软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(8):3549−3573 [doi: 10.13328/j.cnki.jos.006868]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

针对教学场景的 ZFC 集合论 Coq 形式化*

万新熠, 徐 轲, 曹钦翔

(上海交通大学 电子信息与电气工程学院, 上海 200240)

通信作者: 曹钦翔, E-mail: caoqinxiang@sjtu.edu.cn

摘 要: 离散数学是计算机类专业的基础课程之一, 命题逻辑、一阶逻辑与公理集合论是其重要组成部分. 教学实践表明, 初学者准确理解语法、语义、推理系统等抽象概念是有一定难度的. 近年来, 已有一些学者开始在教学中引入交互式定理证明工具, 以帮助学生构造形式化证明, 更透彻地理解逻辑系统. 然而, 现有的定理证明器有较高上手门槛, 直接使用会增加学生的学习负担. 鉴于此, 在 Coq 中开发了针对教学场景的 ZFC 公理集合论证明器. 首先, 形式化了一阶逻辑推理系统和 ZFC 公理集合论; 之后, 开发了数条自动化推理规则证明策略. 学生可以在与教科书风格相同的简洁证明环境中使用自动化证明策略完成定理的形式化证明. 该工具被用在了大一新生离散数学课程的教学中, 没有定理证明经验的学生使用该工具可以快速完成数学归纳法和皮亚诺算术系统等定理的形式化证明, 验证了该工具的实际效果.

关键词: Coq; ZFC 公理集合论; 一阶逻辑

中图法分类号: TP311

中文引用格式: 万新熠, 徐轲, 曹钦翔. 针对教学场景的 ZFC 集合论 Coq 形式化. 软件学报, 2023, 34(8): 3549−3573. http://www.jos.org.cn/1000-9825/6868.htm

英文引用格式: Wan XY, Xu K, Cao QX. Coq Formalization of ZFC Set Theory for Teaching Scenarios. Ruan Jian Xue Bao/Journal of Software, 2023, 34(8): 3549−3573 (in Chinese). http://www.jos.org.cn/1000-9825/6868.htm

Coq Formalization of ZFC Set Theory for Teaching Scenarios

WAN Xin-Yi, XU Ke, CAO Qin-Xiang

(School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200240, China)

Abstract: Discrete mathematics is one of the foundation courses of computer science, whose components include propositional logic, first-order logic and axiomatic set theory. Teaching practice shows that it is difficult for beginners to accurately understand abstract concepts such as syntax, semantics and deduction system. In recent years, some scholars have begun to introduce interactive theorem provers into teaching to help students construct formal proofs so that they can understand logical systems more thoroughly. However, existing theorem provers have a high threshold for getting started, directly employing these tools will increase students' learning burden. To address this problem, this study proposes a ZFC axiomatic set theory prover developed in Coq for teaching scenarios. First-order deduction system and ZFC axiomatic set theory are formalized, then several automated reasoning tactics are developed. Students can utilize these tactics to reason formally in a concise, textbook-style proving environment. This tool has been introduced into the discrete mathematics course for freshmen. Students with no prior theorem proving experience can exploit this tool to quickly construct formal proofs of theorems like mathematical induction and Peano arithmetic system, which verifies the practical effect of this tool.

Key words: Coq; ZFC axiomatic set theory; first-order logic

近些年来, 数学定理的形式化证明得到了广泛的发展和研究. 定理证明工具如 Coq[1], Isabelle[2], Mizar[3], Lean[4]等逐渐趋向成熟, 基于这些工具, 大量数学定理的证明得到了形式化, 包括著名的四色定理[5]、哥德

* 本文由"约束求解与定理证明"专题特约编辑蔡少伟研究员、陈振邦教授、王戟研究员、詹博华副研究员、赵永望教授推荐.
收稿时间: 2022-09-05; 修改时间: 2022-10-13; 采用时间: 2022-12-14; jos 在线出版时间: 2022-12-30

尔不完全性定理[6]、Cauchy-Schwarz 不等式等. 菲尔兹奖得主 Gowers 认为, 计算机将在定理证明中起到关键作用, 并将改变理论数学未来的研究模式[7]. Hanna 和 Yan[8]指出: 定理证明器不仅可以帮助数学家进行理论研究, 还能够改变面向数理逻辑初学者的教学方式.

在我国高校计算机类专业的本科课程设置中, 大多数高校院所都将离散数学课程设置为基础类必修课. 而命题逻辑、一阶逻辑与公理集合论是离散数学课程的重要组成部分. 教学实践中发现: 初学数理逻辑的本科学生要学习并明晰语法、语义、推理系统等抽象概念是有一定难度的; 同时, 学生在学习使用一阶逻辑与公理集合论描述数学命题与数学证明时也往往难以通过具体的例子形成直观理解. 在教学中引入交互式定理证明器有望改善这一问题, 计算机辅助的自动检查能够及时准确地反馈学生写的命题是否具有正确的语法、是否遵循逻辑系统的推理规则等情况, 帮助学生构造一个正确且严格的证明, 而无需等待助教可能长达数周的反馈. 交互式定理证明器实时交互的特点, 也能够让学生动态地看到推理过程, 有助于加深对逻辑系统的理解.

近年来, 国际上越来越多的学者开始在数理逻辑课程, 特别是计算机专业的相关课程中引入定理证明工具来辅助教学. 例如: 牛津大学开发了 Jape 用于谓词逻辑的教学[9], Cezary 和 Freek 基于 Coq 开发了在线教学工具 ProofWeb[10], Avigad 使用 Lean 进行一阶逻辑和集合论的概念的教学[11]. 然而, 以上工作大都使用了定理证明器的内置逻辑, 这在教学中会有很多缺陷.

(1) 大部分定理证明工具如 Coq 采用了“自底向上”的反向证明模式, 用户不断应用证明策略将待证目标分解为一系列子目标, 最终得到所有子目标均为前提完成证明. 而常用证明模式还包括从前提推导出新条件最终得出待证目标的正向推理;

(2) 学生应当根据当前学习的逻辑系统的推理规则进行推理, 而不是使用定理证明器中内置的逻辑, 偏离教学目标;

(3) 直接使用定理证明器进行证明更像是学习一门编程语言, 学生需要额外付出大量时间熟悉操作该特定定理证明器的命令, 不仅增加学习成本, 而且容易忽视逻辑系统本身;

(4) 由于仅提供了基本的证明策略, 证明代码通常十分冗长, 较大的工作量限制了学生完成更加复杂定理的形式化证明;

(5) 缺乏简洁的教科书风格的符号系统, 不利于学生理解.

简而言之, 以辅助数学理论研究为目的开发的定理证明工具具有较高的上手门槛, 在形式上与教科书中的逻辑系统差别较大, 简单地直接引入教学中反而可能会加大学习成本, 甚至使学生将定理证明器的内置逻辑与他正在学习的逻辑系统相混淆.

考虑到这些问题, 部分学者没有使用定理证明器, 转而使用编程语言从头开发专用于教学的简易证明工具. Breitner[12]开发了 Incredible Proof Machine[12], 学生在可视化界面中使用鼠标将代表命题和一阶逻辑推理规则的方块连接起来就可以构造证明. Lerner 等人提出了 Polymorprphic block[13]可视化界面, 学生可以采用类似于拼拼图的方式将命题和推理块连接起来, 完成自然演绎系统的推理. 这些工具能够提供交互更简便、趣味性更强的体验, 但从头开发需要极大的工作量. 现有的交互式定理证明工具已经提供了完善的用户界面, 包括证明界面、证明状态和错误信息, 基于定理证明器开发教学工具能够极大减少开发成本, 例如, 本工作仅花费了 2 300 行 Coq 代码, 在教学中是更为实际的选项; 同时, 针对不同教学场景进行调整也更加简便.

作为辅助逻辑学习的工具, 需要降低定理证明器中的形式化证明与教科书中逻辑系统证明间的差距, 使学生专注于他所学习的逻辑系统而非定理证明器的使用技巧. 一款合格的基于定理证明概念的教学工具应当具有以下特点.

(1) 支持的推理模式应当与传统逻辑教材中使用的推理模式相同;

(2) 使用与教学内容相同的逻辑系统, 所有操作都基于该系统的推理规则, 尽可能少地暴露定理证明器的内置逻辑. 此外, 应当使用教科书上的逻辑术语和符号, 而非定理证明器中新创建的变量名;

(3) 只需要简单的几条指令就可以完成证明, 降低学生使用定理证明器的学习成本. 这几条证明指令需

要具有较高的自动化程度, 从而减少代码量, 帮助学生完成更复杂命题的严格证明.

针对上述问题, 本文在 Coq 中开发了一个包含自动化证明策略的 ZFC 公理集合论证明器, 用于辅助一阶逻辑和公理集合论的教学. 本文在 Coq 中从抽象语法树开始形式化了基于一阶逻辑的 ZFC 公理集合论, 采用 sequent calculus 式的逻辑系统, 并利用 notation 机制提供了简洁的符号. 我们开发了数条自动化证明策略以简化证明过程, 包括命题逻辑自动证明指令 FOL_Tauto, 该指令将可证问题转化为合取范式的布尔可满足性问题, 并调用我们在 Coq 中实现的 DPLL 可满足性求解器自动求解, 实现了逻辑连接词的完全自动化处理. 对于量词推理规则, 我们分别开发了指令以支持多层量词的同时引入和消去; 对等号替换规则开发了指令 peq_sub_tac 自动替换命题中的一阶逻辑项. 以上所有指令均能自动处理α-等价, 且采用了与教材相同的正向推理模式. 通过以上方法, 我们构造了一个简洁快速的集合论证明环境, 隐藏了 Coq 本身的诸多细节, 更贴近教科书风格的证明且更加自动化. 学生使用该工具进行集合论定理的证明相比直接使用现有的定理证明器更为容易. 图 1 给出了一个证明示例, 两个集合的交集必然是其中一个集合的子集.

```
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 proved.
Qed.
```

