arXiv:2601.04653v1 [cs.AI] 8 Jan 2026

大语言模型驱动的定理证明器的氛围编程

Zhe Hou格里菲斯大学
z.hou@griffith.edu.au

摘要

我们介绍了 Isabellm，这是一款面向 Isabelle/HOL、能够全自动进行证明综合的大语言模型驱动的定理证明器。Isabellm 可与 Ollama 上的任意本地大语言模型以及 Gemini CLI 等 API 配合使用，并且专为在消费级计算机上运行而设计。该系统结合了逐步证明器和更高层次的证明规划器：逐步证明器利用大语言模型提出证明命令，并在有界搜索循环中由 Isabelle 验证；证明规划器则生成结构化的 Isar 证明大纲，并尝试填充和修复剩余空缺。该框架包含用于策略的束搜索、策略重排序器机器学习和强化学习模型、基于小型 Transformer 模型的前提选择、由 AFP 构建的 Isar 证明微型检索增强生成，以及反例引导的证明修复。所有代码均由 GPT 4.1 - 5.2、Gemini 3 Pro 和 Claude 4.5 编写。实证结果表明，Isabellm 能够证明某些连 Isabelle 标准自动化（包括 Sledgehammer）都无法攻克的引理，展现了大语言模型引导的证明搜索的实际价值。与此同时，我们发现，即使是 GPT 5.2 Extended Thinking 和 Gemini 3 Pro 等最先进的大语言模型，也难以可靠地实现预期的填充与修复机制，尤其是在涉及复杂算法设计时；这凸显了大语言模型代码生成和推理面临的根本挑战。Isabellm 的代码可在 https://github.com/zhehou/llm-isabelle 获取。

1 引言

自动推理长期以来一直是计算机科学的核心目标，涵盖自动定理证明、模型检测、符号执行和模理论可满足性。经典推理引擎在范围明确的领域中取得了显著成功，例如命题逻辑和一阶逻辑；数十年来的算法进步使 SAT 和 SMT 求解器得以高度优化。然而，随着底层逻辑表达能力的增强，实现完全自动化也变得困难得多。高阶逻辑、丰富的类型系统和庞大的数学库带来了广阔的搜索空间、微妙的依赖关系，以及难以通过纯符号方法捕捉的非局部推理模式。因此，许多实用推理系统采用混合策略，将自动化与人工引导相结合，以管理复杂性，同时保持可靠性。

Isabelle/HOL 等交互式定理证明器 [7] 体现了这种权衡。它们提供了极强的正确性保证：只有当证明通过小型可信内核的类型检查时，才会被接受，因此生成的产物比传统测试或非正式论证可靠得多。严谨性所带来的代价众所周知。证明开发劳动密集，需要专门的专业知识，而且即使有强大的自动化工具辅助，通常仍需谨慎协调证明方法和辅助引理。Isabelle 生态系统通过 Sledgehammer 等工具大幅减轻了这一负担；Sledgehammer 结合前提选择、外部自动证明器和证明重构，能够解决许多常规目标 [2, 1]。Nitpick 等辅助工具也能帮助用户快速反驳错误猜想或揭示缺失的假设。然而，对于任意高阶目标，实现完全自动的“一键式”证明仍然

遥不可及：搜索空间呈组合爆炸式增长，相关上下文可能涉及数千条事实，而正确的证明方法序列则高度依赖具体问题。

大语言模型（LLM）的最新进展从另一个角度重新开启了这一挑战。大语言模型不再仅仅依赖符号启发式方法，而是作为强大的条件生成器，根据从大型语料库中学习到的模式，提出合理的证明步骤、策略，甚至完整的证明脚本。这一能力催生了新一代神经符号定理证明器，它们将生成模型与形式化验证紧密结合。在这些系统中，语言模型通过提出候选方案来探索证明空间，而证明助手则通过接受、拒绝或部分验证这些方案，提供精确且可执行的反馈。早期研究在 MetaMath 等场景中证明了这一方法的可行性：生成模型与引导式搜索和验证循环相结合 [10], 。后续系统引入了结构更完善的搜索和学习机制，其中包括超树证明搜索（HTPS），它将定理证明构建为在动态扩展的搜索树上进行迭代改进的过程 [6]。

在 Isabelle 生态系统中，程序化接口和基准测试集为大规模交互与学习提供了支持，推动了这一研究方向的发展，其中最具代表性的是 LISA 和用于增量式证明执行与数据提取的 PISA 协议 [4]。在这些基础上，近期面向 Isabelle 的系统探索了完整证明生成、验证器引导的修复，以及与 ATP 工具的深度集成，Baldur 和 Thor 即为例证 [3, 5]。与此同时，LeanDojo 等面向 Lean 的其他证明助手研究凸显了在大型库中进行检索和前提选择的关键作用，也展现了标准化基准测试集（如 miniF2F）对于比较评估的重要性 [9, 11]。

