

```
Fixpoint DPLL_UP (P: CNF) (J:partial_asgn) (n:nat):bool:=
match n with
    | O=>true
    | S n′=>
    match unit_pro P J with
        | None=>false
        | Some kJ=>match kJ with
        | nil=>DPLL_filter P J n′
        | _ =>DPLL_UP P (kJ++J) n′
        end
    end
end
with DPLL_filter (P:CNF) (J:partial_asgn) (n:nat):bool:=
    match n with | O=>true | S n′=>
        DPLL_pick (CNF_filter P J) J n′
    end
with DPLL_pick (P:CNF) (J: partial_asgn) (n: nat): bool:=
    match n with | O=>true | S n′=>
        let x:=pick P in
        DPLL_UP P ((x, true):: J) n′ || DPLL_UP P ((x, false):: J) n′
    end
```

The function DPLL_filter is called CNF_filter to simplify the conjunctive normal form to be checked, and CNF_filter would delete all disjunctive clauses that had already been satisfied according to assignment J. Among the remaining clauses, what is known is that the clause could not be satisfied under the current assignment. Except for the "literal" corresponding to the unassigned variable, the "literal" of the assigned variable is bound to be false. Since the disjunction of any proposition and false is equivalent to the proposition itself, the "literal" of the assigned variable could further be deleted, and a conjunctive normal form completely composed of unassigned variables could be obtained for the variable selection in the next step. Variable selection is performed by function `DPLL_pick`, followed by recursion at the next level. What is known is that the variables in conjunctive normal form `P` are all unassigned in this case. Function `pick` simply selects the first propositional variable in the conjunctive normal form for recursion.

Next, we would elaborate on the implementation of assignment derivation. Function `unit_pro` takes the conjunctive normal form and partial assignment as inputs. `None` would be returned if a conflict is observed during the assignment derivation. Otherwise, the newly derived assignment list would be returned. The `unit_pro` in `DPLL_UP` would be repeatedly called until conflicts are identified or new assignments could not be derived. Then, subsequent simplification operations would be conducted. The basis of assignment derivation is derivation for each clause. Function `find_unit_pro_in_clause` is defined to perform this operation, and the type of the return value is `UP_result`.

```
Inductive UP_result := | Conflict | UP (x: ident) (b: bool) | Nothing.
```

The assignment of the last unassigned variable could be derived only when the “literal” of all the other variables in the clause is `false`. Constructor `UP` is used to record this variable and the corresponding derived assignment, while constructors `Conflict` and `Nothing` represent conflicts and no new derivations, respectively. `Find_unit_pro_in_clause` makes decisions successively in a forward fashion while carrying parameter `cont`, whose type is `UP_result` and initial value is `Conflict`. Each “literal” in the clause would be searched for partial assignment `J`. If the variable has already been assigned and its “literal” is `true`, the current clause has been satisfied, and no new assignment could be derived. In this case, `Nothing` is returned. If the value has already been assigned but its “literal” is `false`, further derivation is required. If the current variable could not be found in `J`, it means that an unassigned argument has been encountered. Then, parameter `cont` shall be checked. If the value of `cont` is `Conflict`, it means that an unassigned variable had been encountered for the first time. Then, `cont` shall be modified to an assignment that enabled the current “literal” to be satisfied before checking is continued; If `cont` is already an assignment to a particular variable, it suggests that the variable encountered is the second unassigned variable in the clause, and assignment derivation could not be performed. In this case, `Nothing` is directly returned. When the checking reaches the end of the clause, a statement could be made that the variables in the clause has all been assigned if the value of the parameter `cont` is still `Conflict` and has not been modified. Moreover, all “literal” is `false`, and the current `clause` had one and only one unassigned variable, and the newly derived assignment is recorded in `cont`. The specific implementation of this function is as follow:

```
Fixpoint find_unit_pro_in_clause (c: clause) (J: partial_asgn) (cont:
     UP_result): UP_result :=
match c with
  | nil=>cont
  | (op, x)::c′=>
  match PV.look_up x J with
       | None=>match cont with
       | Conflict=>find_unit_pro_in_clause c′ J (UP x op)
       | UP _ _=>Nothing
       | _ =>Nothing
       end
```

```
   | Some b ⟹if eqb op b then Nothing else find_unit_pro_in_clause c′ J
     cont
  end
end
```

Function `unit_pro` would perform the above operations on each clause. If a `Conflict` occurred, it returns `None`. Otherwise, it would combine all the individual assignments derived into a new assignment list. Note worthily, `x`↦`T` might be derived from one clause when `x`↦`F` is derived from another clause. In this case, `unit_pro` would directly add the two assignments to the newly derived assignment list, and conflict checking would be postponed to the next round of `unit_pro` in `DPLL_UP`. Although different assignments could be made to the same variable in the new assignment `J`, only the one at the top of `J` would be found when `look_up` is used to search for `x` in the assignment. This would inevitably lead to the unsatisfiability of one of the clauses from which two different assignments to `x` are derived in the previous round and ultimately result in a `Conflict`.

(4) Implementation of proof tactic "FOL_Tauto"

The validity denoted as `valid` of first-order logic propositions is defined as follows on the basis of the equivalence between the validity of a proposition and the unsatisfiability of its negative proposition.

```
Definition valid (P: prop): bool:=
match sprop_gen (PNot P) nil "x" with
    | (P′,  _,  n)=>
    match cnf_gen P′ n nil with
        | (P′′,  _)=>negb (DPLL_UP P′′ nil 24)
    end
end
```

For first-order logic proposition `P`, `sprop_gen` is called to generate proposition `P′` of propositional logic corresponding to ¬`P`. Then, `cnf_gen` is called to generate the conjunctive normal form `P′′` corresponding to `P′`. Proposition `P` is claimed valid if the `DPLL` solver could determine that the normal form is unsatisfiable within 24 levels of recursion.