(a) 证明代码

```
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) 最后一步之前的 Coq 证明状态栏

图 1 ZFC 集合论证明器证明环境示例

我们在离散数学课程的实际教学中引入了该工具, 并取得了良好效果. 作为课程项目之一, 学生需要使用该证明器完成数学归纳法的证明. 实践表明, 首次接触一阶逻辑和公理集合论的大一新生经过简单学习后就可以完成该定理的形式化证明. 较为优秀的学生可以在完成离散数学课程后, 利用该工具进一步构造皮亚诺算术系统并证明了加法和乘法的相关性质, 体现了该工具的简洁性和有效性.

本文第 1 节介绍研究背景, 包括 Coq 交互式定理证明器和基于定理证明器的逻辑教学工具研究现状. 第 2 节阐述 ZFC 集合论证明器的设计框架. 第 3 节给出对 ZFC 公理集合论的形式化. 第 4 节介绍自动证明策略的实现. 第 5 节展示我们在离散数学课程教学中引入该证明器的实际效果. 最后总结全文.

1 研究背景

1.1 基于定理证明器的逻辑教学工具研究现状

引入定理证明器辅助教学的主要问题在于: 定理证明器在形式上与教科书中的逻辑系统有较大差距, 如何缩小这一差距成为主要研究方向. 一种思路试图在证明形式上逐渐使定理证明器的代码证明向教科书证明靠拢. Böhne 等人[14]在 Coq 中为每一条推理规则提供了一条证明指令, 要求学生首先在 Coq 中完成命题的证明, 之后在每一行证明代码前后通过注释写出此时对应的待证一阶逻辑命题, 最后要求学生根据 Coq 证明写出与其结构相同的、教科书风格的纸笔证明, 这样的证明具有与 Coq 相同的反向推理模式. 实践表明, 学生能够完成并理解 Coq 中的证明与逻辑系统之间的对应关系. Avigad[11]使用 Lean 的内置逻辑进行一阶逻辑和集合论的教学, 并让学生比较理解证明代码和纸笔证明. 在系统学习 Lean 的使用方法后, 学生也能够成功完成证明. 另一种思路试图直接开发与教科书风格更相近的定理证明工具. 早在 20 世纪末, 牛津大学开发的 Jape[9]就提供了良好的可视化界面, 可以将证明显示成树状或盒状的形式. 受此启发, Kaliszyk 等人基于 Coq 开发了 ProofWeb[10], 其采用了 Gentzen 风格的自然演绎系统, 并提供了轻量化的在线 GUI, 学生同样通过应用每一条推理规则对应的指令自底向上进行证明.

国内的集合论相关研究大多关注重要集合论定理的形式化, 如杨忠道定理[15]、Tukey 引理[16]等, 目前, 对定理证明在教学中的应用研究较少, 且都处于讨论和探索阶段. 李沁[17]探讨了在计算机专业的数理逻辑课程中使用 Isabelle 辅助教学的可行性, 江南等人[18]则比较了慕尼黑工业大学与国内高校有关逻辑与验证相关课程的开设情况, 并呼吁计算机专业的理论课程和实践教学中需要设置更多的定理证明相关内容. 据我们所知, 本文是国内首次将交互式定理证明实际应用在面对数理逻辑初学者的课堂教学中.

1.2 Coq交互式定理证明器

Coq 是一个交互式定理证明工具, 具有强大的表达能力和优秀的扩展性, 是国际上主流的证明助手之一, 在数理逻辑、算法和高可靠性软件的形式化验证方面得到了广泛应用. Coq 的理论基础是归纳构造演算, 为用户提供归纳类型, 比大多数函数式编程语言中的归纳类型更具表达力. 用户可以在 Coq 中形式化定义数学概念、程序语言、逻辑系统等, 并证明相关的性质和定理. 在证明时, 用户首先通过 Proof 指令开始证明, 之后, 应用一系列证明策略与 Coq 进行交互, 证明策略会根据当前证明状态以及已经得到证明的定理、公理等将待证目标分解为数个更简单的子目标或者直接证明目标, 而 Coq 证明检查器中的类型检查算法会根据归纳构造演算的性质机械化地进行检查. 证明完成后, 用户运行 Qed 指令退出证明. 表 1 列出了 Coq 中的一些常用证明策略及其功能.

表 1 Coq 常用证明策略

Coq 证明策略	功能
intros	引入待证目标中的条件
destruct	根据构造子进行分类讨论
induction	根据构造子进行结构归纳
apply	对待证目标应用一个假设或已证定理
eapply	应用时无需指定变元, 推迟变元的实例化
pose proof	在条件中引入一个已证定理
assert	声明一个新的待证目标
assert … by …	声明新待证目标并由 by 后的策略直接解决
rewrite	根据等式重写
congreunce	等式重写自动证明策略
reflexivity	自动比较待证目标等号两侧是否相同
tauto	直觉逻辑自动证明策略

Coq 允许用户通过 Ltac 语言组合现有的证明策略构成新策略, 简化证明过程, 提高证明的自动化程度. 本文所开发的自动证明策略就依赖 Ltac 语言实现, 学生在使用中不需要学习 Coq 自带的证明指令语言, 而只需要使用本文所开发的 7 条自动证明指令, 其中一条与分离公理相关的证明策略没有出现在课程项目中, 本文不做介绍.

2 公理集合论证明器的设计框架

介绍 ZFC 定理证明器的实现框架之前, 需要特别说明, 该定理证明器是针对特定的数理逻辑教学场景开发的. 教学的对象是已经掌握了命题逻辑的学生, 并且当前阶段的教学重点是 sequent calclulus 式的逻辑系统中的 4 条量词推理规则以及 ZFC 公理集合论与日常数学之间的联系. 因此, 我们形式化 ZFC 公理集合论时采用了与教材相同的 sequent calculus 系统, 开发证明策略时, 将所有命题逻辑相关的证明完全自动化地集成在了证明策略 FOL_Tauto 中, 而为每一条量词推理规则提供了单独的证明策略. 此外, 我们引入了符号二元交∩、二元并∪、单元集{·}作为一阶逻辑中的函数符号以及空集∅作为常量符号. 通常, 教材中的 ZFC 集合论并不包含这些符号, 但引入这些符号不会改变整个推理系统的可靠性, 具体证明可见教材[19]. 尽管该定理证明工具在 Coq 中实现, 但本文目标是提供一个教学用的定理证明工具而非被验证的证明器, 因此我们未验证其中涉及到的求解器等模块的正确性.

该证明器的设计分为两个部分: ZFC 公理集合论的形式化和交互式定理证明环境. 两部分均在 Coq 中实现, 详情如图 2 所示.

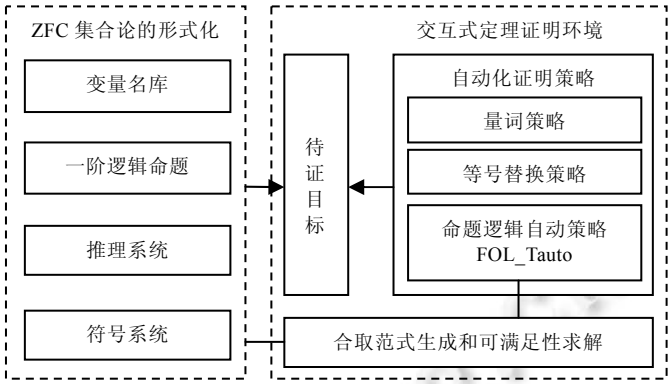

图 2 ZFC 集合论证明器设计框架

第 1 部分在 Coq 中形式化定义 ZFC 公理集合论的相关概念, 构造可靠的推理系统, 包含以下 4 个方面.

(1) 显式变量名库 *StringName*. 我们没有使用 Coq 内置的变量名系统, 而是基于 Coq 原有的 *String* 字符串库为该推理系统专门开发了显式变量名库 *StringName*. 该库允许我们对变量名有更好的控制, 可以自行定义新变量名的引入、变量名的比较等方法, 有利于之后对替换、α-等价等语法操作的自动化. 此外, 从基于字符串的抽象语法树开始形式化逻辑系统, 还能够避免在证明环境中暴露 Coq 的内置逻辑. 如图 1(b)所示: 我们的证明环境中只会显示一系列命题可证的条件, 而不会有形如 *x*:*String* 的条件, 证明开始时, 也无需对各个变量进行 intros 引入操作, 保持与教科书风格证明高度一致;

(2) 一阶逻辑命题. 一阶逻辑的研究对象是一阶逻辑命题, 需要在 Coq 中对命题和项进行形式化;

(3) ZFC 集合论推理系统. 本文的推理系统采用与教材中相同的 sequent calculus 逻辑系统, 证明过程中的每一步都由多个前提和一个结论组成. 推理系统只涉及语法结构, 因此, 本文仅形式化了可证性质 derivable, 没有形式化该系统的语义;

(4) 符号系统. 定理证明器的证明代码通常可读性较差, 不易于学生理解, Coq 的 notation 机制能够很好地解决这一问题. 例如: 实质蕴涵的形式化定义为 *PImpl P Q*, 若定义 Notation “*P*1→*P*2”:=(*PImpl P*1 *P*2), 之后的证明中就可以将其简写为“*P*→*Q*”.

第 2 部分为交互式定理证明环境, 待证定理由之前形式化的命题构成. 用户使用我们提供的自动证明策略, 决定如何应用推理规则. 本文开发的指令全部采用与教材相同的正向推理模式, 每当用户成功应用一条指令, 证明环境中就会根据当前的条件加入由推理规则产生的新推导关系. 当结论可以通过现有前提进行命题逻辑推理得到时, 用户最后调用 The conclusion is already proved 指令(FOL_Tauto 的别名, 用于结束证明)来完成定理的证明. 用户无需除了 pose proof 外的任何 Coq 内置证明策略, 自动证明策略的具体实现在第 4 节中详细阐述.

3 ZFC 集合论的形式化

3.1 显式变量名库

利用 Coq 中的模块机制, 我们首先实现了命名系统模块 NAME_SYSTEM_EXT, 该模组接受如表 2 所示的参数.

基于这些参数, 可以进一步定义其他 Coq 函数并证明相关性质.

定义根据名字列表引入新名字的函数 *list_new_name*, 返回列表中最大名字的下一个名字.

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

表 2 NAME_SYSTEM_EXT 模块参数

参数及需要满足的性质	描述
t: *Type*	名字的类型
max: $t\to t\to t$	比较并返回较大名字的函数
next_name: $t\to t$	返回输入的下一个名字
le: $t\to t\to Prop$	小于等于二元关系
default: t	默认名字
eq_dec: $\forall v_1, v_2$: t, $\{v_1=v_2\}+\{v_1<>v_2\}$	名字具有可比性, 要么相同要么不同
Transitive *le*	小于等于关系具有传递性
$\forall u, v$, *le* u, $v\to u \neq$ *next_name* v	任何名字的下一个名字都比自身大
$\forall v_1, v_2$, *le* v_1 (*max* v_1, v_2)	*max* 函数正确性 1, 比第 1 个参数大
$\forall v_1, v_2$, *le* v_2 (*max* v_1, v_2)	*max* 函数正确性 2, 比第 2 个参数大
$\forall v_1, v_2$, *max* v_1, v_2=*max* v_2, v_1	*max* 函数交换律
$\forall v_1, v_2, v_3$, *max* v_1 (*max* v_2, v_3)=*max* (*max* v_1, v_2) v_3	*max* 函数结合律

实际使用时, 只需要实例化表 2 中的各项参数, 并证明实例化后的类型和函数满足表 2 中的性质, 就可以得到用户指定类型的命名系统模块. 我们将参数 t 实例化为 *String* 库中的 *string* 类型, *max* 和 *le* 使用字符串的字典序, *next_name* 实例化为在输入字符串后加上后缀“”, *default* 实例化为字符串“x”, 得到字符串模块 *StringName*. 上述类型符合表 2 性质的证明并非本文关键, 此处不列出.

在后续开发中, 会以不同的名字引用 *StringName* 库作不同用途. 在推理系统中, 令 *V*:=*StringName*, 使用 *V.t* 作命题中变量名的类型; 而在 DPLL 求解器中, 令 *PV*:=*StringName*, 使用 *ident*:=*PV.t* 作为命题变元的类型. 虽然两者本质上均为字符串类型, 但采用不同名称以避免混淆.

此外, 我们为常用字符串分配了 Coq 内置变量名, 例如:

Definition x:=*EVAL* “x”%*string*;

Definition $x0$:=*EVAL* (*V.next_name* x).

这样, 在证明环境中就会直接显示 x 等字母而非带引号的字符串, x 的下一个名字也可以依次显示为 $x0$, $x1$,$x2$,…, 如图 1 所示.

3.2 命题及其相关性质的形式化

一阶逻辑的基本研究对象是一阶逻辑项, 在 ZFC 公理集合论中, 项代表一个集合, 归纳定义如下:

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

$\varnothing$表示空集常量, x 为变量, $\{t\}$表示单元集, $\cap$和$\cup$分别为二元交和二元并. 使用 Coq 的 *Inductive* 关键字, 项的形式化定义 *term* 为:

1. *Inductive term*:=;
2. |*var*(v:*V.t*)|*empty_set*|*singleton*(x:*term*)|*union*(x y:*term*)|*intersection*(x y:*term*),

其中, *empty_set* 构造子对应空集常量, *var* 构造子对应字符串表示的变量, *singleton* 对应单元集, *union* 和 *intersection* 对应二元并和二元交. 之后, 使用 notation 进行如下简写, 使 Coq 中的符号与式(1)一致. 为保证所作符号不与 Coq 内置的符号冲突, 新定义的命题符号只在双括号“[[ ]]”内有效.

1. Notation “[[e]]”:=e(at *level*0, e custom set at *level*99);
2. Notation “$\varnothing$”:=*empty_set* (in custom set at *level*5, no associativity);
3. Notation “$x\cap y$”:=(*intersection* x y) (in custom set at *level*11, left associativity).

其余简写方式类似.

命题的归纳定义为

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

其中, $t_1=t_2$ 和 $t_1\in t_2$ 为原子命题, 表示集合的相等和属于关系; $\top$和$\bot$分别表示真命题和假命题; 其余构造子分别对应取否、合取、析取、实质蕴涵、当且仅当、全称量词和存在量词. 命题的形式化定义 *prop* 如下.

1. *Inductive prop*: *Type*:=;

2. |*PEq*(t_1 t_2:*term*)|*PRel*(t_1 t_2:*term*)|*PFalse*|*PTrue*;
3. |*PNot*(*P*:*prop*)|*PAnd*(*P Q*:*prop*)|*POr*(*P Q*:*prop*)|*PImpl*(*P Q*:*prop*)|*PIff*(*P Q*:*prop*);
4. |*PForall*(*x*:*V*.*t*)(*P*:*prop*)|*PExists*(*x*:*V*.*t*)(*P*:*prop*).

在 Coq 中, 同样可以使用 notation 给出与式(2)相同的记号.

接下来给出对命题中自由变元进行替换的方法, 定义替换目标σ为变量名和项的有序对构成的列表, 记录每一个变量要被替换成的项, 对命题 P 作替换σ的结果记作 $P[\sigma]$:

$$
\begin{array}{ll}
\varnothing[\sigma] ::= \varnothing & x[\cdot] ::= x \\
x[x \mapsto t;;...] ::= t & x[y \mapsto t;;\sigma'] ::= x[\sigma'] \\
(t_1 \cap t_2)[\sigma] ::= t_1[\sigma] \cap t_2[\sigma] & \\
\top[\sigma] ::= \top & (t_1 = t_2)[\sigma] ::= t_1[\sigma] = t_2[\sigma] \\
(P_1 \wedge P_2)[\sigma] ::= P_1[\sigma] \wedge P_2[\sigma] & \\
(\forall x,\ P)[\sigma] ::= \forall x,\ P[\sigma], & \text{if } x \notin V(\sigma) \\
(\forall x,\ P)[\sigma] ::= \forall u,\ P[x \mapsto u;\sigma], & \text{if } x \in V(\sigma)
\end{array}
$$

对项作替换时, 空集常量在替换前后不变, 变量 x 会从头依次比较替换目标中直到找到 x 对应的项 t, 否则不会替换, 单元集和二元交并分别替换子项. 对命题作替换时类似, 值得关注的是量词的情况, 此时需要检查被量词绑定的变量是否在替换目标中出现: 若没有出现, 继续替换子命题; 否则, 需要引入一个当前环境中没有的新变量 u, 并将 $x \mapsto u$ 加入替换目标前, 将子命题 P 中的 x 先替换成 u 以避免冲突. 以下列出其形式化定义 *prop_sub* 关键的量词部分, *subst_task_occur* 判断名字在替换目标中出现的次数, 函数 *new_var* 负责引入新变量名, 其定义基于第 3.1 节中的 *list_new_name*.