本项目探索这样一个问题：大语言模型能否编写由大语言模型驱动的定理证明器？这个“循环”问题从两个方面考察最新大语言模型的能力：它们能否用于编写定理证明器这类复杂算法？它们能否作为定理证明器的一部分，用于生成形式化证明？最终目标是实现全自动证明综合：给定一个目标陈述（及导入项），系统应在无需用户交互的情况下，输出完整且可检查的 Isabelle 证明。此外，我们希望让这一框架适合在**笔记本电脑上运行**——目前所有代码都已在一台配备 M1 Pro 处理器和 32GB 内存的 2021 款 MacBook Pro 上测试。我们希望构建一个普通 Isabelle/HOL 用户能在消费级电脑上运行的大语言模型证明器，而不是只能运行在昂贵 GPU 服务器上的系统。具体来说，该代码库实现了 Isabellm，这是一个由两个协同层组成的模块化流水线。第一层是逐步证明器，将证明视为序列决策过程：它从证明状态出发，提出候选策略或证明方法，在 Isabelle 中检查这些方法，并通过搜索（例如束式探索）到达终止的已解决状态。这一层将经典 Isabelle 自动化（包括 Sledgehammer、可满足性模理论/自动定理证明后端以及反例检查）与策略重排序、前提选择等学习组件相结合，使证明器能够突破纯提示词在上下文 [2]中所能达到的能力。第二层是规划器，尝试生成结构化的 Isar 风格证明提纲，然后调用逐步证明器解决由此产生的子目标。该规划层的设计依据是：分解和草图构建能够缓解短视的逐步搜索，尤其对于较长的证明；这也与近期文献中的“先构建草图”和递归证明主题相契合 [5, 8]。总体设计旨在支持严谨评估（基准测试和回归测试）、可复现性（通过脚本进行证明检查），以及借助日志和学习得到的模型实现持续改进。

本报告的其余部分将介绍系统架构、接口和学习组件，并将 Isabellm 置于既有神经符号证明器的背景下进行定位。我们将重点讨论使 Isabelle 中的全自动证明生成切实可行所需的技术选择：可靠的证明状态交互、快速验证、库规模的前提管理、严格时间预算下的搜索控制，以及源自验证器结果的训练信号。我们还将强调当前的局限性，尤其是规划和证明修复方面的局限，并探讨如何将验证器反馈转化为系统性改进，而不是临时性的提示调优。

2 基于大语言模型的逐步证明器

逐步证明器是系统的核心自动化证明引擎。它的目标是针对给定目标综合出完整的 Isabelle/HOL 证明脚本，方法是迭代提出证明命令、使用 Isabelle 验证这些命令，并对由此产生的证明状态进行有界搜索。它面向小步证明，基本上发挥着与 Sledgehammer 相同的作用。

2.1 准备

证明综合被表述为在 Isabelle 证明脚本空间中的有界搜索。证明脚本表示为一个有序序列

$$S = \langle s_0, s_1, \ldots, s_k \rangle,$$

其中， s_0 是形式为 lemma "<目标>" 的引理声明，而每个 s_i for $i \geq 1$ 都是一个 Isabelle 证明命令，通常是 apply 风格策略。

在每次迭代中，证明器都会维护一个宽度固定为 B 的候选证明状态束。每个束元素都是一个元组

$$(\sigma, S, h, n),$$

其中：

- S 是当前已接受的证明前缀，
- σ 是用于对束条目进行排序的标量分数，
- h 是表示当前证明状态的文本提示（通过打印状态获得），
- n是 Isabelle 报告的剩余子目标数量。

主要评分指标是剩余子目标的数量 n。优先选择子目标较少的状态，这反映了一种启发式判断：局部进展与全局证明完成度相关。当无法提取 n 时，将使用一个较大的哨兵值。

搜索最多进行 D 个扩展轮次（最大深度），或直到证明完成，或达到全局墙钟超时。

Eac每条候选证明命令都通过构造一个临时 Isabelle 理论并调用 -
ing Isabelle，通过持久服务器连接。对于证明前缀

$$S = \langle s_0, s_1, \ldots, s_k \rangle$$

以及候选命令 c，证明器会构造如下形式的理论

```
theory Scratch imports Main begin
```

$$s_0$$
$$s_1$$
$$\vdots$$
$$s_k$$
$$c$$

后面接以下内容之一：

- 中间步骤使用打印状态和 sorry，或者
- 证明完成器后面不接命令。

该理论通过 Isabelle 的 use theories 命令执行。如果 Isabelle 接受该理论且没有报错，就认为候选步骤成功。对于中间步骤，证明器会提取打印出的证明状态，并解析剩余子目标的数量。

为避免重复探索，会使用如下计算得出的指纹对证明状态进行去重：

$$\mathsf{fp}(h) = \mathsf{SHA1}(\mathsf{normalize}(h)),$$

whe关于规范化：它会移除无关的空白字符。每个指纹仅保留一个束条目 .