In the proving environment, users has multiple premises in form [[$\Gamma\vdash\varphi$]] (i.e., `derivable` Γ φ). Type `der_judgement:=list prop * prop` is defined to record the propositions involved. Moreover, `list prop` is used to record multiple propositions in the premise, and `prop` is the conclusion proposition in the derivation. Users hope to automatically prove a new proposition in form [[$\Gamma\vdash\varphi$]] on multiple existing conditions. Therefore, "FOL_Tauto" needs to handle multiple derivations as conditions and one derivation as a conclusion. The type is denoted as `proof_goal`.

```
Definition proof_goal: Type:=list der_judgement * der_judgement
```

`Proof_goal` shall be converted into a single first-order logic proposition since `Valid` deals with individual propositions. `Pg2prop` was defined to achieve this goal. For each `der_judgement`, logical connective `PImpl` was used to connect the propositions in the premise and the conclusion proposition and thereby obtain a single first-order logic proposition. Furthermore, `PImpl` is reused to connect the first-order logic propositions converted from each `der_judgement` in `proof_goal`, ultimately obtaining a single first-order logic proposition that could finally be solved automatically.

```
Definition der2prop (d: der_judgement):prop:=
    fold_right PImpl (snd d) (fst d)
Definition pg2prop (pg: proof_goal):prop:=
    fold_right (fun x y=>PImpl (der2prop x) y) (der2prop (snd pg)) (fst pg)
```

The reliability of the DPLL solver is guaranteed by the following property describing the conversion from the DPLL solution results to the proving environment in Coq:

```
Axiom dpll_sound: forall pg : proof_goal, valid (pg2prop pg) = true →
    denote_pg pg
```

where `denote_pg` decomposes `proof_goal` and converts it back to the condition `derivable` in the proving environment in Coq. This theorem declares that if any `proof_goal` is converted into a first-order logic proposition and then determined to be valid by the DPLL solver, the conclusion required by the user could be obtained from the premises in the current proving environment in Coq in the manner of proof construction. This property is declared directly using Axiom.

The final implementation of automated proof tactic “FOL_Tauto” is as follows:

```
Ltac FOL_Tauto :=
first
[reify_pg; apply dpll_sound; reflexivity
| fail 1 "This is not an obvious tautology"].
```

Tactic `reify_pg` calculates `proof_goal` according to the current proving environment, and `dpll_sound` is then applied to convert the goal to be proven into the determination of the validity of the first-order logic proposition. Finally, tactic `reflexivity` is called to enable Coq to automatically complete the DPLL solving.

Take the case of proving that an empty set is a subset of any set as an example: the proof tactic “pose proof” is applied to the axiom of the empty set, which is then instantiated into `y`. In this case, the proof status is as shown in Figure 3.

```
Theorem empty_set_subset:
[[ ZF⊢∀x, ∅⊆x]].
Proof.
    "pose proof" Empty.
    universal instantiation H y.
```

(a) Proof code

```
1 subgoal
H :  [[ ZF⊢∀y, ¬y∈∅]]
H0 :  [[ ZF⊢¬y∈∅]]
____________________________________(1/1)
[[ ZF⊢∀x, ∅⊆x]].
```

(b) Current proof state

Figure 3 Example of proof construction for related properties of empty sets

y∈∅ could be obtained according to condition H0, and a claim could then be made that any proposition could be derived from y∉∅. The definition of a subset supported the statement that the elements in the empty set are all in set `x`. Then, “assert [[ZF⊢y∈∅→y∈x]] by FOL_Tauto” could be used to add a new condition `H1` into the premises, and the proof state changes to the one shown in Figure 4.

```
1 subgoal
H : [[ ZF⊢∀y, ¬y∈∅]]
H0 : [[ ZF⊢¬y∈∅]]
H1 : [[ ZF⊢y∈∅→y∈x]]
________________________(1/1)
[[ ZF⊢∀x, ∅⊆x]]
```

Figure 4 New proof state

This process could be decomposed into the following steps internally. The proof tactic "assert [[ZF⊢y∈∅→y∈x]]" first declared a new branch proof goal, and the proof state of the current branch is as shown in Figure 5.

```
2 subgoals
________________________ (1/2)
valid (pg2prop ([
([ ·], [[ ∀y, ¬y∈∅]]);
([ ·], [[ ¬y∈∅]])
],
([ ·], [[ y∈∅→y∈x]])))=true
```

(a) Proof state of current branch after assert

```
2 subgoals
________________________ (1/2)
valid
[[( ∀y, ¬y∈∅) →
¬y∈∅→
y∈∅→
y∈x]] = true
```

(b) Proof state after `reify_pg`

Figure 5 Decomposition of proof steps of "FOL_Tauto"

"FOL_Tauto" would first execute the tactic "reify_pg", and the conditions and conclusion in Figure 5(a) would be converted into the form `denote_pg` in Figure 5(b) by instructions such as "reverse" and "change". The `proof_goal` corresponding to Figure 5(a) has already been calculated. This proof objective consists of two parts. Part 1 is the list `der_judgement` calculated from the premises `H` and `H0`; since the environments in both `H` and `H0` are empty environment ZF, Part 1 of each `der_judgement` is invariably an empty list of propositions while Part 2 is the `der_judgement` corresponding to the conclusion declared by the user.

Afterward, `dpll_sound` is applied to change the proof condition to validity check after `proof_goal` conversion: Coq could automatically calculate `pg2Prop` and convert `proof_goal` into the first-order logic proposition given in Figure 6(b). Finally, `valid` would automatically call proposition conversion, generation of conjunctive normal forms, DPLL-based solving, and other steps to calculate that the proposition is truly valid, and the proof state in Figure 4 is returned. In a user's actual proving environment, all the above steps are hidden in the one-step operation of the tactic "`assert` [[ZF⊢y∈∅→y∈x]] by `FOL_Tauto`".

4.2 Automated proof tactic for quantifiers

Section 3.3 presented two reasoning rules for universal quantifiers and existential quantifiers, respectively. We have defined four automated proof tactics for these four rules, respectively, and

```
2 subgoals
H : [[ ZF⊢∀y, y∈∅]]
H0 : [[ ZF⊢y∈∅]]
________________________ (1/2)
[ZF⊢y∈∅→ y∈x]
```

(a) Proof state after `dpll_sound` is applied

```
2 subgoals
________________________ (1/2)
Denote_Pg[
([ ·], [∀y, ∀y∈∅]]);
([ ·], [μy∈∅]])
],
([ ·], [y∈∅→y∈x]])
```

(b) Proof state after `pg2Prop` is calculated

Figure 6 proof_goal conversion

the implementation of two of them is presented below.