```
1.  Fixpoint prop_sub (st:subst_task) (d:prop): prop:=
2.    match d with
3.    |[[∀x,P]]⇒match subst_task_occur x st with
4.      |O⇒PForall x (prop_sub st P)
5.      |_⇒let x′:=new_var P st in
6.        PForall x′ (prop_sub (cons (x, var x′) st) P)
7.      end
8.    |…
9.    end.
```

命题的另一重要语法性质是α-等价, 它描述了对量词绑定变量的重命名, 如$\forall x$, $x=x$ 和$\forall y$, $y=y$, α-等价的命题具有相同的语义. α-等价的定义通常被描述为多次对量词绑定变量的重命名, 即$\forall x$, $P=_\alpha \forall y$, $P[x \mapsto y]$, 但基于替换的定义不利于自动化, 本文提出了一种带环境的、不依赖替换的归纳定义.

环境θ是一个三元组列表:

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

其中, u, v 为变量名, 记录两个命题中被量词约束的变量的对应关系; b 是布尔值, true 表示该对应关系当前有效, false 表示该对应已被新对应覆盖, 因此无效.

环境θ记录了两个命题自顶向下检查量词对应关系时经过的路径, 以下先给出一阶逻辑项的带环境α-等价$=_\theta$的部分归纳定义:

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

$A_\varnothing$规则表示在任何环境θ下, 空集都只与自身等价; *ABind* 规则处理约束变量, 如果 u 和 v 在环境中存在有效对应, 则 u 与 v 等价; *AFree* 规则处理自由变量, 如果变量 s 不在环境中, 那么该变量自由出现, 只能和自身等价; 二元交判断子项是否各自等价; 单元集和二元并的情况类似.

基于项的α-等价, 可以给出命题的带环境α-等价, 以下为部分典型规则:

$$A_{\in}\frac{t_1=_\theta t_2\ \ t_3=_\theta t_4}{t_1\in t_3=_\theta t_2\in t_4},\ A_{\forall}\frac{P=_{(x,y,\mathrm{true});\theta\backslash(x,y)}Q}{\forall x,\ P=_\theta \forall y,\ Q},\ A_{\top}\frac{}{\top=_\theta\top},\ A_{\wedge}\frac{P_1=_\theta Q_1\ \ P_2=_\theta Q_2}{P_1\wedge P_2=_\theta Q_1\wedge Q_2}.$$

量词对应的规则是唯一会修改环境的规则, 在自顶向下的检查过程中, 若遇到$\forall x,P$ 和$\forall y,Q$, 此时有 P 中的 x 与 Q 中的 y 对应, 因此需要将(x,y,true)记录到环境中再检查子命题, 而θ中原先左侧为 x 或右侧为 y 的记录(相当于外层量词的对应)在子命题 P 不再有效, 需要置为 false, 用$\theta\backslash(x,y)$表示.

在最外层环境为空, 可以自然得到原始α-等价的定义 $P=_\alpha Q:=P=_{[\cdot]}Q$. 容易看出, 这一过程是可判定的, Coq 中将判定α-等价的过程形式化为返回布尔值的递归函数 *alpha_eq*, 以便于自动计算.