2.2 通过语言模型进行候选生成

候选证明命令由充当逐步搜索循环中条件提议机制的大语言模型（LLM）生成。LLM 并不是用于一次性生成完整证明；相反，系统会反复调用它们，提出规模较小且局部有效的证明命令，然后由 Isabelle 对这些命令进行检查。这种设计确保所有正确性保证都由证明助手而非语言模型来执行。

调用模型。 LLM 通过统一抽象层进行调用，该抽象层同时支持本地模型和远程模型（例如通过 HTTP 或命令行接口）。每次调用都会指定：

- 定义允许使用的动作语法的系统提示，
- 用于编码当前证明上下文的用户提示，
- 解码参数，包括温度、最大令牌数和样本数量。

证明器不假设任何特定的模型架构。相反，它将大语言模型视为黑盒条件生成器，将文本形式的证明上下文映射为一组简短的候选证明命令。

提示模式。我们使用两种不同的提示模式，分别对应证明搜索的不同阶段：

1. **步骤模式**：生成以下形式的中间证明命令

   ```
   apply <tactic>.
   ```

   这些命令旨在缩减或转换当前的子目标集合，而不闭合证明。

2. **完成模式**：生成闭合证明命令，例如

   ```
   by simp,  by auto,  by (metis ...),  done.
   ```

   当证明器认为剩余的子目标可能只需一步即可解决时，就会调用完成模式。

每种模式都使用不同的系统提示，以严格约束输出的句法形式。

系统提示。系统提示会明确指示模型：

- 输出多个候选命令（通常为 3 到 8 个），
- 每行输出一条命令，
- 将输出限制为固定的语法格式（步骤模式下使用 apply 命令，完成模式下使用 by/done），
- 避免提供解释、注释或自然语言文本。

通过在提示级别施加这些约束，证明器减轻了下游解析的负担，并显著降低了无效 Isabelle 调用的发生率。

用户提示构造。 用户提示将当前证明上下文编码为一个结构化文本块，其中包含：

1. **目标陈述：** 待证明的原始引理。 2. **已接受的证明步骤：** 已通过 Isabelle 验证的证明命令前缀。 3. **当前证明状态：** 最新的打印状态输出，展示剩余子目标。 4. **有用事实：** 通过前提挖掘和检索获得的精选引理名称列表。

-

这些信息以一致、机器可读的格式呈现，以稳定模型在不同证明状态下的行为。证明状态按原样纳入，以呈现剩余目标的逻辑结构；有用的事实则提供软性引导，而不施加硬性约束。

解码控制与探索。大语言模型的解码具有随机性，主要通过温度进行控制。为平衡探索与利用，证明器会根据搜索停滞情况动态调整温度。

令 s 表示最小子目标数连续未改善的搜索深度次数。解码温度调整如下：

$$T_{\text{step}} = \min(0.9,\ 0.5 + 0.1s), \qquad T_{\text{finish}} = \min(0.6,\ 0.2 + 0.05s).$$

停滞程度较低时，温度保持较低，以优先选择确定性强、概率较高的命令。随着停滞程度加深，温度随之升高，以鼓励生成的命令在语法和语义上更多样。

候选采样。 每次调用时，都会要求模型在单次调用中生成一小批候选项。这比重复进行单样本调用更高效，也使证明器能够利用批次内部的多样性。每次调用的候选项数量固定，且不受束宽影响，从而确保计算成本可预测。

后处理与清理。 语言模型的原始输出会经过严格的后处理：

- 代码块、编号、项目符号和多余的空白均会被移除；
- 仅保留以批准的前缀开头的行；
- 超出固定字符数限制的命令将被丢弃；
- 重复的命令将被移除，同时保留原有顺序。

任何不符合预期语法的输出都会被静默忽略。这种保守的过滤机制确保只有语法上合理的 Isabelle 命令会进入耗时的验证阶段。

与重排序的集成。 如果有学习型重排序器可用，每个候选命令都会立即进行特征提取和评分，然后再交由 Isabelle 评估。重排序器的分数用于重新排列候选项，以便优先检查更有希望的命令。这种交互使学习能够影响搜索，同时不绕过符号验证。

启发式变体注入。 当停滞程度超过阈值时，证明器会用启发式模板补充大语言模型生成的候选项，例如：

apply (对 xs 归纳)，apply (对 x 分类讨论)，apply (应用某个引理).