(1) Tactic “universal generalization” for introducing universal quantifiers

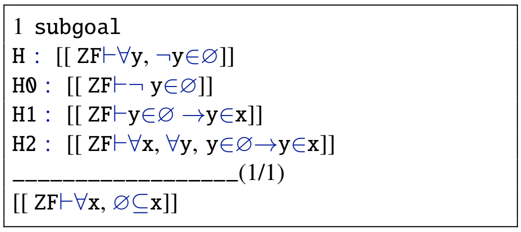
In the proof construction for the statement that the empty set is a subset of any set in Figure 4, condition [[ZF⊢y∈∅→y∈x]] is obtained by “FOL_Tauto”. However, the goal to be proven is [[ZF⊢∀x, ∅∈x]], that is, [[ZF⊢∀x∀y, y∈∅→y∈x]] (the subset symbol is also abbreviated through “notation”), and condition H1 differed from the goal to be proven by two levels of universal quantifiers. Rule `PForall_intros` for introducing quantifiers is represented here.

$$\forall_{Intro}\frac{\forall\varphi\in\Gamma,\ x\notin FV(\varphi)\quad \Gamma\vdash P}{\Gamma\vdash\forall x, P}$$

The condition that the variable does not appear freely in all the propositions in the premise should be met when a universal quantifier is to be added before the conclusion proposition. We have developed proof tactic “universal_generalization_constr”, and this tactic takes two parameters. One is the number, such as H1, of the premises calling for the addition of quantifiers in the Coq environment, or more accurately, the proof goal of the condition (i.e., `derivable` Γ φ). The other parameter is the variable name list `xs` used to implement the one-time introduction of multiple universal quantifiers. The implementation of this tactic is as follows: when the list to be introduced is empty, the introduction ends, and `H` is returned. Otherwise, the derivation of whether condition `H` is derivable is checked, and rule `PForall_intros` is then applied to construct a new proof condition. Specifically, tactic “check_free_occurrence” could automatically determine whether variable `x` appeares freely in the premise `Phi`. Tactic “universal generalization” calls “universal_generalization_constr”. if the call is successful, “position proof” is applied to the new premise in the proving environment. Otherwise, an error message would be displayed.

```
Ltac universal_generalization_constr H xs :=
match xs with
    | nil => constr:(H)
    | ?x::?xs0=>
    match type of H with
        | [[? Phi⊢?P]]=>let H0 := constr:(PForall_intros Phi x P
          ltac:(check_free_occurrence) H) in
         universal_generalization_constr H0 xs0
      end
end
Ltac universal_generalization H xs :=
    first
    [let H0 := universal_generalization_constr H xs in pose proof H0
        | fail 1"Universal generalization fails"]
```

In the proof construction displayed in Figure 4, a new condition H2 could be obtained by running the tactic "universal generalization H1 x y". Then, H2 is the goal to be proven, and tactic "The conclusion is already proven" is called to end the proof.



```
1 subgoal
H :  [[ ZF⊢∀y, ¬y∈∅]]
H0 :  [[ ZF⊢¬ y∈∅]]
H1 :  [[ ZF⊢y∈∅ →y∈x]]
H2 :  [[ ZF⊢∀x, ∀y, y∈∅→y∈x]]
____________________(1/1)
[[ ZF⊢∀x, ∅⊆x]]
```

Figure 7 Running effect of universal generalization

(2) Tactic "existential generalization" for introducing existential quantifiers

Tactic "existential generalization" for automatically introducing existential quantifiers differs from "universal generalization" in style. The rule PExists_intros for introducing existential quantifiers is the following:

$$\exists_{Intro}\frac{\Gamma \vdash P[x \mapsto t]}{\Gamma \vdash \exists x, P}$$

∃x, P could be derived from the premise if the conclusion after the x in P is instantiated could be derived from it. However, different from the case of universal quantifiers, users could no longer just point out the first-order logic items they wanted to abstract. If premise [[ZF⊢∅=∅]] is present in the proving environment, the conclusion the user needed is likely to be [[ZF⊢∃x, x=∅]]. If the proposed tactic only allowed the user to specify the term ∅ to be abstracted, conclusion [[ZF⊢∃x, x=x]], which is inconsistent with the user's expectations, would be obtained. For this reason, the tactic "existential generalization" takes not only the number of premises but also a proposition specified by the user. Users only need to directly write the conclusions they want to obtain. The above tactic would automatically search the first-order logic term abstracted by the user to apply the rules for introducing existential quantifiers.

Assuming that a condition H: [[ZF;; x∪y=z⊢z=x∪y]] is present in the proving environment and that the user executed the tactic "existential generalization H[[∃u, y=u]]", a new condition H0:[[ZF;; x∪y=z⊢∃u, y=u]] could be directly added to the proving environment. This tactic would automatically search for u corresponding to the original term x∪y and apply the rules for introducing existential quantifiers. This tactic also allows users to introduce multiple existential quantifiers in one go.

The implementation of the tactic "existential generalization" is as follows, with parameter H being the number of the premise and P being the proposition declared by the user.

```
Ltac existential_generalization_tac H P:=
    first [
        let H0:=fresh "H" in
        match type of H with
           | [[? Phi⊢?Q]]=>first [let ts:= generate_exists_term_tac
               P Q in assert [[Phi⊢P]] as H0;
               [apply (existential_generalization_tac_aux ts);
               apply Alpha_congruence with Q; cbv; easy |] |
               idtac "Cannot infer the terms to be quantified in" H;
               fail]
        end; | fail 1 "Existential generalization fail"]
```

Specifically, “`generate_exists_term_tac`” is called according to proposition `P` specified by the user and the conclusion proposition `Q` in the premise to search for the first-order logic term abstracted by the user. This tactic would compare `P` with `Q` to determine the list of existential quantifier variables newly introduced by the user into `Q`. Then, it searched for each variable `x` separately. If the variable in proposition `Q` is `x` whereas the one in the same position in proposition `P` is a first-order logic term `t`, an assumption is made that the user had abstracted `t` into a first-order logic term. Sub-terms are searched separately when logical connectives appears. No conflicting results on both sides shall be ensured when results are returned, and the corresponding term list `ts` is ultimately generated. Afterward, the export rule “`existential_generalization_tac_aux`” is applied. This rule is a version involving multiple substitutions of the reasoning rule for introducing quantifiers. After the proposition abstracted by the user is instantiated again using `ts`, rule $\alpha_{CONGRUENCE}$ is applied. The function `aeq` in Section 3.2 is called to determine whether the instantiated proposition is equivalent to the original proposition `Q` in the condition, thereby completing the introduction and proof of the new condition.

(3) Quantifier elimination tactic

The tactic “universal instantiation” shares the same style with the tactic “universal generalization”. If conditions similar to `H`: [[Γ⊢∀`x`, ∀`y`, . . .]] are present in the environment, the tactic “universal instantiation `H t1 t2`” could be executed to sequentially instantiate `x, y,` . . . in the proposition into `t1, t2,` $\cdots$. A specific example is illustrated in Figure 2(b) as the effectiveness from condition H to condition H0.

Tactic “existential instantiation” has the same style and idea as those of tactic “existential generalization”. Users directly write new propositions to which they want to add the existential quantifiers in the premises, and tactic “existential instantiation” would automatically search for the corresponding proposition in the specified premise and apply rules $\exists_{Elim}$ and $\alpha_{CONGRUENCE}$ to generate a new condition.