```
1. Fixpoint alpha_eq(l:binder_list)(P Q:prop):bool:=
2.     match P, Q with
3.     |[[t1∈t2]],[[t3∈t4]]⇒term_alpha_eq l t1 t3 && term_alpha_eq l t2 t4
4.     |[[P1∧P2]],[[Q1∧Q2]]⇒alpha_eq l P1 Q1 && alpha_eq l P2 Q2
5.     |[[∀x, P1]],[[∀y, Q1]]⇒alpha_eq((x,y,true)::(update x y l)) P1 Q1
6.     |…
7.     |_,_⇒false
8.     end.
9. Definition aeq(P Q:prop):bool:=alpha_eq nil P Q.
```

3.3 推理系统的形式化

推理系统采用 sequent calculus 形式的逻辑系统, 证明中的每一步是一个命题序列$\varphi_1\ \varphi_2\ldots\varphi_n\ \varphi$, 其中, 前 n 个命题为前提, 末尾的命题φ为结论, 记作$\varphi_1\ \varphi_2\ldots\varphi_n\vdash\varphi$. 之后, 使用$\Gamma$表示命题序列, φ表示命题, $\Gamma\ \varphi$表示在命题序列Γ后增加命题φ构成新的命题序列. 推理规则描述了证明过程中的一步, 它接受多个条件并得到一个新的公式序列作为结论. 本文采用的推理规则全部来自实际教学中使用的逻辑教材[19], 此处选择性列出一部分, 包括:

- 有关逻辑连接词的推理规则:

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

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

- 有关量词的 4 条推理规则, 其中, $FV(\varphi)$表示命题φ中所有自由出现的变量构成的集合:

$$\forall_{Elim}\ \frac{\Gamma\vdash\forall x,\ P}{\Gamma\vdash P[x\mapsto t]},\ \exists_{Intro}\ \frac{\Gamma\vdash P[x\mapsto t]}{\Gamma\vdash\exists x,\ P},\ \forall_{Intro}\ \frac{\forall\varphi\in\Gamma,\ x\notin FV(\varphi)\ \ \Gamma\vdash P}{\Gamma\vdash\forall x,\ P},$$

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

- 有关等号的推理规则:

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

- ZFC 集合论公理:

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

$$Infinity\ \frac{}{\vdash\exists x,\ \varnothing\in x\wedge\forall y,\ (y\in x\to y\cup\{y\}\in x)},\ Separation\ \frac{x\notin FV(P)\ y\notin FV(P)}{\vdash\forall x,\ \exists y,\forall z,\ z\in y\leftrightarrow z\in x\wedge P};$$

- 集合相关符号的公理:

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

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

在 Coq 中, 将前提和结论构成的公式序列分开进行形式化, 结论命题即为上一节中的 *prop* 类型, 前提定义为 *context*:=*prop*→*Prop* 类型, 即命题的集合. 之后, 可以进一步定义不包含任何命题的空前提, 简写为 ZF; 以及在前提后加入新命题的方法, 简写为Γ;;φ.

1. *Definition empty_context*:*context*:=*fun*_⇒False;
2. Notation "'*ZF*'":=*empty_context* (in custom set at *level*20);
3. Notation "*Phi*;;*x*":=(*Union_Phi*(*Singleton_x*)) (in custom set at *level*31, left associativity).

使用 Inductive 关键字, 可以形式化地归纳定义上述推理规则, 得到可证关系 derivable, 其类型为 *context*→*prop*→*Prop*, "derivable Γ φ"即形式化地表示一阶逻辑命题φ可由前提Γ经过推理规则推导得到. 在 Coq 中, 将 derivable 简记为⊢:

Notation "*Phi*⊢*P*":=(derivable *Phi P*) (in custom set at *level*41, no associativity).

接下来, 以$\forall_{Intro}$规则、外延公理和无穷公理展示 derivable 的形式化定义.

(1) $\forall_{Intro}$规则可以形式化地表示为:

1. |*PAnd_intros*: forall *Phi P Q*;
2. derivable *Phi P*→;
3. derivable *Phi Q*→;
4. derivable *Phi* [[*P*∧*Q*]].

如果证明时 Coq 的前提条件中分别有 *P* 和 *Q* 能够从同一个前提 *Phi* 推导得到, 那么可以进一步得到 *Phi* 能够推导出[[*P*∧*Q*]].

(2) 外延公理可以形式化地表示为

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

外延公理的形式化具有与教材中的公理完全相同的形式. 我们直接将公理中的命题作为结论, 此处 *x*, *y* 等变量名为第 3.1 节中已定义的字符串, 因此, 形式化时不需要通过 forall 来引入 Coq 中的变量. 证明时, 通过 pose proof Extensionality 指令引入外延公理, Coq 证明状态中就会直接加入新条件[[*ZF*⊢∀*x*, ∀*y*, (∀*z*,*z*∈*x*↔*z*∈*y*)↔*x*=*y*]]. 之后, 可以应用消去全称量词的证明策略将 *x*, *y* 实例化成实际需要的项继续证明.

(3) 形式化无穷公理时, 我们希望能将目标集合是归纳集这一命题简写成谓词的形式.

定义带参数 *t* 的命题 *is_inductive_def* 为: Definition *is_inductive_def*(*t*:*term*):=[[(∅∈*x*∧∀*y*, (*y*∈*x*→*y*∪{*y*}∈*x*))[*x*↦*t*]]], 表示空集在集合 *t* 中, 并且 *t* 中任意元素的后继都在集合 *t* 中. 引入 *x*↦*t* 的替换操作是为了在当参数 *t* 为变量 *y* 时, 利用替换将原命题中全称量词绑定的 *y* 重命名成其他变量, 避免出现∀*y*, *y*∈*y*→*y*∪{*y*}∈*y* 的错误命题.

之后, 使用 notation 将 *is_inductive_def t* 在命题中简记为 *is_inductive t*:

Notation "'*is_inductive*' *t*":=(*is_inductive_def t*) (in custom set at *level*20, t_1 at *level*15, no associativity).

无穷公理可被形式化表示为

|Infinity: derivable *empty_context*[[∃*x*, *is_inductive x*]].

证明时, 通过 pose proof Infinity 指令即可引入条件[[*ZF*⊢∃*x*, *is_inductive x*]].

4 自动证明策略

本节介绍我们为用户提供的自动证明策略, 在证明环境中, 用户只需使用我们开发的证明策略和 Coq 内置策略 pose proof 进行证明, 这些策略都只会在证明环境中加入新条件而不会修改待证目标, 符合教材中的

正向推理模式. 表 3 列出了学生证明时需要的全部证明策略及其功能, 具体使用效果可见第 5.1 节, 其中详细对照了 Coq 证明与教材中的对应形式化证明.

表 3 学生使用的全部证明策略及其功能

证明策略	功能
assert ... by FOL_Tauto	判断用户声明的推导关系能否根据现有条件使用命题逻辑得到
The conclusion is already proved.	FOL_Tauto 的别名, 用于证明最后一步
universal instantiation	使用全称量词消去规则$\forall_{Elim}$实例化指定结论前的多个全程量词
universal generalization	使用全程量词引入规则$\forall_{Intro}$在指定结论前引入多个全称量词
existential instantiation	使用存在量词规则$\exists_{Elim}$在指定前提前引入多个存在量词
existential generalization	使用存在量词引入规则$\exists_{Intro}$在指定结论前引入多个存在量词
peq_sub_tac	引入一个新的等号替换关系
pose proof (Coq 内置)	引入一个已证定理

4.1 命题逻辑自动证明策略FOL_Tauto

一阶逻辑通常是不可判定的, 意味着无法找到有效的算法来判断某一命题是否能够由一组命题推导得到. 但通过将包含量词的一阶逻辑命题抽象成命题变元, 可以将一阶逻辑转化成可判定的命题逻辑. 再通过命题的有效性与其否命题的不可满足性等价, 将问题转化为可满足性求解问题, 使用求解器实现逻辑连接词的自动化处理. 基于这一思路, 开发了命题逻辑自动证明策略 FOL_Tauto, 本节将具体阐述该策略的实现. 需要注意: 尽管求解器在 Coq 中实现, 但其正确性并未被验证.

(1) 命题变元预处理

在求解之前, 需要将一阶逻辑中的原子命题和含量词的命题抽象成命题变元, α-等价的命题具有相同的语义, 因此需要将其抽象为同一个命题变元. 我们首先在 Coq 中定义了一个简单的键值对结构 *prop_table* 用来记录命题对应的变元, 使用列表储存命题和对用的 *ident*(即命题变元类型, 见第 3.1 节)的二元组. 调用 *prop_look_up* 函数查找时, 会从头依次比较输入的命题与当前的键是否α-等价: 若等价, 则返回对应变元; 否则, 返回 *None*.

之后, 定义命题逻辑命题类型 *sprop*:

1. *Inductive sprop*: *Type*:=;
2. |*SId*(*x*: *ident*)|*SFalse*|*STrue*|*SNot*(*P*: *sprop*)|*SAnd*(*P Q*: *sprop*)|*SOr*(*P Q*: *sprop*)|*SImpl*(*P Q*: *sprop*).

其中, *SId* 构造子表示原子的命题变元; 其余构造子对应真假命题和常规的逻辑连接词, 含义与一阶逻辑命题 *prop* 中的连接词一致. 为行文方便, 后文中简写为与式(2)相同的记号. 注意: *sprop* 没有连接词↔, 转换时, 会将其展开为两个实质蕴涵的合取.

容易定义一阶逻辑命题 *prop* 到命题逻辑命题 *sprop* 的递归转换函数 *sprop_gen*. 该函数接收 3 个参数, 分别为待转换命题 P、记录已分配命题及变元的 *prop_table* KV 以及可用的命题变元字符串 s. 返回 3 个值, 分别为 P 的转换结果 P'、修改后的 *prop_table* KV'以及新的可用命题变元 s'. *sprop_gen* 的定义(简写为 *sprop*)如下, 在不关键的情况下省略参数 s, 未列出的情况类似:

$$
\begin{aligned}
&sprop(P,KV) ::= KV(P), KV,\ \text{if}\ prop\_look\_up(P,KV) \neq None\\
&sprop(\top,KV) ::= STrue, KV\\
&sprop(t_1 \in t_2, KV, s) ::= SId\ s,\ (t_1 \in t_2 \mapsto s;;KV),\ next\_name(s)\\
&sprop(\forall x.P,\ KV, s) ::= SId\ s,\ (\forall x, P \mapsto s;;KV),\ next\_name(s)\\
&sprop(P \wedge Q,\ KV, s) ::= \text{let}\ P', KV', s' := sprop(P, KV, s)\ \text{in}\\
&\qquad\qquad \text{let}\ Q', KV'', s'' := sprop(Q, KV', s')\ \text{in}\\
&\qquad\qquad P' \wedge Q',\ KV''
\end{aligned}
$$

若 KV 中已经为与 P 等价的命题被分配了命题变元, 那么 *sprop_gen* 直接返回该命题变元. 如果 KV 中没有分配, 则需要根据待转换命题的结构进行分类讨论: 对于真命题$\top$, 直接对应转换成 *STrue* 即可, 此时无需修改 KV 和 s; 对于原子命题或者量词开头的一阶逻辑命题, 将当前命题抽象为命题变元 s, 将这一对应记录在

KV 中, 并将 *s* 的下一个名字作为新的可用命题变元; 处理二元逻辑连接词时, 先对左侧子命题做转换, 再根据转换完的 *KV′*和可用变元 *s′*对右侧子命题做转换, 最终使用相应的逻辑连接词连接两个子命题各自的转换结果.

(2) 合取范式生成

为了提高效率, 现代可满足性求解器会将命题转化为合取范式形式再进行求解. 合取范式可以看作一系列析取子句的合取, 每个析取子句由一系列命题变元或其否定形式(称为 literal)合取而成. 下式给出了子句类型 *clause* 和合取范式类型 *CNF* 的形式化定义. 析取子句用布尔值和命题变元构成的二元组列表表示, 布尔值为 true 表示命题变元自身, false 表示其否定形式. 合取范式 *CNF* 则用析取子句 *clause* 的列表表示.

1. Definition *clause*:=*list*(*bool***ident*);
2. Definition *CNF*:=*list clause*.

任何命题在不引入新命题变元的情况下都可以转换成逻辑等价的合取范式形式, 但会导致逻辑连接词数量的指数级增长. 在引入新变元的情况下, 能够在保持可满足性不变而非逻辑等价的条件下生成线性倍数大小的合取范式. 最简单的此类方法是 Tseitin 算法[20], 它为每一个子命题引入一个新变元. 本文给出了一个避免引入过多新变元的合取范式 *CNF* 生成方法 *cnf_gen*(简写为 *cnf*). 该函数采用后继风格, 接收 3 个参数: 待生成的 *sprop* 命题 *P*、可用命题变元 *s* 和已生成的合取范式φ. 返回两个值, 分别为 *P* 的转换结果合取φ后的新合取范式和新的可用命题变元 *s′*. 不关键时, 省略第 2 个参数 *s*:

$$
\begin{aligned}
&cnf(SId\ x,s,\varphi) ::= (x)\wedge\varphi,s\\
&cnf(\top,s,\varphi) ::= \varphi,s\\
&cnf(\bot,s,\varphi) ::= (impossible)\wedge(\neg impossible),s\\
&cnf(P\wedge Q,s,\varphi) ::= \text{let } \varphi',s := cnf(P,s,\varphi) \text{ in } cnf(Q,s,\varphi')\\
&cnf(P\vee Q,s,\varphi) ::= \text{let } \tau,\varphi',s' := clause(P,s,nil,\varphi) \text{ in}\\
&\qquad\qquad \text{let } \tau',\varphi'',s'' := clause(Q,s',\tau,\varphi') \text{ in}\\
&\qquad\qquad\quad \tau\wedge\varphi',s\\
&cnf\_gen(P\rightarrow Q,s,\varphi) ::= \text{let } \tau,\varphi',s' := neg\_clause(P,s,nil,\varphi) \text{ in}\\
&\qquad\qquad \text{let } \tau',\varphi'',s'' := clause(Q,s',\tau,\varphi') \text{ in}\\
&\qquad\qquad\quad \tau\wedge\varphi'',s\\
&cnf\_gen(\neg P) ::= \text{let } \tau,\varphi',s' := neg\_clause(P,next\_name(s),s,\neg n\wedge\varphi) \text{ in}\\
&\qquad\qquad \tau\wedge\varphi',s'
\end{aligned}
$$

若待生成命题 *P* 为单一命题变元, 则直接将命题变元作为一个完整的析取子句与原有的φ进行合取. 若为真命题, 任意命题合取真命题与自身逻辑等价, 保留原先的合取范式. 若为假命题, 则该命题合取φ后不可能被满足, 因此直接构造一个不可满足的合取范式作为生成结果, impossible 为命题变元的名称. 对于 $P\wedge Q$, 可以将其拆分为两个子合取范式, 先对 *P* 生成 *CNF* 后, 再对 *Q* 生成并将两个合取范式分别合取到φ前. 较为复杂的是析取、实质蕴涵和否定的情况, 此时不能根据命题的结构进行简单递归, 使用 *clause* 方法生成析取子句并加入到合取范式中.

Clause 函数接收 4 个参数: 待生成的 *sprop* 命题 *P*、可用命题变元 *s*、已生成的 *clause*、已生成的合取范式φ. 返回 3 个值, 分别为包含 *P* 的转换结果的析取子句τ、修改后的合取范式φ'和新的可用命题变元 *s′*. 以下给出 *clause* 的定义, 不关键时省略参数 *s*:

$$
\begin{aligned}
&clause(SId\ x,\tau,\varphi) ::= x\vee\tau,\varphi\\
&clause(\bot,\tau,\varphi) ::= \tau,\varphi\\
&clause(\top,\tau,\varphi) ::= (tauto\vee\neg tauto,\varphi)\\
&clause(P\wedge Q,s,\tau,\varphi) ::= \text{let } \tau',\varphi',s' := clause(P,next\_name(s),\neg s,\varphi) \text{ in}\\
&\qquad\qquad \text{let } \tau'',\varphi'',s'' := clause(Q,s',\neg s,\tau'\wedge\varphi') \text{ in}\\
&\qquad\qquad\quad s\vee\tau,\tau''\wedge\varphi''
\end{aligned}
$$

$$
\begin{aligned}
clause(P\vee Q,\tau,\varphi) ::= &\ \text{let } \tau',\varphi',s' := clause(P,s,\tau,\varphi) \text{ in}\\
&\ \text{let } \tau'',\varphi'',s'' := clause(Q,s',\tau',\varphi') \text{ in}\\
&\ \tau'',\varphi''\\
clause(P\rightarrow Q,\tau,\varphi) ::= &\ \text{let } \tau',\varphi',s' := neg\_clause(P,s,\tau,\varphi) \text{ in}\\
&\ \text{let } \tau'',\varphi'',s'' := clause(Q,s',\tau',\varphi') \text{ in}\\
&\ \tau'',\varphi''\\
clause(\neg P,\tau,\varphi) ::= &\ neg\_clause(P,\tau,\varphi)
\end{aligned}
$$

对于命题变元, 直接将其析取在现有析取子句中. 假命题析取任何析取子句都与自身等价, 所以不改变析取子句. 真命题析取任何子句都与真命题逻辑等价, 所以构造一个在任何情况下都能满足的析取子句代替 τ. 对于析取命题 $P\vee Q$, 可以分别对两个子命题生成析取子句, 再析取构成一个大的子句, 而实质蕴涵则可以转化为$\neg P\vee Q$ 的等价形式, 除了是对 P 的否命题进行生成(*neg_clause*), 其余操作与析取时相同.

比较复杂的是合取 $P\wedge Q$ 的情况, 此时类似 Tseitin 算法, 我们会引入新命题变元 s 来代表 $P\wedge Q$. 其基本思路是: 在之前已经生成的析取子句τ中只加入命题变元 s, 而将 P 和 Q 生成的 *clause* 直接合取到外层的合取范式中, 最终得到如下 *CNF* 形式:

$$(s\vee\tau)\wedge(\neg s\vee clause(P))\wedge(\neg s\vee clause(Q))\wedge\varphi.$$

容易看出, 上式等价于如下形式:

$$(s\vee\tau)\wedge(s\rightarrow clause(P))\wedge(s\rightarrow clause(Q))\wedge\varphi.$$

对 *clause* 的第一个参数作结构归纳, 可知该形式与$((P\wedge Q)\vee\tau)\wedge\varphi$具有相同的可满足性.

Neg_clause 方法对 P 的否命题生成子句, 具有与 *clause* 对偶的形式. 例如, $\neg(P\wedge Q)$与$\neg P\vee\neg Q$ 等价. 因此, 对合取子句 $P\wedge Q$ 的否命题生成时, 只需分别转化后析取即可, 其余情况类似:

$$
\begin{aligned}
neg\_clause(P\wedge Q,\tau,\varphi) ::= &\ \text{let } \tau',\varphi',s' := neg\_clause(P,s,\tau,\varphi) \text{ in}\\
&\ \text{let } \tau'',\varphi'',s'' := neg\_clause(Q,s',\tau',\varphi') \text{ in}\\
&\ \tau'',\varphi''
\end{aligned}
$$

以命题$P\rightarrow(Q\wedge R)$为例, 由于该命题与$\neg P\vee(Q\wedge R)$等价, 首先对P的否命题生成析取子句, 得到析取子句的第 1 部分$\neg P$; 之后, 根据 *cnf* 的定义计算 $clause(Q\wedge R,\neg P,nil)$. 由上述讨论, 此时会引入新变元 s 代替 $Q\wedge R$, 将 s 与$\neg P$ 合取, 而对 Q 和 R 生成单独的析取子句, 最终得到合取范式:

$$(\neg P\vee s)\wedge(\neg s\vee Q)\wedge(\neg s\vee R).$$

该式的可满足性与原命题相同, 若 $Q\wedge R$ 的真值为 false 时原命题可满足, 则可以取 s 为 false; 否则, 取 s 为 true.

(3) DPLL 求解器

本文在 Coq 中实现了一个 DPLL[21]可满足性求解器, 相比于可满足性求解领域的最新成果, DPLL 算法的效率并不高, 但在本文的验证场景下, 面对的求解问题通常并不复杂, 并且 DPLL 算法可以递归实现, 在 Coq 定理证明器中更容易开发.

DPLL 算法的函数类型为 CNF→*partial_asgn*→*nat*→*bool*. 第 1 个参数为待求解的合取范式; 第 2 个参数为部分赋值 *partial_asgn*:=*list*(*ident***bool*), 是命题变元和布尔值的有序对构成的列表类型, 记录当前部分命题变元的赋值; 第 3 个参数为递归深度, 记录还可以选取的命题变量个数. 本文实现的 DPLL 求解器基于深度进行递归, 分为赋值推导、范式化简、变量选取这 3 个步骤, 定义如下, 其中, φ为合取范式, J 为变元赋值, n 为递归深度.

DPLL(φ,*J*,*n*)

- if n=0, return T
- else let J'=*UnitPro*(J)
  - if J' isconflict return F
  - else let φ'=*filter*(φ,J')

then pick one new variable x

return $DPLL(\varphi', x\mapsto T;;J', n-1) \parallel DPLL(\varphi', x\mapsto F;;J', n-1)$

注意到: 当剩余递归深度为 0 时, 会直接返回 T. 这是由于求解目标是检查待证命题的否命题是否不可满足, 因此求解器应当保证能够正确反馈不可满足的情况. 当命题过于复杂导致求解超出递归深度时, 返回 T 表示求解失败, 无法确认该范式是否不可满足.

Coq 中对应的 *DPLL* 函数定义如下.