这些模板使用从当前证明状态中提取的变量名称实例化。即使大语言模型未能明确提出相关结构，这一机制也能注入与领域相关的结构，充当轻量级回退方案。

设计依据。 候选生成机制刻意限制了语言模型的表达能力。通过将输出限制为简短的语法约束命令，并对每个提议进行符号验证，系统将大语言模型从一个不可靠的证明生成器转变为局部证明操作的概率提议分布。这种设计使整体证明器保持可靠、易于调试，并适于通过基于学习的改进加以提升。

2.3 束扩展与评分

对于每个束状态，证明器都会生成一组有限的候选后续步骤。每个候选项都会由 Isabelle 独立验证。验证成功的候选项会生成新的束条目，并更新证明前缀和证明状态。

令 $\mathcal{C}$ 为当前束中所有成功扩展构成的多重集。下一束通过按字典序对 $\mathcal{C}$ 排序来构造：

$$(n,\ |S|),$$

其中 n 是剩余子目标的数量， $|S|$ 是证明脚本的长度。保留排名最靠前且指纹各不相同的 B个条目。

这会在证明状态上进行最佳优先搜索，并受束宽和深度的限制。

我们曾测试蒙特卡洛树搜索，但没有采用，因为它比束搜索慢得多；此外，如上所述，逐步证明器旨在证明较小的步骤目标，而非复杂目标。

2.4 基于学习的策略重排序

逐步证明器采用基于学习的重排序器，对语言模型提出的候选证明命令进行排序偏置。重排序器作为轻量级评分函数运行，用于预测候选命令应用于当前证明状态时取得进展的可能性。重要的是，重排序器不会 替代符号验证：所有候选命令仍由 Isabelle 检查。它的作用仅仅是降低搜索的有效分支因子。

重排序接口。 在运行时，重排序器以函数的形式提供

$$f_\theta : \mathbb{R}^d \to [0, 1],$$

其中 d是特征维度，而 $f_\theta(x)$用于估计候选步骤成功的概率。分数越高，在束扩展中的优先级就越高。重排序分数通过调整启发式分数纳入候选命令排序：

$$\mathrm{score}(c) = \mathrm{heuristic}(c) - \lambda \cdot f_\theta(x_c),$$

其中， $\lambda > 0$ 是固定权重（通过环境变量配置）， x_c 是为候选项 c提取的特征向量。

特征表示。 每个候选步骤都表示为一个固定长度的数值特征向量，由四组特征组成：

1. **搜索上下文特征：**

   $$(\text{depth},\ n_{\text{sub}},\ t_{\text{elapsed}},\ \text{cache_hit}),$$

   其中，深度是当前搜索深度， n_{sub} 是剩余子目标的数量（如果已知），缓存命中表示重用了缓存的 Isabelle 结果。
2. **目标和状态标志：** 从目标和证明状态中提取的二元指标，包括：

   $$\{\text{is_listy},\ \text{is_natty},\ \text{is_sety},\ \text{has_quantifier},\ \text{is_boolean}\}.$$

   这些特征概括了目标的粗略结构属性。
3. **策略前缀编码：** 针对固定词汇表中的策略前缀进行独热编码（例如，apply simp、apply auto、apply (归纳)、apply (分类讨论)、apply (规则)等）。这表示所提议命令的句法类别。
4. **前提交互特征：** 从前提选择中提取的数值摘要（稍后介绍），包括余弦相似度统计信息以及候选项之间的重叠-引用的引理和检索到的前提。

最终的特征向量会填充或截断至训练好的模型所需的固定维度。这使重排序器能够稳健地处理不断扩展的特征。

训练数据。 重排序器的训练数据是在证明搜索过程中自动收集的。每条尝试的候选命令都会生成一个带标签的样本

$$(x_c, y_c),$$

其中 x_c 是上文所述的特征向量，且

$$y_c = \begin{cases} 1 & \text{if Isabelle accepts the step,} \\ 0 & \text{otherwise.} \end{cases}$$

成功和失败的尝试都会被记录下来。这会生成一个极度不平衡但规模庞大的数据集，反映真实的搜索行为，而非精选证明。

监督学习。该系统支持多个监督学习后端：

- **逻辑回归：** 一种便于快速迭代且易于解释的线性基线。
- **梯度提升树（XGBoost）：** 用于建模特征之间的非线性交互。

对于概率分类器，直接将预测概率 $P(y = 1 \mid x)$ 用作 $f_\theta(x)$。对于回归器，输出会归一化至 $[0, 1]$。

离线强化学习。 除了监督学习之外，系统还支持基于记录的轨迹进行离线强化学习。在这种表述中，证明搜索被视为一个马尔可夫决策过程：

$$(s_t, a_t, r_t, s_{t+1}),$$

其中， s_t 是证明状态， a_t 是候选策略，奖励 r_t 则根据子目标减少量或证明完成情况得出。

实现了两种算法：

1. **优势加权回归（AWR）：** 根据估算的优势为候选项加权，优先选择能更快减少子目标的步骤。2. **拟合Q学习（DQN风格）：** 从记录的转移中学习动作价值函数 $Q(s, a)$，并将其用作重排序信号。