4.3 Automated proof tactic for equal sign replacement

The following reasoning rule is available for the equal sign. If the first-order logic term `t` is the same as `t′`, `P[x↦t]` implies `P[x↦t′]`. Nevertheless, directly applying this reasoning rule is troublesome. Users usually have two propositions with the same terms and containing one of them, and the terms in this proposition shall then be replaced by a new one instead of a proposition before instantiation. For this reason, the tactic “`peq_sub_tac`” for replacing the equal sign is developed for the purpose of generating and replacing proposition P.

$$=_{Subst}\ \frac{}{\vdash t = t' \to P[x \mapsto t] \to P[x \mapsto t']}$$

`peq_sub_tac` takes three parameters, namely, the equal terms `t` and `t′`, as well as the `t`-containing proposition `P` declared by the user. This tactic would introduce a new variable name v that does not appear in P. replace_prop_tac is applied to replace all terms t in P with v to obtain proposition Q, and rule PEq_sub is used to construct a new proposition Q[v↦t′]. A new condition [[ZF⊢t=t′→P→Q′]] is added into the proving environment, and Q′ is in the form after Q [v↦t′] is expanded, with the abbreviation of the predicate remaining unexpanded. The implementation of tactic "peq_sub_tac" is as follows.

```
Ltac peq_sub_tac t t′ P:=
    let H0:=fresh "H" in
    let H1:=fresh "H" in
    let v:=get_new_var P ShortNames.x [[∅]] ShortNames.x in
    let Q:=replace_prop_tac v t P in
         pose proof (PEq_sub Q v t t′) as H1;
    let Q′:=subst_aeq_constr_tac Q v t′in
        assert [[ZF;; t=t′;; P⊢Q′]] as H0 by FOL_Tauto; clear H1
```

5 Application in Teaching

We introduced the prover we developed into the teaching of the course discrete mathematics for freshmen and assigned the proof construction for mathematical induction with this prover as a course project. Practice shows that students who choose this project can successfully construct a formal proof of mathematical induction by drawing on the proof construction idea for this theorem in textbooks. Some outstanding students have further formalized the Peano arithmetic with this tool after the course ended.

5.1 Course project: Formal proof of mathematical induction

Although the set theory ZFC has been widely accepted as a foundation of mathematics, we prefer that students know the connection between ZFC and everyday mathematics. In this aspect, mathematical induction, as a well-known method of proof construction for students, is a fair research object. Although students usually acquiesce in the correctness of this method, mathematical induction is a provable theorem in the framework of the axiomatic set theory ZFC. For this reason, we assigned the proof construction for mathematical induction with the proposed prover as one of the two optional additional course projects, and students could choose either this project or the other optional task.

(1) Project design

We have provided a tutorial document for the above project. This v file to be used in the prover Coq includes 300 lines of annotated explanations, demonstrating the syntax of propositions and logic systems, the definition of predicate symbols such as subset relationship, the basic operations of Coq (key usage and special symbol input), and the functions of five automated proof tactics (FOL_Tauto and tactics for quantifiers since this project does not require the use of the tactic "peq_sub_tac" for equal sign replacement). Furthermore, the proof construction in a total of 30 lines of proof code for four simple theorems is provided in the document to familiarize students with how to construct proofs in the proposed proving environment. We have also provided a supporting 15 min demonstration video for this tutorial. With the video and the tutorial document in hand, students are generally well-prepared to construct proofs in this environment, and no additional class hours are provided for teaching the usage of Coq.

For the proof construction for mathematical induction that students shall complete, we have defined the predicates is_inductive and is_natural_number, which indicated that a set is an inductive set and a set is a natural number set, respectively.

```
Definition is_inductive_def(t:term):=[[(∅∈x∧∀y, (y∈x →y∪{y}∈x)) [x↦t]]].
Definition is_natural_number_def(t:term):=[[(is_inductivex∧∀w,
    (is_inductive w)→x⊆w)[x↦t]]].
```

Then, we presented the goal to be proven so that students could gradually construct formal proofs of theorems such as mathematical induction. Students are required to use the proposed proof tactics and tactic "pose proof" to prove the following goals one after the other.

- The sets that are subsets of each other are equal.

```
Theorem subset_subset_equal: [[ZF⊢∀x, ∀y,x ⊆y → y⊆x→x=y]].
```

- A set of natural numbers is unique.

```
Theorem Nat_unique: [[ZF;;is_natural_number x;;is_natural_number y⊢x=y]].
```

- The basic step of mathematical induction is as follows: if N is a set of natural numbers and the natural number 0 is in set X, 0 is also in X∩N.

```
Theorem Nat_inductive_base_step:
 [[ ZF;; is_natural_number N;;∅∈X;;∀n, n∈N→n∈X→n∪{n}∈X⊢∅∈X∩N]].
```

- The inductive step of mathematical induction is as follows: If 0 and $S(n)$ of an arbitrary natural number n are both in set X, X∩N is an inductive set.

```
Theorem Nat_intersection_ inductive:
[[ ZF;; is_natural_number N;;∅∈X;;∀n, n∈N→n∈X→n∪{n}∈X⊢is_inductive
    (X∩N)]].
```

- Mathematical induction: If 0 is in set X, and $S(n)$ for an arbitrary natural number n is also in set X, this arbitrary natural number n is in set X.

```
Theorem mathmetical_induction:
[[ ZF;; is_natural_number N;;∅∈X;;∀n, n∈N→n∈X→n∪{n}∈X⊢∀n, n∈N→n∈X]].
```

Students can also add and prove theorems in intermediate steps themselves as needed by the proof.

(2) Teaching effectiveness

Among the 15 students who chose to participate in the additional course projects, 8 chose the project of constructing a formal proof of mathematical induction, demonstrating their interest in using theorem provers to complete formal proofs. These eight students completed the proof, and statistics showed that they used an average of 88 lines of code to prove the above goal. In particular, two students completed the proof in 65 and 66 lines of code, respectively. The proof drafted by the teacher and the teaching assistant divided the proof process into multiple sub-goals and took a total of 67 lines. The remaining six students all took around 100 lines of code. The reason for their high number of lines of code is that they have divided the proof that can be done in one step by the automated proof tactic "FOL_Tauto" in the proof process into several more basic proofs close to the reasoning rule for each logical connective and complete each proof by applying "FOL_Tauto", respectively. Consequently, some students failed to solve the problem during the proof because the recursion depth of the DPLL exceeded the preset value, which could be further attributed to the excessive conditions in the proof environment. After discussing with the teaching assistant, they disassembled the proof into more sub-goals and finally completed the formal proof of mathematical induction.