```
1.  Fixpoint DPLL_UP(P:CNF) (J:partial_asgn) (n:nat):bool:=
2.     match n with|O⇒true|S n′⇒
3.        match unit_pro P J with
4.        |None⇒false
5.        |Some kJ⇒match kJ with
6.           |nil⇒DPLL_filter P J n′
7.           |_⇒DPLL_UP P(kJ++ J) n′
8.           end
9.        end
10.    end
11. with DPLL_filter(P:CNF) (J:partial_asgn) (n:nat): bool:=
12.    match n with|O⇒true|S n′⇒
13.       DPLL_pick(CNF_filter P J) J n′
14.    end
15. with DPLL_pick(P:CNF) (J:partial_asgn) (n:nat): bool:=
16.    match n with|O⇒true|S n′⇒
17.       let x:=pick P in
18.       DPLL_UP P((x,true)::J) n′||DPLL_UP P((x,false)::J) n′
19.    end.
```

DPLL_filter 函数调用 *CNF_filter* 对待检查的合取范式进行化简, *CNF_filter* 会根据赋值 J 删除所有已经被满足的析取子句. 在剩余的子句中, 已知该子句在当前赋值下不可满足, 除了未被赋值变元对应的 *literal*, 已赋值变量的 *literal* 一定为 false, 而任何命题与 false 析取都与自身等价, 因此, 可以进一步删去已赋值变量的 *literal*, 得到一个完全由未赋值变量构成的合取范式进行下一步变量选取. *DPLL_pick* 函数执行变量选取并进行下一层递归, 已知此时合取范式 P 中均为未赋值变量, *pick* 函数简单选取合取范式中第 1 个命题变元进行递归即可.

接下来阐述赋值推导的实现. *Unit_pro* 函数接受合取范式和部分赋值作为输入, 若赋值推导发现冲突, 返回 *None*; 否则, 返回新推导出的赋值列表. *DPLL_UP* 中会反复调用 *unit_pro*, 直到发现冲突或者推导不出新赋值为止, 再进行后续的化简操作. 赋值推导的基础是对每一个子句进行推导, 定义函数 *find_unit_pro_in_clause* 执行此操作, 返回值类型为 *UP_result*:

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

只有当子句中除了最后一个未赋值的变元外, 其他变元的 *literal* 都为 false 时, 才能够推导出该变元的赋值, 使用 *UP* 构造子记录该变元及其推导出的赋值, *Conflict* 和 *Nothing* 构造子则表示冲突和没有新的推导. *Find_unit_pro_in_clause* 会带着一个类型为 *UP_result*、初始值为 *Conflict* 的参数 *cont* 从前向后依次判断, 对子句中的每个 *literal*, 查找部分赋值 J, 若该变元已经被赋值且 *literal* 为 true, 则当前子句已经被满足, 推导不出新赋值, 返回 *Nothing*; 若已经被赋值但 *literal* 为 false, 需要继续进行后续推导. 若 J 中找不到当前变元, 说明遇见了没有被赋值的变元, 此时需要检查参数 *cont*: 如果 *cont* 的值为 *Conflict*, 表示是首次遇到未赋值的变元,

那么修改 *cont* 为使当前 *literal* 被满足的赋值, 继续向后检查; 如果 *cont* 已经为对某一变元的赋值, 那么此时遇到了子句中第 2 个没有被赋值的变元, 无法进行赋值推导, 直接返回 *Nothing*. 当检查进行到子句末尾时, 如果参数 *cont* 仍为 *Conflict* 没有被修改, 说明子句中所有的变元都被赋值, 且所有 *literal* 都为 false, 当前子句不可满足; 否则, 当前子句有且仅有一个未赋值变元, 且推导出的新赋值被记录在 *cont* 中. 该函数的具体实现如下.

```
1.  Fixpoint find_unit_pro_in_clause (c:clause) (J:partial_asgn) (cont:UP_result): UP_result:=
2.    match c with
3.    |nil⇒cont
4.    |(op,x)::c'⇒
5.      match PV.look_up x J with
6.      |None⇒match cont with
7.        |Conflict⇒find_unit_pro_in_clause c' J (UP x op)
8.        |UP__⇒Nothing
9.        |_⇒Nothing
10.       end
11.     |Some b⇒if eqb op b then Nothing else find_unit_pro_in_clause c' J cont
12.   end
13. end.
```

Unit_pro 函数会对每个子句执行上述操作, 若出现 *Conflict*, 返回 *None*; 否则, 将所有推导出的单个赋值组合成新赋值列表. 注意到可能出现一个子句推导出 $x\mapsto T$ 而另一个子句推导出 $x\mapsto F$ 的情况, *unit_pro* 会直接将两个赋值都加入推导出的新赋值列表中, 冲突检查被推迟到了 *DPLL_UP* 的下一轮 *unit_pro*. 虽然新赋值 *J* 中对同一变元有不同赋值, 但使用 *look_up* 在赋值中查找 *x* 时, 只会找到 *J* 中靠前的赋值, 这将必然导致上一轮推导出两个不同 *x* 赋值子句的其中一个不能被满足, 从而产生 *Conflict*.

(4) FOL_Tauto 证明策略实现

基于命题有效性和其否命题的不可满足性的等价性, 定义一阶逻辑命题的有效性 *valid* 为:

```
1.  Definition valid (P:prop): bool:=
2.    match sprop_gen (PNot P) nil "x" with
3.    |(P',_,n)⇒
4.      match cnf_gen P' n nil with
5.      |(P'',_)⇒negb(DPLL_UP P'' nil 24)
6.      end
7.    end.
```

对一阶逻辑命题 *P*, 首先调用 *sprop_gen* 生成$\neg P$ 对应的命题逻辑命题 *P*′, 再调用 *cnf_gen* 生成 *P*′的合取范式 *P*″. 若 DPLL 求解器能在 24 层递归内判定该范式不可满足, 那么称命题 *P* 是有效的.

在证明环境中, 用户有多个具有[[$\Gamma\vdash\varphi$]]形式的前提条件(即 derivable Γ φ), 定义类型 *der_judgement*:=*list prop***prop* 来记录其中涉及的命题, *list prop* 记录前提中的多个命题, 而 *prop* 为该推导中的结论命题. 用户希望根据现有的多个条件自动证明一个新的具有[[$\Gamma\vdash\varphi$]]的形式的命题, 因此, FOL_Tauto 需要处理的是多个作为条件的推导和一个作为结论的推导, 记作 *proof_goal* 类型:

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

Valid 处理的是单个命题, 因此需要将 *proof_goal* 转化为单一的一阶逻辑命题. 定义 *pg2prop* 实现这一目标, 对于每一个 *der_judgement*, 使用 *PImpl* 逻辑连接词连接前提中的命题和结论命题, 形成单一的一阶逻辑命题; 再将 *proof_goal* 中每个 *der_judgement* 转化后的一阶逻辑命题再次使用 *PImpl* 连接, 得到最终可以被自

动求解的单一一阶逻辑命题.

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

DPLL 求解器的可靠性由以下性质保证, 描述了从 DPLL 求解结果到 Coq 证明环境的转换:

Axiom *dpll_sound*: forall *pg*: *proof_goal*, *valid*(*pg*2*prop pg*)=true→*denote_pg pg*,

其中, *denote_pg* 将 *proof_goal* 拆开并转化回 Coq 证明环境中的 derivable 条件. 该定理声明: 对任意 *proof_goal*, 如果将其转化成一阶逻辑命题后由 DPLL 求解器判定有效, 那么就可以从当前 Coq 证明环境中的前提条件证明得到用户要求的结论. 这一性质使用 Axiom 直接声明.

FOL_Tauto 自动证明策略的最终实现为:

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

Reify_pg 指令会根据当前证明环境计算出 *proof_goal*; 之后, 应用 *dpll_sound* 将待证目标转化为一阶逻辑命题的有效性判断; 最后, 通过 reflexivity 指令由 Coq 自动完成 DPLL 求解.

以证明空集是任何集合的子集为例.

首先 pose proof 空集公理, 并将空集公理实例化成 y 后, 此时证明状态如图 3 所示.

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

(a) 证明代码

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

(b) 当前证明状态

图 3 证明空集相关性质的示例

由条件 $H0$ 已知 $y\notin\varnothing$, 由此可以知道 $y\in\varnothing$ 能够推出任何命题, 根据子集的定义, 此处需要证明空集中的元素都是集合 x 中的元素, 那么通过 assert $[[ZF\vdash y\in\varnothing\to y\in x]]$ by FOL_Tauto 就可以直接在前提中加入新条件 $H1$, 证明状态变为如图 4 所示.

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

图 4 新证明状态

这一过程内部事实上可以分解为以下步骤.

- 证明策略 assert $[[ZF\vdash y\in\varnothing\to y\in x]]$ 首先声明了一个新的证明目标分支, 当前分支的证明状态如下.

FOL_Tauto 先会执行 *reify_pg* 指令, 将图 5(a)中的条件和结论通过 revert, change 等指令变成图 5(b)中 *denote_pg* 的形式, 此时已经计算出图 5(a)对应的 *proof_goal*, 该证明目标由两部分组成: 第 1 部分是由前提条件 H 与 $H0$ 计算得到的 *der_judgement* 列表, 由于 H 和 $H0$ 条件中均为空环境 ZF, 因此每个 *der_judgement* 的第 1 部分均为空命题列表; 第 2 部分则为用户声明的结论对应的 *der_judgement*.

- 之后, 应用 *dpll_sound*, 将证明条件变为 *proof_goal* 转化后的有效性检查.

Coq 能够自动计算 *pg2prop*, 将 *proof_goal* 转化为图 6(b)中的一阶逻辑命题, 最后 *valid* 会自动调用命题转化, 合取范式生成, DPLL 求解等步骤计算出该命题确实有效, 回到图 4 的证明状态. 而在用户实际的证明环境中, 上述所有步骤都隐藏在指令 assert [[$ZF\vdash y\in\varnothing\to y\in x$]] by FOL_Tauto 一步的操作中.

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

(a) assert 后当前分支的证明状态

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

(b) *reify_pg* 后的证明状态

图 5 FOL_Tauto 证明步骤拆解

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

(a) 应用 *dpll_sound* 后的证明状态

```
2 subgoals
_______________________ (1/2)
denote_pg ([
([·],[[∀y, ¬y∈∅]]);
([·],[[¬y∈∅]])
],
([·],[[y∈∅→y∈x]]))
```

(b) 计算 *pg2prop* 后的证明状态

图 6 *proof_goal* 转换

4.2 量词自动证明策略

在第 3.3 节中, 全称量词和存在量词各有两条推理规则, 我们为这 4 条规则分别定义了 4 条自动证明策略, 此处我们介绍其中两条的实现.

(1) 全称量词引入指令 universal generalization

在图 4 空集是任何集合子集的证明中, 经过 FOL_Tauto, 我们获得了条件[[$ZF\vdash y\in\varnothing\to y\in x$]], 然而待证目标为[[$ZF\vdash\forall x, \varnothing\in x$]], 即[[$ZF\vdash\forall x, \forall y, y\in\varnothing\to y\in x$]](子集符号也是通过 notation 进行的简写), 条件 $H1$ 与待证目标相差两层全称量词, 回忆量词的引入规则 *PForall_intros*:

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

要在结论命题前增加全称量词, 需要保证变量在前提的所有命题中都没有自由出现. 我们开发了证明策略 *universal_generalization_constr*, 该指令接受两个参数: 其一为 Coq 环境中需要添加量词的前提条件的编号, 如 $H1$, 准确地说是该条件(即 derivable $\Gamma\ \varphi$)的证明对象; 另一个参数为变量名列表 *xs*, 用来实现多个全称量词的一次性引入. 该指令实现如下: 当待引入列表为空时, 引入结束, 返回 H; 否则, 先检查条件 H 是否是 derivable 推导, 之后应用 *PForall_intros* 规则构造新的证明条件, 其中, *check_free_occurrence* 指令能够自动判断变量 x 是否在前提 *Phi* 中自由出现. *Universal generalization* 指令调用 *universal_generalization_constr*, 若成功, 则在证明环境中 pose proof 新前提; 否则显示错误信息.