学习得到的模型会导出为 TorchScript 模块或 joblib构件，并在运行时动态加载。

设计依据。 重排序器刻意设计得浅而快速。它的目的并非独立证明定理，而是编码在 Isabelle 中哪些做法通常有效的经验规律，让符号证明器能够将计算预算集中在有希望的分支上。

2.5 使用神经编码器进行前提选择

前提选择用于为证明器和语言模型提供一小部分相关引理，这些引理来自周边理论和库。系统不再单纯依赖 Isabelle 的内部启发式方法，而是实现了一个基于检索的前提选择器，并可选用神经编码器。

前提索引。 所有候选前提都以成对形式存储在内存索引中

$$(\ell_i, t_i),$$

其中， ℓ_i 是引理标识符， t_i 是其文本表示。每个前提还可以附带元数据，例如源文件和局部上下文。

两阶段检索。 前提选择分两个阶段进行：

1. **选择阶段：** 快速、面向召回的检索，生成一个包含 K_{select} 个候选前提的集合。2. **重排序阶段：** 可选的、面向精确率的重新评分，对排名前 K_{rerank} 的前提进行处理。

选择阶段编码器。 选择阶段支持三种后端：

- **TF–IDF 余弦相似度：** 当 scikit-learn 可用时，使用 TF–IDF 对前提进行向量化，并按余弦相似度评分。
- **词元重叠（回退方案）：** 当没有可用的外部库时，会对词元集合计算杰卡德重叠。

• **神经双编码器：**当存在训练好的编码器时，使用句子转换器模型将目标和前提嵌入共享向量空间中。余弦相似度的计算方式为

$$\mathrm{sim}(g,p) = \frac{\langle e(g), e(p)\rangle}{\|e(g)\|\|e(p)\|}.$$

所有嵌入向量都在索引定稿期间计算一次，并缓存在内存中。

重排序阶段。 重排序阶段可选择使用交叉编码器，对成对的对象（g，p）进行联合评分。给定一批候选前提 $\{p_i\}$ 和一个目标 g，交叉编码器会计算分数

$$r_i = \mathrm{CE}(g, p_i),$$

这些分数用于对前提重新排序。如果没有可用的重排序器，则重复使用选择分数。

训练前提编码器。 前提选择的训练数据取自成功的证明尝试。对于给定的目标 g:

- 正前提是在成功的证明步骤中明确引用的前提。
- 负前提从检索池中抽取，但未用于证明。

双编码器采用对比学习（例如多重负样本排序损失）进行训练，促使模型

$$\mathrm{sim}(g,p^+) > \mathrm{sim}(g,p^-)$$

将正前提 p^+ 与负前提 p^-区分开来。交叉编码器则通过监督回归或分类，基于 $(g,\ p)$ 对进行训练。

与证明搜索的集成。 选出的前提有两种用途:

1. 将其原样注入语言模型提示词，作为上下文提示。2. 将汇总统计信息（最高相似度、平均相似度、与候选引用引理的重叠度）附加到重排序器特征向量中。

这种紧密耦合使前提选择与策略重排序能够相互促进，而不受硬性约束。

设计依据。 前提选择被视为一种软引导机制，而非硬过滤器。通过让语言模型和重排序器都能利用检索到的前提，系统既能从检索中获益，又能在面对检索错误时保持稳健；对于规模庞大且不断演进的 Isabelle 库而言，这一特性至关重要。

2.6 后处理

在执行开销较大的 Isabelle 检查之前，可以使用轻量级反驳工具剪除候选步骤。系统会选择性地调用 Quickcheck 和 Nitpick 来检测反例。若任一工具发现反例，则丢弃该候选项，不再进行后续评估。

这种剪枝会谨慎地应用，主要针对涉及布尔结构或量词的目标，并以可配置的间隔执行，以控制开销。

为减少重复调用 Isabelle，证明器采用两级缓存:

- 单次运行缓存，按 (S, c) 为键，
- 一个在多次运行之间共享的全局有界缓存。

Cach条目存储成功标志、子目标计数、证明状态提示和时间信息。缓存命中会被记录下来，并在之后用作重排序器训练期间的特征。

找到成功的证明后，证明器会尝试对其进行简化。最小化过程会在较短的超时时间内，贪心地应用以下变换:

1. 如果可能，将证明压缩为单行证明；2. 从
simp 或 metis 调用中移除未使用的事实；3. 删
除多余的中间步骤；4. 再次尝试单行证明。

此外，证明器会尝试使用简单的骨架模板，将非结构化的 apply 风格证明转换为结构化的 Isar 证明，在不牺牲自动化程度的前提下提升可读性。

3 LLM 驱动的证明规划器

本节介绍证明规划器，它的抽象层次高于逐步证明器。逐步证明器对单条证明命令进行有界搜索，而证明规划器则围绕结构化的 *Isar* 证明进行推理，明确地将复杂目标分解为中间断言和子证明。规划器旨在综合出一个证明大纲，以把握论证的全局结构，然后系统地消除所有剩余空缺，直至获得完全验证通过的证明。

形式化地说，给定一个目标公式 G，规划器要寻找一个证明脚本

$$P = \langle s_0, s_1, \ldots, s_m \rangle$$

使得 s_0 是 G 的引理声明， P 是一个格式良好的 Isar 证明，并且 Isabelle 验证 P 时不存在未解决的空缺。与逐步证明器不同，中间脚本可以包含占位符（sorry），这些占位符会被视为显式的未知项，并在后续阶段填充。

3.1 高层规划模型

规划器以两种模式之一运行：

- **大纲模式**，该模式返回结构化的 Isar 证明骨架，其中可能包含空缺。
- **自动模式**，该模式会迭代执行大纲生成、空缺填充、修复和重新生成，直到生成经过验证的证明，或耗尽全局预算。

在整个规划过程中，Isabelle 都通过持久服务器会话访问。令 $\mathcal{I}$ 表示此会话。所有大纲检查、填充和验证步骤都针对 $\mathcal{I}$ 执行，从而摊销初始化成本，并能在预算紧张的情况下重复进行理论构造。

多样化大纲采样。大纲生成通过查询语言模型来完成，以生成候选 Isar 骨架。令 $\mathcal{T} = \{t_1, \ldots, t_r\}$ 为一组固定的采样温度。对于每个 $t_i \in \mathcal{T}$，规划器最多采样 k 个候选大纲，得到一个多重集

$$\mathcal{S} = \bigcup_{i=1}^{r} \mathcal{S}_{t_i},$$

其中 $\mathcal{S}_{t_i}$ 是温度为 t_i 时的样本。这种机制近似于从证明结构上的混合分布中进行采样，以牺牲确定性为代价换取多样性。

提示级条件化。 每次采样调用都以以下内容为条件：

1. 目标 G，2. 一组可选的推荐引理 $H = \{h_1, \ldots, h_m\}$，
3. 一项要求提供单个 Isar 证明代码块的系统约束。

提示会在生成之前注入，使其能够影响全局结构（例如归纳变量的选择），而不仅仅是局部步骤。

规范化为规范骨架。原始模型输出会被规范化为规范骨架

$$S = (\text{header}, \text{body}, \text{holes}),$$

其中：

- 头部被强制与 G 匹配，
- 主体是语法完整的 Isar 证明，
- 所有缺失的证明依据都替换为 sorry，
- 启用大纲强制执行时，inlineproofs(by ...) 可能会被改写为空缺。

这种规范化保证每个候选大纲都有一组定义明确的空缺，并且可以由 Isabelle 检查，不会产生语义歧义。

空洞提取。令 holes(S) $= \{h_1, \ldots, h_\ell\}$ 表示 S 中最大 sorry 片段的集合。填充和修复过程中，每个空缺 h_i 都被视为一个独立的子问题。
规划器提供以下参数以进行实验控制：

- 大纲多样性参数 (k, $\mathcal{T}$)，
- 强制使用显式空缺，
- 使用结构模板，
- 提示来源和检索限制，
- 评分权重（α, β, γ），
- 阶段上限（c_1, c_2）和全局时间预算。

这些参数共同定义了一个涵盖证明结构和修复策略的搜索空间，从而能够系统地评估规划效果。

3.2 Micro-RAG：用于规划的轻量级检索增强指导

证明规划器采用了一种经过刻意精简的检索增强生成（RAG）形式，称为*micro-RAG*，旨在为大纲生成和修复提供偏向引导，同时避免引入重量级神经检索或大型外部索引。其设计目标是提供结构和语义提示，使其计算成本低、跨运行稳定，并适用于在笔记本级环境中执行。
与逐步证明器中的前提选择不同，后者与策略层决策紧密耦合，micro-RAG 在证明规划层面运行。它的输出会影响提出哪些证明结构以及如何修复空缺，但绝不会成为硬性约束。