To sum up, students who have never been exposed to interactive theorem proving can quickly familiarize themselves with the method of constructing formal proofs of mathematical theorems using interactive theorem provers in the proving environment developed in this study. All students can successfully construct a strict formal proof of mathematical induction with the help of the Coq-based interactive theorem prover, and some students are able to reach a high level of application, which is difficult in the traditional teaching mode. The introduction of interactive theorem provers into teaching can help students understand the logic system and the axiomatic set theory ZFC more thoroughly. Overall, we have met the teaching objectives and demonstrated the availability and usability of the proposed tool.

(3) Proof instances

This section presents the proof construction in Coq of the basic step of mathematical induction and compares it with the proof of logic in textbooks to intuitively reflect the effectiveness of the proposed theorem proving tool. One step in the proof construction in Coq is likely to correspond to multiple steps in the proof construction in textbooks. For this reason, the corresponding steps share the same line numbers for the convenience of comparison.

The basic step denoted as `Nat_inductive_base_step` is defined as follows: if `N` is a set of natural numbers and the natural number 0 is in set `X`, 0 is also in `X∩N`.

```
1  Lemma Nat_inductive_base_step:
2  [[ ZF;; is_natural_number N;;∅∈ X;;∀n, n∈N→n∈X→n∪{n}∈X⊢∅∈X∩N]].
3  Proof.
4      pose proof Intersection_iff.
5      universal instantiation H X N [[∅]].
6      The conclusion is already proven.
7  Qed.
```

The formal proof in textbooks corresponding to this basic step is as follows.

```
4  ZF⊢∀x, ∀y, ∀z, z∈x∩y↔z∈x∧z∈y                         (binary intersection rule)
5  ZF⊢∀y, ∀z, z∈X∩y↔z∈X∧z∈y                    (apply ∀Elim to instantiate x into X)
5  ZF⊢∀z, z∈X∩N↔z∈X∧z∈N                       (apply ∀Elimt to instantiate y into N)
5  ZF⊢∅∈X∩N↔∅∈X∧∅∈N                           (apply ∀Elim to instantiate z into ∅)
6  ZF⊢∅∈X∧∅∈N→∅∈X∩N                                          (elimination rule ↔)
6  ZF;; is_natural_number N;;∅∈X;;∀n,n∈N→n∈X→n∪{n}∈X⊢∅∈X∧∅∈N→∅∈X∩N
       (rule Weaken)
6  ZF;; is_natural_number N⊢∅∈N  (definition of is_natural_number and conjunction and
       elimination rule)
6  ZF;; is_natural_number N;;∅∈X;;∀n, n∈ N→n∈X→n∪{n}∈X⊢∅∈N            (rule Weaken)
6  ZF;; ∅∈X⊢∅∈X                                                         (rule Assu)
6  ZF;; is_natural_number N;;∅∈X;;∀n, n∈N→n∈X→n∪{n}∈X⊢∅∈X             (rule Weaken)
6  ZF;; is_natural_number N;;∅∈X;;∀n, n∈ N→n∈X→n∪{n}∈X⊢∅∈X∧∅∈N        (rule for
       conjunction and introduction)
6  ZF;; is_natural_number N;;∅∈X;;∀n, n∈N→n∈X →n∪{n}∈X⊢∅∈X∩N        (rule Modus
       Ponens and lemma is proven)
```

If the reasoning rules are applied one by one strictly according to the textbooks, the proof construction for this lemma takes more than ten steps. In contrast, it can be done by the proof tactic we developed in Coq in only three lines. Specifically, the tactic "universal instantiation" in line four used the reasoning rule $\forall_{Elim}$ three times in one step and instantiated three quantifiers. Moreover, the tactic "The conclusion is already proved" (FOL_Tauto) combines the reasoning rules for multiple logical connectives. The final conclusion is directly proven, with no rules for quantifiers involved. On the premise of strict formalization, students can construct proofs more conveniently with the tool we developed in Coq than by constructing pen-and-paper proofs. In addition, Coq can check the correctness of the proofs constructed by students in realtime.

Furthermore, each step in the proof construction in textbooks is a derivation relationship, which is exactly what the proof state bar in Coq displays when the proposed tool is used to construct proofs (Figure 1(b)). The proof state bar records the conclusions the student has currently proven. The proof code of Coq input by students determines how to select appropriate conditions and apply the reasoning rules. The status of the proof state bar is equivalent to the explanation of the reasoning rules in parentheses in the textbook-style proof above. The proposed approach is thus consistent with the idea of the proofs in textbooks and equally intuitive as those proofs.

The proof construction for the inductive step `Nat_intersection_inductive` is presented below, including the Coq code and proof state. The proof state can also be seen as the step in the proof construction in textbooks. Before proving `Nat_intersection_inductive`, we proved two lemmas first.

The successors of the elements in `X∩N` are all in `X`.

```
1  Lemma Nat_inductive_inductive_step_X:
2  [[ ZF;; is_natural_number N;;∅∈X;;∀ n, n∈N→n∈X→n∪{n}∈X;;y∈X∩N⊢y∪{y}∈X]].
3  Proof.
4      pose proof Intersection_iff.
5      universal instantiation H X N y.
6      assert [[ZF;; ∀n,  n∈N→n∈X→n∪{n}∈X⊢∀n, n∈N→n∈X→n∪{n}∈X]] by
        FOL_Tauto.
7      universal instantiation H1 y.
8      The conclusion is already proven.
9  Qed.
```

The corresponding proof state is as follows, with the line numbers corresponding to those of the previous Coq code. In addition, the condition numbers, such as `H` and `H0`, displayed in Coq have also been added for easy comparison with the tactics in Coq.

```
4  H:[[ ZF⊢∀x, ∀y, ∀z,  z∈x∩y↔z∈x∧z∈y]]                     (binary intersection rule)
5  H0:[[ ZF⊢y∈X∩N↔y∈X∧y∈N]]             (universal instantiation instantiates x, y,
        and z in H separately)
6  H1:[[ ZF;; ∀n,  n∈N→n∈X→n∪{n}∈X⊢ZF;;∀n, n∈N→n∈X→n∪{n}∈X]]        (rule Assu)
7  H2:[[ ZF;; ∀n,  n∈N→n∈X→n∪{n}∈X⊢y∈N→y∈X→y∪{y}∈X]]    (instantiate H1 into y)
8  [[ ZF;; is_natural_number N;;∅∈X;;∀n, n∈N→n∈X→n∪{n}∈X;;y∈X∩N⊢y∪{y}∈X]]
        (apply "FOL_Tauto" to construct a proof according to H0 H2)
```