```
1. Ltac universal_generalization_constr H xs:=
2.    match xs with
3.    |nil⇒constr:(H)
4.    |?x::?xs0⇒
5.       match type of H with
6.       |[[?Phi⊢?P]]⇒
7.          let H0:=constr:(PForall_intros Phi x P ltac:(check_free_occurrence) H) in
```

```
8.          universal_generalization_constr H0 xs0
9.        end
10.     end.
11. Ltac universal_generalization H xs:=
12.     first
13.     [let H0:=universal_generalization_constr H xs in pose proof H0
14.       |fail 1 "Universal generalization fails"].
```

在图 4 的证明中, 运行指令 *universal generalization H*1 *x y*, 就可以得到新条件 *H*2, 此时, *H*2 就是待证目标, 运行 The conclusion is already proved 指令结束证明(如图 7 所示).

```
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]]
```

图 7 *Universal generalization* 运行效果

(2) 存在量词引入指令 *existential generalization*

存在量词自动化引入指令 *existential generalization* 具有与 *universal generalization* 不同的风格. 回忆存在量词的引入规则 *PExists_intros*:

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

如果前提能够推出将 P 中的 x 进行实例化后的结论, 那么前提能够推导出$\exists x,\ P$. 但与全称量词的情况不同, 用户不能仅仅指出他要抽象的一阶逻辑项, 假如证明环境中有前提条件$[[ZF\vdash\varnothing=\varnothing]]$, 用户可能需要的结论是$[[ZF\vdash\exists x,\ x=\varnothing]]$. 如果我们的命令只能让用户指定抽象的项$\varnothing$, 则会得到结论$[[ZF\vdash\exists x,\ x=x]]$, 与用户的期望不符. 因此, *existential generalization* 指令除了接受前提条件的编号, 还会接受一个由用户指定的命题, 用户只需直接写出他希望得到的结论, 该指令会自动搜索用户抽象的一阶逻辑项来应用存在量词的引入规则.

假设证明环境中有条件 H: $[[ZF;;x\cup y=z\vdash z=x\cup y]]$, 用户执行指令 *existential generalization H* $[[\exists u,\ y=u]]$, 那么证明环境中就可以直接添加新条件 $H0$: $[[ZF;;x\cup y=z\vdash\exists u,\ y=u]]$, 该指令会自动搜索得到 u 对应原来的项 $x\cup y$ 并应用存在量词引入规则. 该指令同样支持用户一次性引入多个存在量词.

Existential generalization 策略的实现如下, 参数 H 为前提条件编号, P 为用户声明的命题.

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

首先, 根据用户指定的命题 P 和前提条件中的结论命题 Q 调用 *generate_exists_term_tac* 搜索用户抽象掉的一阶逻辑项, 该指令会先比较 P Q 确定用户在 Q 中新引入的存在量词变量列表, 再对每一个变量 x 进行分

别搜索, 若命题 Q 中为变量 x, 而命题 P 的同一位置为一阶逻辑项 t, 则认为用户将 t 抽象成了一阶逻辑项, 遇到逻辑连接词时分别搜索子项, 返回时确认两边不会有冲突结果, 最终生成对应的项列表 ts. 之后, 应用导出规则 *existential_generalization_tac_aux*, 该规则是多次替换量词引入推理规则的版本, 将用户抽象后的命题使用 ts 再次实例化后, 应用$\alpha_{CONGRUENCE}$规则, 调用第 3.2 节中的 *aeq* 函数判断实例化的命题与条件中的原命题 Q 是否等价, 从而完成新条件的引入和证明.

(3) 量词消去指令

Universal instantiation 指令具有与 *universal generalization* 指令相同的风格, 若证明环境中有条件形如 H:[[$\Gamma\vdash\forall x, \forall y,\ldots$]], 那么执行指令 *universal instantiation* $H\ t_1\ t_2$, 就可以依次将命题中的 $x,y,\ldots$实例化为 $t_1,t_2,\ldots$. 具体例子可见图 2(b)中从条件 H 到条件 $H0$ 的效果.

Existential instantiation 指令则与 *existential generalization* 具有相同的风格和思路, 用户直接写出前提中想增加存在量词的新命题, 该指令会自动在指定条件的前提中搜索相应命题并应用$\exists_{Elim}$规则和$\alpha_{CONGRUENCE}$规则生成新的条件.

4.3 等号替换自动证明策略

关于等号有如下推理规则: 若一阶逻辑项 t 与 t'相同, 那么 $P[x\mapsto t]$蕴含 $P[x\mapsto t']$. 但直接使用该推理规则是麻烦的, 用户通常有两个项相同以及包含其中一个项的命题, 需要将该命题中的项替换成新的项, 而非一个实例化之前的命题. 为此, 开发了等号替换指令 *peq_sub_tac* 以实现命题 P 的生成和替换:

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

Peq_sub_tac 接受 3 个参数, 分别为相等的项 t 和 t'以及用户声明的含 t 的命题 P. 该指令将引入一个 P 中没有出现的新变量名 v, 通过 *replace_prop_tac* 将 P 中所有项 t 替换为 v, 得到命题 Q; 之后, 使用 *PEq_sub* 规则构造新命题 $Q[v\mapsto t']$, 在证明环境中加入新条件[[$ZF\vdash t=t'\rightarrow P\rightarrow Q'$]], Q'为展开 $Q[v\mapsto t']$后并且保持谓词简写不被展开的形式. *Peq_sub_tac* 策略的实现如下.