3.2.1 提示来源与表示

Micro-RAG 从两个独立来源汇总提示：

$$H = H_{\text{ctx}} \cup H_{\text{lex}},$$

其中每个提示 $h \in H$ 都是一个符号标识符（通常是引理名称或定理常量）。

上下文衍生提示。上下文衍生的提示集 H_{ctx} 直接从 Isabelle 中提取。给定一个目标 G，规划器会构造一个最简理论来打开该目标，并请求打印证明状态。从这个状态块中，它会提取：

- 局部绑定事实，
- 假设，
- 当前上下文中可用的先前引入的引理。

令 CtxFacts(G) 表示出现在此状态中的符号多重集。规划器会应用语法规范化（移除重复项、去除限定符、过滤平凡事实），并将列表截断至固定预算 k_{ctx}。所得集合为

$$H_{\text{ctx}} := \mathsf{TopK}(\mathsf{CtxFacts}(G), k_{\text{ctx}}).$$

这些提示反映了 Isabelle 自身认为在局部相关的内容，因此在引导归纳变量或分类讨论等结构性证明选择时尤其有效。

词典派生提示。词典派生提示集 H_{lex} 来自预先计算的提示词典，并以 JSON 映射的形式存储：

$$\mathcal{L} : \text{token} \mapsto \{(h, w_h)\},$$

其中，每个条目都将一个词元（例如函数名、构造器、类型名）与一个带权重的引理标识符列表关联起来。

在运行时，规划器会将目标 G 标记化为一个多重集 Tok(G)。对于 $\mathcal{L}$ 中出现的每个词元 $t \in \text{Tok}(G)$，它都会检索关联的引理列表并累积分数：

$$\text{score}(h) = \sum_{t \in \text{Tok}(G)} w_{t,h}.$$

随后，词典派生提示集为

$$H_{\text{lex}} := \mathsf{TopK}\big(\{(h, \text{score}(h))\},\ k_{\text{lex}}\big),$$

其中，k_{lex} 对应 hintlex top。

该词汇表本身是通过离线分析大型 Isabelle 语料库（例如 AFP）构建的，方法是关联目标中的词元与成功的 Isar 证明中出现的引理。重要的是，这一过程完全基于符号，不需要神经编码器。

3.2.2 提示聚合与规范化

组合后的提示集构造如下：

$$H := \mathsf{Dedup}\big(H_{\text{ctx}} \cup H_{\text{lex}}\big),$$

随后截断至全局上限 k_{hint}。规划器按来源优先级保留相对顺序，通常优先采用从上下文中提取的提示，而非词汇表提示。

此处并不试图确保 H 的完备性或最优性。Micro-RAG 明确采用启发式方法，侧重稳定性和低方差，而非最大限度地提高召回率。

3.2.3 集成到大纲生成中

在大纲生成期间，提示集 H 会在提示级别注入。具体而言，系统提示保持不变，而用户提示则包含一条明确的偏好条款：

`HINTS: Prefer using` $h_1, \ldots, h_m$ `if applicable.`

这条指令仅供参考，并非强制要求。语言模型可以自由忽略提示，但经验表明，这种软条件化往往会影响高层次的证明选择，例如选择一种归纳原理或重用一个特征引理。

提示会在大纲采样之前 注入，使其能够影响全局结构，而不是在修复期间再补入。

3.2.4 集成到修复与重新生成中

Micro-RAG 也用于修复和完整证明再生成阶段。在这些阶段，提示有两种作用：

1. **修复条件化：** 重新生成代码块或子证明时，提示集会与有效目标和错误诊断一同提供，引导大语言模型倾向于使用已知的有用引理。

2. **评分信号：** 在大纲中使用提示引理，会通过提示奖励项对大纲的综合评分产生正向贡献。

-

形式上，设 Use(S, H) 为 H 中在骨架S 语法上出现的不同提示数量。奖励项计算如下：

$$\text{hint_bonus}(S) = \min\big(\text{Use}(S, H),\ k_{\text{hint}}\big),$$

从而确保影响有界。

3.2.5 设计依据和局限性

Micro-RAG 刻意避免使用密集嵌入索引等重量级检索机制 s
or c交叉编码器。这一选择体现了以下几方面的考量：

- 规划决策是粗粒度的，相较于细粒度的语义相似度，它们更受益于符号线索；
- 规划器提示词较短，因此受益于稳定、低噪声的提示；
- 笔记本电脑级别的运行环境无法支持大型神经索引。

因此，应将 Micro-RAG 视为一种结构性偏置机制，而非战术级证明中前提选择的替代方案。它的优势在于引导规划器采用熟悉的证明模式，同时将所有正确性检查交由 Isabelle 完成。