The lemma that the successors of the elements in `X∩N` are all in `N` can be proven in a similar manner, and the inductive step `Nat_intersection_inductive` can be further obtained. The final proof construction for mathematical induction is presented as follows.

```
1  Theorem mathmetical_induction:
2      [[ ZF;; is_natural_number N;;∅∈X;;∀n, n∈N→n∈X→n∪{n}∈X⊢∀n,
        n∈N→n∈X]].
3  Proof.
4      pose proof Nat_intersection_inductive.
5      assert ([[ZF;; is_natural_number N⊢|−−∀w, is_inductive w→N⊆w]]) by
        FOL_Tauto.
6      universal instantiation H0 [[X∩N]].
7      assert ([[ZF;; is_natural_number N;;is_inductive X∩N⊢N⊆X∩N]]) by
```

```
     FOL_Tauto.
8    universal instantiation H2 n.
9    pose proof Intersection_iff.
10   universal instantiation H4 X N n.
11   assert ([[is_natural_number N;;∅∈X;;∀n,
      n∈N→n∈X→n∪{n}∈X⊢n∈N→n∈X]]) by FOL_Tauto.
12   universal generalization H6 n.
13   The conclusion is already proven.
14 Qed.
```

The corresponding proof states are as follows.

4 H: [[ZF;; is_natural_number N;;∅∈X;;∀n, n∈N→n∈X→n∪{n}∈X⊢is_inductive X∩N]] (the inductive step Nat_intersection_inductive already proven by "pose proof")

5 H0:[[ZF;; is_natural_number N⊢∀w, is_inductive w→N⊆w]] (definition of is_inductive and conjunction and elimination)

6 H1:[[ZF;; is_natural_number N⊢is_inductive X∩N→N⊆X∩N]] (instantiation of w in H0 into X∩N by universal instantiation)

7 H2:[[ZF;; is_natural_number N;;is_inductive X∩N⊢N⊆X∩N]] (property of material implication→)

8 H3:[[ZF;; is_natural_number N;;is_inductive X∩N⊢n∈N→n∈X∩N]] (definition of subsets)

9 H4:[[ZF⊢∀x, ∀y, ∀z, z∈x∩y↔z∈x∧z∈y]] ((binary intersection rule pose proof)

10 H5:[[ZF⊢n∈X∩N↔n∈X∧n∈N]] ((separate instantiation of x, y, and z in H4 by universal instantiation)

11 H6:[[ZF;; is_natural_number N;;∅∈X;;∀n, n∈N→n∈X→n∪{n}∈X⊢n∈N→n∈X]] (obtained by "FOL_Tauto" according to H H3 H5)

12 H7:[[ZF;; is_natural_number N;;∅∈X;;∀n, n∈N→n∈X→n∪{n}∈X⊢∀n, n∈N→n∈X]] (introduction of quantifier n before the conclusion of H6 by universal generalization and mathematical induction is proven)

5.2 Formalization of Peano arithmetic

Some outstanding students further used the proposed tool to formalize addition and multiplication in Peano arithmetic after completing the course of discrete mathematics, which reflected the effectiveness of this tool. Some important definitions and theorems are listed below.

First, addition is defined: regarding any natural number n, 3-tuple $(0, n, n)$ is in set X; regarding any 3-tuple (n, d, e) in X, 3-tuple $(S(n), d, S(e))$ is also in set X. The predicate "in_rel 3 x y z d" represents that 3-tuple (x, y, z) is in d.

```
Definition is_legal_plus_def(t1 t2:term):prop:=
    [[(( ∀n, n∈N→in_rel3∅n n x)∧(∀n,∀d,∀e, n∈N∧d∈N ∧e∈N→( in_rel3 n d e
     x→in_rel3 n∪{n} d e∪{e} x))) [x↦t1; N↦t2]]]
```

- Set X defines that the addition relationship on the natural number N is represented by the predicate "is_plus X N", and set X is defined as the one of the minimum satisfaction "is_legal_plus".

```
Definition is_plus_def(t1 t2:term):=
    [[( is_legal_plus x N∧(∀y, is_legal_plus y N→x⊆y))[x↦t1 ; N↦t2]]]
```

Subsequently, the definition of multiplication can be presented.

For any natural number n, 3-tuple $(n, 0, 0)$ is in set f, and for any 3-tuple (x, y, z) in set f, 3-tuple $(x, S(y), a)$ is in set f can be derived if (z, x, a) is in additional relationship e.

```
Definition is_legal_mult_def(t1 t2 t3:term):prop:=
    [[((  ∀n,  n∈N→in_rel3 n∅∅f)∧(∀x, ∀y, ∀z, ∀a,
     x∈N∧y∈N∧z∈N∧a∈N→in_rel3 x y z f→(in_rel3 z x a e→in_rel3 x y∪{y}
     a f ))) [ f↦t1; e↦t2; N↦t3]]]
```

- Set X defines the multiplication relationship on the natural number N based on the additional relationship e and represents it by the predicate “is_mult X e N”. Set X is defined as the one of the minimum satisfaction “is_legal_mult”.

```
Definition is_mult_def(t1 t2 t3:term):=
    [[( is_legal_mult x e N∧(∀y, is_legal_mult y e N→x⊆y))[x↦t1; e↦t2;
     N↦t3]]]
```

The above definitions can be used to further present the proof construction for several Peano postulates, which have already been proven by mathematical induction.

- The natural number 0 is not a successor of any number.

```
Lemma not_empty: [[ZF⊢∀n, ¬n∪{n}=∅]]
```

- If the successors of two natural numbers are equal, the two natural numbers are equal, and the same is true in the opposite direction according to the rule for equal sign replacement.

```
Lemma Sn_inversion: [[ZF⊢∀x,∀y,x∪{x}=y∪{y}→x=y]]
```

Afterward, the relevant properties of addition are proven.

- An addition set is unique.

```
Lemma plus_unique: [[ZF;;is_natural_number N⊢∀x,∀y,is_plus x
     N\wedgeis_plus y N→x=y]]
```

- For any two natural numbers, a natural number that is their sum is bound to be present.

```
Lemma in_plus_exists:
    [[ ZF;; is_natural_number N;;is_plus e N⊢∀x,∀y, x∈N∧y∈N→∃z,z∈N∧
     in_rel3 x y z e]]
```

The functionality, commutative property, and associative property of addition require separate proof construction for their basic steps and inductive steps. The mathematical induction proven in the previous section is then applied to construct the proofs. Here, only the proof construction for the functionality of addition is presented as an example.