```
1.  Ltac peq_sub_tac t t' P:=
2.      let H0:= fresh "H" in
3.      let H1:= fresh "H" in
4.      let v:=get_new_var P ShortNames.x [[∅]] ShortNames.x in
5.      let Q:=replace_prop_tac v t P in
6.      pose proof (PEq_sub Q v t t') as H1;
7.      let Q':=subst_aeq_constr_tac Q v t' in
8.      assert [[ZF;;t=t';;P⊢Q']] as H0 by FOL_tauto; clear H1.
```

5 教学应用

在大一新生的离散数学课程教学中引入了我们开发的证明器, 将使用该证明器证明数学归纳法作为课程项目. 实践表明: 选择该项目的学生均能根据教科书上数学归纳法的证明思路, 成功给出该定理的形式化证明. 有较优秀的同学在课程结束后, 进一步利用该工具形式化了皮亚诺算术系统.

5.1 课程项目: 数学归纳法的形式化证明

虽然ZFC集合论作为一种数学基础被广泛接受, 但是我们更希望让学生认识到ZFC集合论和日常数学间的联系. 数学归纳法作为学生们熟知的证明方法, 是一个很好的研究对象. 学生们通常只是默认该方法的正确性, 但在 ZFC 公理集合论体系下, 数学归纳法实际上是可以被证明的定理. 我们将使用该证明器证明数学归纳法作为两个可选的附加课程项目之一, 学生可在该项目和另一个可选任务间任选其一.

(1) 项目设计

我们为该项目提供了一个教程文档, 为 Coq 证明器使用的 *v* 文件, 其中包括 300 行注释形式的说明、展示命题和逻辑系统的语法、一些谓词符号如子集关系的定义、Coq 的最基本操作(按键使用、特殊符号输入)以及 5 条自动化证明指令的功能(FOL_Tauto 和量词证明策略, 该项目不需要用到等号替换指令 *peq_sub_tac*). 另外, 在文档中提供了 4 个简单定理的证明, 共计约 30 行证明代码, 让学生熟悉如何在该证明环境中进行证明. 我们为该教程提供了一个配套的 15 分钟演示视频, 学生结合视频和教程文档, 就基本具备了在该环境中进行证明的能力. 除此之外, 不另设课时进行 Coq 相关教学.

关于学生需要完成的数学归纳法证明, 我们事先给出了谓词 *is_inductive* 和 *is_natural_number* 的定义, 分别表示某集合是归纳集和某集合是自然数集.

- Definition *is_inductive_def*(*t*:*term*):=[[$(\varnothing\in x\wedge\forall y, (y\in x\rightarrow y\cup\{y\}\in x))[x\mapsto t]$]];
- Definition *is_natural_number_def*(*t*:*term*):=[[(*is_inductive* $x\wedge\forall w$, (*is_inductive* w)$\rightarrow x\subseteq w)[x\mapsto t]$]].

之后, 我们提供了待验证目标, 使学生能够逐步构造数学归纳法等定理的形式化证明. 学生需要使用我们开发的证明策略和 pose proof 指令依次证明以下目标.

- 互为子集的集合相同:

  Theorem *subset_subset_equal*: [[$ZF\vdash\forall x, \forall y, x\subseteq y\rightarrow y\subseteq x\rightarrow x=y$]];
- 自然数集唯一:

  Theorem *Nat_unique*: [[ZF;;*is_natural_number* x;;*is_natural_number* $y\vdash x=y$]];
- 数学归纳法的基础步骤: 若 N 为自然数集, 且自然数 0 在集合 X 中, 那么 0 也在 $X\cap N$ 中:

  Theorem *Nat_inductive_base_step*: [[ZF;;*is_natural_number* N;;$\varnothing\in X$;;$\forall n, n\in N\rightarrow n\in X\rightarrow n\cup\{n\}\in X\vdash\varnothing\in X\cap N$]];
- 数学归纳法的归纳步骤: 若 0 在集合 X 中, 且对于任意自然数 n, $S(n)$都在集合 X 中, 那么 $X\cap N$ 为归纳集:

  Theorem *Nat_intersection_inductive*:

  [[ZF;;*is_natural_number* N;;$\varnothing\in X$;;$\forall n, n\in N\rightarrow n\in X\rightarrow n\cup\{n\}\in X\vdash$ *is_inductive*$(X\cap N)$]];
- 数学归纳法: 若 0 在集合 X 中, 且对于任意自然数 n, $S(n)$都在集合 X 中, 那么任意自然数 n 都在集合 X 中:

  Theorem *mathmetical_induction*:

  [[ZF;;*is_natural_number* N;;$\varnothing\in X$;;$\forall n, n\in N\rightarrow n\in X\rightarrow n\cup\{n\}\in X\vdash\forall n, n\in N\rightarrow n\in X$]].

学生也可以根据证明需要, 自己添加并证明中间步骤的定理.

(2) 教学效果

在 15 名选择参加附加课程项目的同学中, 有 8 人选择了形式化证明数学归纳法的项目, 学生确实展现出了对使用定理证明器完成形式化证明的兴趣. 8 名学生均成功完成了证明. 经统计, 8 名同学平均使用了 88 行代码完成了上述目标的证明. 其中, 两名同学分别使用 65 行和 66 行就完成了证明. 由老师和助教草拟的证明将证明过程拆分成了多个子目标, 总共花费了 67 行. 剩余 6 名同学的证明行数都在 100 行上下, 耗费行数较多的原因在于: 他们在证明过程中将可以由 FOL_Tauto 自动证明策略一步解决的证明拆分成了数步更基础的、贴近每一个逻辑连接词推理规则的证明, 并分别使用 FOL_Tauto 解决. 因此, 有同学在证明过程中由于证明环境条件太多而导致 DPLL 超过预设的递归深度求解失败. 与助教讨论后, 将证明拆解成了更多子目标, 最终也成功完成了数学归纳法的形式化证明.

综上, 从未接触过交互式定理证明的学生能够通过本文开发的证明环境快速熟悉使用交互式定理证明工具对数学定理进行形式化证明的方法, 所有同学均能够在 Coq 交互式定理证明器的帮助下成功给出数学归纳法的严格形式化证明, 并且部分同学能够达到较高的应用水平, 这在传统教学模式下是难以做到的. 在教学中引入交互式定理证明器, 确实有助于学生更透彻地理解逻辑系统和 ZFC 公理集合论. 总体实现了教学目标, 展现了该工具的可用性和易用性.

(3) 证明实例

本节给出数学归纳法基础步骤的 Coq 证明, 并与教科书中逻辑的证明相对照, 以便直观体现该定理证明工具的效果. Coq 证明中的一步可能对照教科书证明中的多步, 为方便比较, 对应的步骤采用相同的行号.

基础步骤 *Nat_inductive_base_step*: 若 N 为自然数集, 且自然数 0 在集合 X 中, 那么 0 也在 $X\cap N$ 中.

1. Lemma *Nat_inductive_base_step*:
2. [[ZF;;*is_natural_number* N;;$\varnothing\in X$;;$\forall n, n\in N\to n\in X\to n\cup\{n\}\in X\vdash\varnothing\in X\cap N$]].
3. *Proof*.
4. pose proof *Intersection_iff*.
5. *universal instantiation H X N* [[$\varnothing$]].
6. The conclusion is already proved.
7. Qed.

这一基础步骤对应的教科书形式化证明如下:

4. $ZF\vdash\forall x, \forall y, \forall z, z\in x\cap y\leftrightarrow z\in x\wedge z\in y$ (二元交规则)
5. $ZF\vdash\forall y, \forall z, z\in X\cap y\leftrightarrow z\in X\wedge z\in y$ (应用$\forall_{Elim}$, 将 x 实例化为 X)
5. $ZF\vdash\forall z, z\in X\cap N\leftrightarrow z\in X\wedge z\in N$ (应用$\forall_{Elim}$, 将 y 实例化为 N)
5. $ZF\vdash\varnothing\in X\cap N\leftrightarrow\varnothing\in X\wedge\varnothing\in N$ (应用$\forall_{Elim}$, 将 z 实例化为$\varnothing$)
6. $ZF\vdash\varnothing\in X\wedge\varnothing\in N\to\varnothing\in X\cap N$ ($\leftrightarrow$消去规则)
6. ZF;;*is_natural_number* N;;$\varnothing\in X$;;$\forall n, n\in N\to n\in X\to n\cup\{n\}\in X\vdash\varnothing\in X\wedge\varnothing\in N\to\varnothing\in X\cap N$ (Weaken 规则)
6. ZF;;*is_natural_number* $N\vdash\varnothing\in N$ (*is_natural_number* 定义和合取消去规则)
6. ZF;;*is_natural_number* N;;$\varnothing\in X$;;$\forall n, n\in N\to n\in X\to n\cup\{n\}\in X\vdash\varnothing\in N$ (Weaken 规则)
6. ZF;;$\varnothing\in X\vdash\varnothing\in X$ (Assu 规则)
6. ZF;;*is_natural_number* N;;$\varnothing\in X$;;$\forall n, n\in N\to n\in X\to n\cup\{n\}\in X\vdash\varnothing\in X$ (Weaken 规则)
6. ZF;;*is_natural_number* N;;$\varnothing\in X$;;$\forall n, n\in N\to n\in X\to n\cup\{n\}\in X\vdash\varnothing\in X\wedge\varnothing\in N$ (合取引入规则)
6. ZF;;*is_natural_number* N;;$\varnothing\in X$;;$\forall n, n\in N\to n\in X\to n\cup\{n\}\in X\vdash\varnothing\in X\cap N$ (Modus Ponens 规则)

引理得证.

若按照教科书严格地逐条应用推理规则, 该引理的证明需要 10 多步. 而在 Coq 中, 使用我们开发的证明策略只需要 3 行. 其中, 行号 4 的 *universal instantiation* 指令一步使用了 3 次$\forall_{Elim}$推理规则, 进行了 3 个量词的实例化; 而 The conclusion is already proved (FOL_Tauto)指令则组合使用了多个逻辑连接词的推理规则, 在不涉及量词规则的情况下, 直接证明了最终结论. 在同样保持严格形式化的前提下, 学生使用我们开发的工具在 Coq 中进行证明相比纸笔证明更简便, 且 Coq 能够实时检查学生构造证明的正确性.

此外能够看出, 教科书证明中的每一步都是一个推导关系, 这也正是使用本工具证明时, Coq 证明状态栏显示的内容(如图 1(b)所示), 它记录了学生当前已经证明出的结论. 而学生输入的 Coq 证明代码决定了如何选取合适的条件并应用推理规则, 其地位相当于上述教科书证明中括号里对推理规则的说明, 这与教科书证明是思路一致且同样直观的.

下列出归纳步骤 *Nat_intersection_inductive* 的证明, 包括 Coq 代码和证明状态, 其中, 证明状态也可以视作教科书证明中的步骤. 为证明 *Nat_intersection_inductive*, 我们先证明两个引理.

$X\cap N$ 中元素的后继都在 X 中.

1. Lemma *Nat_inductive_inductive_step_X*:
2. [[ZF;;*is_natural_number* N;;$\varnothing\in X$;;$\forall n, n\in N\to n\in X\to n\cup\{n\}\in X$;;$y\in X\cap N\vdash y\cup\{y\}\in X$]].
3. *Proof*.
4. pose proof *Intersection_iff*.
5. *universal instantiation H X N y*.

6. assert [[ZF;;$\forall n$, $n\in N\to n\in X\to n\cup\{n\}\in X\vdash\forall n$, $n\in N\to n\in X\to n\cup\{n\}\in X$]] by FOL_tauto.
7. *universal instantiation* $H1$ y.
8. The conclusion is already proved.
9. Qed.

对应证明状态如下, 行号与前文 Coq 代码相对应, 此外, 还加入了 Coq 中显示的条件编号如 H, $H0$, 方便与 Coq 指令对照.

4. H: [[$ZF\vdash\forall x$, $\forall y$, $\forall z$, $z\in x\cap y\leftrightarrow z\in x\wedge z\in y$]] (二元交规则)
5. $H0$: [[$ZF\vdash y\in X\cap N\leftrightarrow y\in X\wedge y\in N$]] (*universal instantiation* 将 H 中的 x, y, z 分别实例化)
6. $H1$: [[ZF;;$\forall n$, $n\in N\to n\in X\to n\cup\{n\}\in X\vdash ZF$;;$\forall n$, $n\in N\to n\in X\to n\cup\{n\}\in X$]] (Assu 规则)
7. $H2$: [[ZF;;$\forall n$, $n\in N\to n\in X\to n\cup\{n\}\in X\vdash y\in N\to y\in X\to y\cup\{y\}\in X$]] (将 $H1$ 的 n 实例化为 y)
8. [[ZF;;*is_natural_number* N;;$\varnothing\in X$;;$\forall n$, $n\in N\to n\in X\to n\cup\{n\}\in X$;;$y\in X\cap N\vdash y\cup\{y\}\in X$]] (根据 $H0$ $H2$ 由 FOL_Tauto 证明)

类似地, 可以证明 $X\cap N$ 中的元素的后继都在 N 中, 并进一步得到归纳步骤 *Nat_intersection_inductive*. 现给出最终数学归纳法的证明.

1. Theorem *mathmetical_induction*:
2. [[ZF;;*is_natural_number* N;;$\varnothing\in X$;;$\forall n$, $n\in N\to n\in X\to n\cup\{n\}\in X\vdash\forall n$, $n\in N\to n\in X$]].
3. *Proof*.
4. pose proof *Nat_intersection_inductive*.
5. assert ([[ZF;;*is_natural_number* $N\vdash\forall w$, *is_inductive* $w\to N\subseteq w$]]) by FOL_tauto.
6. *universal instantiation* $H0$ [[$X\cap N$]].
7. assert ([[ZF;;*is_natural_number* N;;*is_inductive* $X\cap N\vdash N\subseteq X\cap N$]]) 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;;$\varnothing\in X$;;$\forall n$, $n\in N\to n\in X\to n\cup\{n\}\in X\vdash n\in N\to n\in X$]]) by FOL_tauto.
12. *universal generalization* $H6$ n.
13. The conclusion is already proved.
14. Qed.

对应证明状态如下.

4. H: [[ZF;;*is_natural_number* N;;$\varnothing\in X$;;$\forall n$, $n\in N\to n\in X\to n\cup\{n\}\in X\vdash$ *is_inductive* $X\cap N$]] (pose proof 已证的归纳步骤 *Nat_intersection_inductive*)
5. $H0$: [[ZF;;*is_natural_number* $N\vdash\forall w$, *is_inductive* $w\to N\subseteq w$]] (*is_inductive* 定义和合取消去)
6. $H1$: [[ZF;;*is_natural_number* $N\vdash$ *is_inductive* $X\cap N\to N\subseteq X\cap N$]] (*universal instantiation* 将 $H0$ 中的 w 实例化为 $X\cap N$)
7. $H2$: [[ZF;;*is_natural_number* N;;*is_inductive* $X\cap N\vdash N\subseteq X\cap N$]] (实质蕴涵→的性质)
8. $H3$: [[ZF;;*is_natural_number* N;;*is_inductive* $X\cap N\vdash n\in N\to n\in X\cap N$]] (子集定义)
9. $H4$: [[$ZF\vdash\forall x$, $\forall y$, $\forall z$, $z\in x\cap y\leftrightarrow z\in x\wedge z\in y$]] (pose proof 二元交规则)
10. $H5$: [[$ZF\vdash n\in X\cap N\leftrightarrow n\in X\wedge n\in N$]] (*universal instantiation* 将 $H4$ 中的 x, y, z 分别实例化)
11. $H6$: [[ZF;;*is_natural_number* N;;$\varnothing\in X$;;$\forall n$, $n\in N\to n\in X\to n\cup\{n\}\in X\vdash n\in N\to n\in X$]] (由 H $H3$ $H5$ 使用 FOL_Tauto 得到)
12. $H7$: [[ZF;;*is_natural_number* N;;$\varnothing\in X$;;$\forall n$, $n\in N\to n\in X\to n\cup\{n\}\in X\vdash\forall n$, $n\in N\to n\in X$]] (*universal generalization* 在 $H6$ 的结论前引入量词 n, 数学归纳法得证)

5.2 皮亚诺算术系统的形式化

有较优秀的同学在完成离散数学的课程学习后, 进一步使用该工具形式化了皮亚诺算术系统中的加法和乘法, 体现了该工具的有效性, 以下列出部分重要定义和定理.

先给出加法的定义: 对任意自然数 n, 三元组$(0,n,n)$在集合 X 中; 对 X 中的任意三元组(n,d,e), 三元组$(S(n),d,S(e))$也在集合 X 中. 其中, 谓词 *in_rel*3 *x y z d* 表示三元组(x,y,z)在 d 中.

Definition *is_legal_plus_def*(t_1 t_2:*term*):*prop*:=

$[[((\forall n, n\in N\rightarrow in\_rel3\ \varnothing\ n\ n\ x)\wedge(\forall n, \forall d, \forall e, n\in N\wedge d\in N\wedge e\in N\rightarrow(in\_rel3\ n\ d\ e\ x\rightarrow in\_rel3\ n\cup\{n\}\ d\ e\cup\{e\}\ x)))[x\mapsto t_1; N\mapsto t_2]]]$.

- 集合 X 定义了自然数 N 上的加法关系, 用谓词 *is_plus X N* 表示, 其定义为最小的满足 *is_legal_plus* 的集合.

Definition *is_plus_def*(t_1 t_2:*term*):=

$[[(is\_legal\_plus\ x\ N\wedge(\forall y, is\_legal\_plus\ y\ N\rightarrow x\subseteq y))[x\mapsto t_1; N\mapsto t_2]]]$.

之后, 可以给出乘法的定义:

对任意自然数 n, 三元组$(n,0,0)$在集合 f 中, 对 f 中的任意三元组(x,y,z), 如果(z,x,a)在加法关系 e 中, 那么有三元组$(x,S(y),a)$在集合 f 中.

Definition *is_legal_mult_def*(t_1 t_2 t_3:*term*):*prop*:=

$[[((\forall n, n\in N\rightarrow in\_rel3\ n\ \varnothing\ \varnothing\ f)\wedge(\forall x, \forall y, \forall z, \forall a, x\in N\wedge y\in N\wedge z\in N\wedge a\in N\rightarrow in\_rel3\ x\ y\ z\ f\rightarrow(in\_rel3\ z\ x\ a\ e\rightarrow in\_rel3\ x\ y\cup\{y\}\ a\ f)))[f\mapsto t_1; e\mapsto t_2; N\mapsto t_3]]]$.

- 集合 X 基于加发关系 e 定义了自然数 N 上的乘法关系, 用谓词 *is_mult X e N* 表示, 其定义为最小的满足 is_*legal_mult* 的集合.

Definition *is_mult_def*(t_1 t_2 t_3:*term*):=

$[[(is\_legal\_mult\ x\ e\ N\wedge(\forall y, is\_legal\_mult\ y\ e\ N\rightarrow x\subseteq y))[x\mapsto t_1; e\mapsto t_2; N\mapsto t_3]]]$.

基于以上定义, 可以给出几条皮亚诺公设的证明, 数学归纳法已证.

- 自然数 0 不是任何数的后继:

Lemma *not_empty*: $[[ZF\vdash\forall n, \neg n\cup\{n\}=\varnothing]]$;

- 若两个自然数的后继相等, 那么这两个自然数相等, 反方向由等号替换规则显然:

Lemma *Sn_inversion*: $[[ZF\vdash\forall x, \forall y, x\cup\{x\}=y\cup\{y\}\rightarrow x=y]]$.

之后, 证明加法的相关性质.

- 加法集合唯一:

Lemma *plus_unique*: $[[ZF;;is\_natural\_number\ N\vdash\forall x, \forall y, is\_plus\ x\ N\wedge is\_plus\ y\ N\rightarrow x=y]]$;

- 对任意两个自然数, 都存在自然数为它们的和:

Lemma *in_plus_exists*: $[[ZF;;is\_natural\_number\ N;;is\_plus\ e\ N\vdash\forall x, \forall y, x\in N\wedge y\in N\rightarrow\exists z, z\in N\wedge in\_rel3\ x\ y\ z\ e]]$.

加法的函数性、交换律、结合律, 都需要分别证明基础步骤和归纳步骤, 再使用上一节证明的数学归纳法进行证明, 此处仅以加法函数性为例.

- 加法函数性基础步骤: 如果 $0+x=y$, 那么 $x=y$.

Lemma *plus_func_zero*:

$[[ZF;;is\_natural\_number\ N;;is\_plus\ e\ N\vdash\forall x, \forall y, x\in N\wedge y\in N\wedge in\_rel3\ \varnothing\ x\ y\ e\rightarrow x=y]]$.

- 加法函数性归纳步骤: 如果 $x+y=z$ 并且 $x+y=a$ 能够推导 $z=a$, 那么 $S(x)+y=z'$ 和 $S(x)+y=a'$ 能够得到 $z'=a'$.

Lemma *plus_func_induction*:

$[[ZF;;is\_natural\_number\ N;;is\_plus\ e\ N\vdash$

$\forall x, x\in N\wedge(\forall y, \forall z, \forall a, y\in N\wedge z\in N\wedge a\in N\wedge in\_rel3\ x\ y\ z\ e\wedge in\_rel3\ x\ y\ a\ e\rightarrow z=a)\rightarrow$

$x\cup\{x\}\in N\wedge(\forall y, \forall z, \forall a, y\in N\wedge z\in N\wedge a\in N\wedge in\_rel3\ x\cup\{x\}\ y\ z\ e\wedge in\_rel3\ x\cup\{x\}\ y\ a\ e\rightarrow z=a)]]$.

- 加法函数性, 任意两个自然数的和唯一.

Lemma *plus_func*:

$[[ZF;;is\_natural\_number\ N;;is\_plus\ e\ N\vdash\forall x, \forall y, \forall z, \forall a, x\in N\wedge y\in N\wedge z\in N\wedge a\in N\wedge$
$in\_rel3\ x\ y\ z\ e\wedge in\_rel3\ x\ y\ a\ e\rightarrow z=a]]$.

- 加法交换律: 如果 $x+y=z$, 那么 $y+x=z$.

Lemma *plus_comm*:

$[[ZF;;is\_natural\_number\ N;;is\_plus\ e\ N\vdash\forall x, \forall y, \forall z, x\in N\wedge y\in N\wedge z\in N\wedge a\in N\wedge$
$in\_rel3\ x\ y\ z\ e\rightarrow in\_rel3\ y\ x\ z\ e]]$.

- 加法结合律: 如果 $x+y=a$, $y+z=b$, 那么有 $a+z=c$ 当且仅当 $x+b=c$, 即 $x+(y+z)=(x+y)+z$.

Lemma *plus_assoc*:

$[[ZF;;is\_natural\_number\ N;;is\_plus\ e\ N\vdash$
$\forall x, \forall y, \forall z, \forall a, \forall b, \forall c, x\in N\wedge y\in N\wedge z\in N\wedge a\in N\wedge b\in N\wedge c\in N\wedge$
$in\_rel3\ x\ y\ a\ e\wedge in\_rel3\ y\ z\ b\ e\rightarrow(in\_rel3\ a\ z\ c\ e\leftrightarrow in\_rel3\ x\ b\ c\ e)]]$.

最后给出乘法相关性质的证明.

- 乘法集合唯一性.

Lemma *mult_unique*:

$[[ZF;;is\_natural\_number\ N;;is\_plus\ e\ N\vdash\forall x, \forall y, is\_mult\ x\ e\ N\wedge is\_mult\ y\ e\ N\rightarrow x=y]]$.

- 对任意两个自然数, 都存在它们的乘积.

Lemma *in_mult_exists*:

$[[ZF;;is\_natural\_number\ N;;is\_plus\ e\ N;;is\_mult\ f\ e\ N\vdash\forall x, \forall y, x\in N\wedge y\in N\rightarrow\exists z, z\in N\wedge in\_rel3\ x\ y\ z\ f]]$.

- 乘法函数性, 任意两个自然数的乘积唯一.

Lemma *mult_func*:

$[[ZF;;is\_natural\_number\ N;;is\_plus\ e\ N;;is\_mult\ f\ e\ N\vdash$
$\forall x, \forall y, \forall z, \forall a, x\in N\wedge y\in N\wedge z\in N\wedge a\in N\wedge$
$in\_rel3\ x\ y\ z\ f\wedge in\_rel3\ x\ y\ a\ f\rightarrow z=a]]$.

- 乘法交换律: 如果 $x\times y=z$, 那么 $y\times x=z$.

Lemma *mult_comm*:

$[[ZF;;is\_natural\_number\ N;;is\_plus\ e\ N;;is\_mult\ f\ e\ N\vdash$
$\forall x, \forall y, \forall z, x\in N\wedge y\in N\wedge z\in N\wedge$
$in\_rel3\ x\ y\ z\ f\rightarrow in\_rel3\ y\ x\ z\ f]]$.

- 乘法结合律: 如果 $x\times y=a$, $y\times z=b$, 那么有 $a\times z=c$ 当且仅当 $x\times b=c$, 即 $(x\times y)\times z=x\times(y\times z)$.

Lemma *mult_assoc*:

$[[ZF;;is\_natural\_number\ N;;is\_plus\ e\ N;;is\_mult\ f\ e\ N\vdash$
$\forall x, \forall y, \forall z, \forall a, \forall b, \forall c, x\in N\wedge y\in N\wedge z\in N\wedge a\in N\wedge b\in N\wedge c\in N\wedge$
$in\_rel3\ x\ y\ a\ f\wedge in\_rel3\ y\ z\ b\ f\rightarrow(in\_rel3\ a\ z\ c\ f\leftrightarrow in\_rel3\ x\ b\ c\ f)]]$.

- 乘法分配律: 如果 $x\times y=a$, $x\times z=b$, $x\times d=c$, $y+z=d$, 那么有 $a+b=c$, 即 $x\times y+x\times z=x\times(y+z)$.

Lemma *mult_dist*:

$[[ZF;;is\_natural\_number\ N;;is\_plus\ e\ N;;is\_mult\ f\ e\ N\vdash$
$\forall x, \forall y, \forall z, \forall a, \forall b, \forall c, \forall d, x\in N\wedge y\in N\wedge z\in N\wedge a\in N\wedge b\in N\wedge c\in N\wedge d\in N\wedge$
$in\_rel3\ y\ z\ d\ e\wedge in\_rel3\ x\ y\ a\ f\wedge in\_rel3\ x\ z\ b\ f\wedge in\_rel3\ x\ d\ c\ f\rightarrow in\_rel3\ a\ b\ c\ e]]$.

6 与相关工作的比较

对学生而言, 本工作开发的定理证明工具是封闭的, 即使是在 Coq 中使用我们开发的工具完成了数学归纳法证明的学生, 也无需了解任何 Coq 中最基础、与课程教学内容无关的知识, 如归纳类型, 这与现有的工作相当不同. Avigad 在针对本科生的逻辑课程中, 使用 Lean 进行了朴素集合论和公理化集合论的教学. 课程直接采用了 Lean 的内置逻辑形式化了集合论相关概念, 这种方法无需太多的开发工作, Lean 本身的交互也较为友善, 在学生已经熟知如何使用 Lean 后, 也能够成功完成集合论相关定理的证明. 不过, 作者需要拿出专门的课时进行 Lean 的使用教学, 将其作为课程内容的一部分. 由于缺乏改进的证明策略, Lean 中的证明略显冗长, 因此, 作者只要求学生使用 Lean 证明较简单的例子, 这适用于不在一阶逻辑的框架下直接介绍集合论相关概念的教学场景, 在一阶逻辑的框架下直接进行形式化证明会太过复杂. 作者希望 Lean 中能够提供更多的自动化, 以便将来学生能够更简单地完成更多证明. 而本工作利用 Coq 中的 Ltac 开发了较方便的证明策略, 并通过 notation 提供了传统的逻辑符号, 学生根据 15 分钟的教程视频, 就能够初步掌握我们开发的自动化证明策略, 快速完成数学归纳法等重要定理的证明. 学习使用该工具的成本是很低的, 因此, Coq 并不需要作为课堂教学的内容.

另据我们所知, 目前国内尚无将交互式定理证明器引入集合论教学中的尝试.

7 总 结

交互式定理证明工具能够帮助学生更好地学习数理逻辑, 给予学生快速且准确的支持. 针对现有的定理证明器上手门槛高、与教科书差别大的问题, 我们在 Coq 中开发了一个公理集合论证明器, 并为学生提供了与教科书证明类似的证明环境. 我们从逻辑公式的抽象语法树开始形式化了 ZFC 公理集合论的推理系统, 并针对该推理系统提供了数条自动化证明策略, 包括使用了析取范式生成、调用在 Coq 中实现的 DPLL 可满足行求解器的命题逻辑自动证明策略 FOL_Tauto、支持多层量词的引入消去以及自动搜索的量词证明策略和更简单的等号替换证明策略. 相比直接使用现有定理证明器, 我们提供的封闭证明环境中隐藏了原生定理证明器的内置逻辑, 贴近教科书风格, 使学生能够专注于他所学习的逻辑. 自动化证明策略也减轻了学习使用定理证明器带来的不必要学习负担, 易于理解和使用. 在实际教学中, 学生仅使用我们提供的几条证明策略, 就可以证明数学归纳法和皮亚诺算术系统等复杂定理, 体现了该工具的易用性和有效性. 需要说明的是: 本工具针对专门的教学场景开发, 面对不同的教学需求, 可能还需要根据实际情况进行调整和改变.

除了数理逻辑, 交互式定理证明工具也适用于其他教学场景. 目前, 在教学中使用较多的交互式定理证明器有 Coq 和 Lean: Lean 对形式化数学有更好的支持, 如 Thoma 和 Iannone[22]使用 Lean 进行数论的教学; 而 Coq 则更适合计算机相关理论的形式化. Pierce 等人撰写了著名的 Software Foundations[23], 涵盖了程序逻辑、程序语义等概念, 在程序语言理论课程中得到了广泛应用. 但在数学课程中, 交互式定理证明工具的使用依然集中于数理逻辑的教学, 其应用尚待进一步探索.

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, *et al*. Mizar: State-of-the-art and beyond. In: Kerber M, *et al*. eds. Proc. of the Int'l Conf. on Intelligent Computer Mathematics (CICM 2015). 2015. 261−279. [doi: 10.1007/978-3-319- 20615-8_1]

[4] Moura L, Kong S, Avigad J, *et al*. The Lean theorem prover (system description). In: 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 LC. 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.

[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, *et al*. 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 Proof Technology in Mathematics Research and Teaching. Mathematics Education in the Digital Era. Cham: Springer, 2019. 277−290. [doi: 10.1007/978-3-030-28483-1_13]
[12] Breitner J. Visual theorem proving with the incredible proof machine. In: Proc. of the Int'l Conf. on Interactive Theorem Proving. 2016. 123−139. [doi: 10.1007/978-3-319-43144-4_8]
[13] Lerner S, Foster SR, Griswold WG. Polymorphic blocks: Formalism-inspired UI for structured connectors. In: 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. A mechanized proof of equivalence between the axiom of choice and Tukey's lemma. Journal of Beijing University of Posts and Telecommunications, 2019, 42(5): 1−7 (in Chinese with English abstract). [doi: 10.13190/j.jbupt.2019-001]
[17] Li Q. On theteaching reform of mathematical logic experiment course for computer majors. Journal of Anhui University of Technology (Social Sciences), 2011, 28(3): 112−113 (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): 111−115 (in Chinese). [doi: 10.16512/j.cnki.jsjjy.2021.01.027]
[19] Ebbinghaus HD, Flum J, Thomas W, *et al*. 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, *et al*. Software foundations. 2010. http://www. cis.upenn.edu/bcpierce/sf/current/index.html

附中文参考文献:

[15] 严升, 郁文生, 付尧顺. 基于 Coq 的杨忠道定理形式化证明. 软件学报, 2022, 33(6): 2208−2223. http://www.jos.org.cn/1000-9825/6578.htm [doi: 10.13328/j.cnki.jos.006578]
[16] 孙天宇, 郁文生. 选择公理与 Tukey 引理等价性的机器证明. 北京邮电大学学报, 2019, 42(5): 1−7. [doi: 10.13190/j.jbupt.2019-001]
[17] 李沁. 面向计算机专业的数理逻辑实验课的教学改革. 安徽工业大学学报(社会科学版), 2011, 28(3): 112−113.
[18] 江南, 何炎祥. 关于计算机专业开设逻辑与验证类课程教学的思考. 计算机教育, 2021(1): 111−115. [doi: 10.16512/j.cnki.jsjjy.2021.01.027]

万新熠(2000－), 男, 硕士生, CCF 学生会员, 主要研究领域为数理逻辑的形式化.

曹钦翔(1990－), 男, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为基于定理证明的程序验证, 程序逻辑.

徐轲(2000－), 男, 硕士生, CCF 学生会员, 主要研究领域为基于定理证明的程序验证.