- The basic step of the functionality of addition is as follows: if 0+x=y, x=y.

```
Lemma plus_func_zero:
    [[ ZF;; is_natural_number N;;is_plus e N⊢∀x, ∀y, x∈N∧y∈N∧in_rel3 ∅x y
     e→x = y]]
```

- The inductive step of the functionality of addition is expressed below: if z=a can be derived from x+y=z and x+y=a, then z'=a' can be derived from S(x)+y=z' and S(x)+y=a'

```
Lemma plus_func_induction:
    [[ ZF;; is_natural_number N;;is_plus e N⊢∀x,
     x∈N∧(∀y,∀z,∀a,y∈N∧z∈N∧a∈N ∧in_rel3 x y z e∧in_rel3 x y a e→z =
     a)→x∪{x}∈N∧(∀y, ∀z, ∀a,y∈N∧z∈N∧a∈N∧in_rel3 x∪{x} y z e∧in_rel3
     x∪{x} y a e→z=a)]
```

- The functionality of addition can be expressed as follows: the sum of any two natural numbers is unique.

```
Lemma plus_func:
    [[ ZF;; is_natural_number N;;is_plus e N ⊢∀x,∀y, ∀z, ∀a,
     x∈N∧y∈N∧z∈N∧a∈N∧in_rel3 x y z e∧in_rel3 x y a e→z=a]]
```

- The commutative property of addition is as follows: if x+y=z, then y+x=z.

```
Lemma plus_comm:
    [[ ZF;; is_natural_number N;;is_plus e N⊢∀x, ∀y, ∀z,
     x\inN∧y∈N∧z∈N∧a∈N∧in_rel3 x y z e→in_rel3 y x z e]]
```

- The associative property of addition can be expressed as follows: if x+y=a and y+z=b, then a+z=c if and only if x+b=c, that is, x+(y+z)=(x+y)+z.

```
Lemma plus_assoc:
[[ ZF;; is_natural_number N;;is_plus e N⊢
    ∀x, ∀y, ∀z, ∀a, ∀b, ∀c, x∈N∧y∈N∧z∈N∧a∈N∧b∈N∧c∈N∧in_rel3 x y a
     e∧in_rel3 y z b e→(in_rel3 a z c e↔in_rel3 x b c e)]]
```

Finally, proof construction for the related properties of multiplication is provided.

- Multiplication sets are unique.

```
Lemma mult_unique:
    [[ ZF;; is_natural_number N;;is_plus e N⊢∀x, ∀y, is_mult x e N∧is_mult
     y e N→x=y]]
```

- For any two natural numbers, their product is bound to be present.

```
Lemma in_mult_exists:
    [[ ZF;; is_natural_number N;;is_plus e N;;is_mult f e N⊢∀x, ∀y,
     x∈N∧y∈N→∃z, z∈N∧in_rel3 x y z f]]
```

- The functionality of multiplication can be expressed as follows: The product of any two natural numbers is unique.

```
Lemma mult_func:
    [[ ZF;; is_natural_number N;;is_plus e N;;is_mult f e N⊢∀x, ∀y, ∀z, ∀a,
     x∈N∧y∈N∧z∈N∧a∈N∧in_rel3 x y z f∧in_rel3 x y a f→z=a]]
```

- The commutative property of multiplication is expressed as follows: if x×y=z, then y×x=z.

```
Lemma mult_comm:
    [[ ZF;; is_natural_number N;;is_plus e N;;is_mult f e N⊢∀x, ∀y, ∀z,
     x∈N∧y∈N∧z∈N∧in_rel3 x y z f→in_rel3 y x z f]]
```

- The associative property of multiplication is expressed as follows: if x×y=a and y×z=b, then a × z=c if and only if x×b=c, i.e., (x×y)×z=x×(y)×z).

```
Lemma mult_assoc:
    [[ ZF;; is_natural_number N;;is_plus e N;;is_mult f e N⊢∀x, ∀y, ∀z, ∀a, ∀
     b, ∀c, x∈N∧y∈N∧z∈N∧a∈N∧b∈N∧c∈N∧in_rel3 x y a f∧in_rel3 y z b
    f→(in_rel3 a z c f↔in_rel3 x b c f)]]
```

- The distributive property of multiplication can be expressed as follows: `a+b=c`, i.e., `x×y+x\times z=x×(y+z)` if `x×y=a`, `x×z=b`, `x×d=c`, and `y+z=d`.

```
Lemma mult_dist:
  [[ ZF;; is_natural_number N;;is_plus e N;;is_mult f e N⊢∀x, ∀y,∀z, ∀a, ∀b, ∀
    c, ∀d, x∈N∧y∈N∧z∈N∧a∈N∧b∈N∧c∈N∧d∈N∧in_rel3 y z d e∧in_rel3 x y
    a f∧in_rel3 x z b f∧in_rel3 x d c f→in_rel3 a b c e]]
```

6 Comparison with Previous Research

For students, the theorem proving tool developed in this study is closed. They do not need to learn anything that is most fundamental in Coq but unrelated to the teaching content of the course, such as inductive types, even if they have completed the proof construction for mathematical induction with the tool we developed in Coq. In this sense, the present study is quite different from the previous ones. Avigad used Lean to teach naive set theory and the axiomatic set theory in logic courses for undergraduate students. The logic built into Lean is directly used in the course to formalize the relevant concepts of set theory. This method did not require much development work, and the interaction in Lean is also friendly. Students could successfully construct proofs of the theorems in set theory as well after they familiarized themselves with how to use Lean. However, the author needed to allocate dedicated class hours to teach the usage of Lean as part of the course content. Furthermore, the proofs constructed in Lean appeared to be slightly lengthy due to the lack of improved proof tactics. For this reason, the author only required students to construct proofs for simple examples with Lean. His approach is applicable to teaching scenarios where the related concepts of set theory are not directly presented in the framework of first-order logic as directly constructing formal proofs in the framework of the first-order logic would be overcomplicated. The author expected that more automated procedures can be provided in Lean to facilitate students to construct more proofs more easily in the future. In this study, Ltac in Coq is employed to develop more convenient proof tactics, and traditional logic symbols are provided through "notation". Students can preliminarily grasp the automated proof tactics we developed from the 15 min tutorial video and then quickly complete the proof construction for important theorems, such as mathematical induction. Since the cost of learning to use this tool is low, Coq does not need to be taught as course content.

As far as we know, attempts to introduce interactive theorem provers into the teaching of set theory have rarely been reported in China.

7 Summary

Interactive theorem provers can help students learn mathematical logic better by giving them fast and accurate support. To solve the problem that the existing theorem provers have a high threshold for getting started with them and differ greatly from textbooks, we developed a prover for the axiomatic set theory in Coq and provided students with a proving environment similar to the one in textbooks. Starting from the abstract syntax tree of logic formula, we formalized the reasoning system of the axiomatic set theory ZFC. We also provided several automated proof tactics for the reasoning system, including the automated proof tactic "FOL_Tauto" for propositional logic that involved the generation of disjunctive normal form and calls the DPLL satisfiability solver implemented in Coq. The other automated proof strategies developed are the proof tactics for quantifiers that supported the introduction and elimination of multi-level quantifiers, as well as the simpler proof strategy for equal sign replacement. Compared with directly using existing theorem provers, the closed proving environment we provided hid the logic built into the native theorem prover and is thus close to the textbook style. In this way, it allowed students to focus on the logic they are learning. The automated proof tactics also alleviated the unnecessary burden of learning the usage of theorem provers, making it easy to understand and use. In actual teaching, the proof strategies we provided will suffice for students to prove mathematical induction and complex theorems such as Peano arithmetic, reflecting the accessibility and effectiveness of the proposed tool. Note worthily, this tool is developed for particular teaching scenarios, and it may need to be adjusted according to actual situations to satisfy different teaching needs.

Interactive theorem provers are also applicable to other teaching scenarios in addition to that of mathematical logic. At present, Coq and Lean are the most used interactive theorem provers in teaching. Lean provides better support for formal mathematics. For instance, Thoma and Iannone[22] used Lean to teach number theory. In contrast, Coq is more applicable for formalizing computer-related theories. Pierce *et al.* wrote the famous Software Foundations[23], covering concepts such as program logic and program semantics, and it has been widely applied in courses of program language theories. However, the use of interactive theorem provers in mathematics courses still focuses on the teaching of mathematical logic, and the application of such provers remains to be further investigated.

References

[1] The Coq Development Team. The Coq Proof Assistant. http://coq.inria.fr.

[2] Nipkow T, Wenzel M, Paulson LC. Isabelle/HOL: A proof assistant for higher-order logic. Springer, 2002. [doi: 10.1007/3-540-45949-9]

[3] Bancerek G, Byliński C, Grabowski A, Korniłowicz A, Matuszewski R, Naumowicz A, Pak K, Urban J. Mizar: State-of-the-art and beyond. Proc. of the Int'l Conf. on Intelligent Computer Mathematics. 2015. 261–279. [doi: 10.1007/978-3-319-20615-8_1]

[4] de Moura L, Kong S, Avigad J, Doorn F, Raumer J. The Lean theorem prover (system description). Proc. of the Int'l Conf. on Automated Deduction (CADE 2015). 2015. 378-388.[doi: 10.1007/978-3-319-21401-6_26]

[5] Gonthier G. Formal proof—The four-color theorem. Notices of the AMS, 2008, 55(11): 1382–1393.

[6] Paulson L. Gödel's incompleteness theorems. https://isa-afp.org/entries/Incompleteness.html

[7] Gowers W. Rough structure and classification. Geometric and Functional Analysis, Special Volume-GAFA, 2000, 79–117. [doi: 10.1007/978-3-0346-0422-2_4].

[8] Hanna G, Yan XK. Opening a discussion on teaching proof with automated theorem provers. For the Learning of Mathematics, 2021, 41(3): 42–46.

[9] Bornat R, Sufrin B. A minimal graphical user interface for the Jape proof calculator. Formal Aspects of Computing, 1999, 11(3): 244–271. [doi: 10.1007/s001650050050].

[10] Hendriks M, Kaliszyk C, Raamsdonk F, Wiedijk F. Teaching logic using a state-of-the-art proof assistant. Acta Didactica Napocensia , 2010, 3(2): 35–47.

[11] Avigad J. Learning logic and proof with an interactive theorem prover. In: Hanna G, Reid D, Villiers M. eds. Proc. of the 2019 Proof Technology in Mathematics Research and Teaching. Mathematics Education in the Digital Era. Springer. 2019: 277–290. [doi: https://doi.org/10.1007/978-3-030-28483-1_13].

[12] Breitner J. Visual theorem proving with the Incredible Proof Machine. Proc. of the Int'l Conf. on Interactive Theorem Proving. 2016. 123139. [doi: 10.1007/978-3-319-43144-4_8].

[13] Lerner S, Foster SR, Griswold WG. Polymorphic blocks: Formalism-inspired UI for structured connectors. Proc. of the 33rd Annual ACM Conf. on Human Factors in Computing Systems. 2015. 3063-3072. [doi: 10.1145/2702123.2702302].

[14] Böhne S, Kreitz C. Learning how to prove: From the Coq proof assistant to textbook style. arXiv:1803.01466, 2018,

[15] Yan S, Yu WS, Fu YS. Formalization of C.T.Yang's theorem in Coq. Ruan Jian Xue Bao/Journal of Software, 2022, 33(6): 2208–2223 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6578.htm [doi: 10.13328/j.cnki.jos.006578].

[16] Sun TY, Yu WS. Mechanized proof of equivalence between the axiom of choice and Tukey's lemma. Journal of Beijing University of Posts and Telecommunications, 2019, 42(5): 17 (in Chinese with English abstract). [doi: 10.13190/j.jbupt.2019-001].

[17] Li Q. On the teaching reform of mathematical logic experiment course for computer majors. Journal of Anhui University of Technology (Social Sciences), 2011, 28(3): 112113 (in Chinese with English abstract).

[18] Jiang N, He YX. Reflections on the teaching of logic and verification courses in computer science. Computer Education, 2021(1): 111115 (in Chinese). [doi: 10.16512/j.cnki.jsjjy.2021.01.027].

[19] Ebbinghaus HD, Flum J, Thomas W, Ferebee AS. Mathematical Logic. Springer, 1994.

[20] Tseitin GS. On the complexity of derivation in propositional calculus. Springer, 1983. 466–483.

[21] Davis M, Putnam H. A computing procedure for quantification theory. Journal of the ACM (JACM), 1960, 7(3): 201–215. [doi: 10.1145/321033.321034].

[22] Thoma A, Iannone P. Learning about proof with the theorem prover LEAN: The abundant numbers task. Int'l Journal of Research in Undergraduate Mathematics Education, 2022, 8(1): 64-93. [doi: 10.1007/s40753-021-00140-1].

[23] Pierce BC, Casinghino C, Gaboardi M, Greenberg M, Hriţcu C, Sjöberg V, Yorgey B. Software foundations. 2010. http://www.cis.upenn.edu/bcpierce/sf/current/index.html

Xinyi Wan, master. His research interest is formalization of mathematical logic.

Qinxiang Cao, Ph.D., associate professor, doctoral supervisor. His research interests include the program verification based on theorem proving and program logic.

Ke Xu, master's degree candidate. His research interests include program verification based on theorem proving.