3.3 空缺填充与反例引导的证明修复

本小节将以与逐步证明器相同的验证器在环算法风格，阐述规划器的填充+修复机制。其核心设计目标是将部分正确的 Isar 大纲视为一个结构化搜索对象，并在 Isabelle 验证下，将每个剩余的 sorry 转化为一系列有界综合问题。

3.3.1 对象与不变量

令 P表示当前的 Isar 脚本（一系列行）。空缺 是文本中对应于一个 sorry 标记出现位置的最大跨度

$$h = [a, b) \subseteq \{0, \ldots, |P|\}$$

。令 Holes(P) $= \{h_1, \ldots, h_\ell\}$。
规划器维护以下不变量：

1. **验证器门控：** 只有当完整脚本在 Isabelle 中验证通过时，才会提交更改。2. **局部进展容许机制：** 即使更改未能验证通过，仍可将其保留为部分进展，但必须立即通过打开新的显式空缺来规范化，以便后续迭代能够以这些空缺为目标。3. **稳定缺口标识：** 每个空缺都有一个稳定标识符，用于跟踪文本编辑前后的修复阶段和尝试次数。

稳定缺口标识符。 由于编辑后行索引会发生偏移，每个空缺都通过一个窗口化指纹来标识：

$$\text{hid}(P, h) = \text{SHA1}\big(P[\max(0, a - w) : \min(|P|, b + w)]\big)[:16],$$

其中 w 是固定的字符窗口。规划器使用此标识符来维护状态

$$\Pi = (\text{stage}[\text{hid}],\ \text{tries}[(\text{hid}, s)],\ \text{focus}),$$

其中 stage[hid] $\in \{0, 1, 2\}$ 表示该空缺当前所处的修复阶段，tries 表示每个阶段的尝试次数，而 focus 则可选地将遍历固定到部分进展后指定的某个空缺。

3.3.2 有效目标提取

对于脚本 P 中的每个空缺 h，规划器通过查询 Isabelle，获取紧邻其前的证明状态，从而计算出 有效目标 G_h h。

具体而言，令 $P_{\prec h}$ 为 h中 sorry 之前位置处结束的前缀。规划器会构造一个临时理论，执行 $P_{\prec h}$，并以 print state（或等效的状态打印命令）结束，从而让 Isabelle 输出一个文本证明状态块。令 State(P, h) 表示该代码块。有效目标通过以下文本提取函数计算：

$$G_h := \mathsf{EffGoal}\big(\mathsf{State}(P, h),\ G,\ P,\ h\big),$$

该函数优先选择 (i) 当前活动子目标陈述；若状态解析失败，则 (ii) 一致地回退到原始目标 G。

这样便将全局填充转化为局部子问题：

$$G \rightsquigarrow \{G_{h_1}, \ldots, G_{h_\ell}\}.$$

3.3.3 填充：调用 thestepwiseprover 作为局部合成器

给定一个空缺 h 及其有效目标 G_h，规划器首先尝试通过调用逐步证明器执行填充 步骤，并采用较小的预算和浅层搜索参数：

$$\mathsf{Solve}(G_h;\ \text{budget}, \text{depth}, \text{beam}, \text{facts}, \ldots) \ \Rightarrow\ \text{a sequence of commands } C.$$

返回的命令经过后处理，以提取：

- apply 步骤： $A = [c \in C \mid c$ 以 apply$]$, 开头
- 收尾命令： $f \in C$ 以 by 开头，或等于 done。

随后，规划器尝试将这些命令拼接到空缺区域中。区分两种情况：

情况 1：有可用的收尾命令。如果存在 f，则将空缺替换为一个缩进代码块，其中包含A，后接 f。令 Replace(P, h, Δ) 表示将跨度 h 替换为文本 Δ 后得到的脚本。填充候选项为

$$P' := \mathsf{Replace}(P, h, \mathrm{Indent}(A +\!\!+ [f])).$$

如果 Isabelle 验证通过 P'，则提交填充。

情况 2：仅应用式进展。如果只有 apply 步骤可用，规划器绝不会立即宣告成功。相反，它会将仅应用式输出视为一种变换：该变换可能减少子目标，但不会完成局部证明义务。此外，在 Isar 中，apply 步骤并非在所有位置都符合语法要求（尤其是在“prove”模式下的 have/show/obtain 标题下）。因此，规划器会执行结构放置检查：

1. 如果空缺位于允许使用 apply 的上下文中，则插入 A，然后通过 sorry 强制留下一个新的显式空缺；2. 否则，将 A 包装进一个小型子证明中（例如 proof - . . . sorry qed），以确保脚本格式正确。

In b在其他情况下，结果会被视为部分进展，并传递给下文的下一步归一化步骤。

3.3.4 部分进展规范化：开启最小化 sorry 占位符

一项核心工程设计选择是，规划器绝不会在缺少显式空缺的“半编辑”脚本上继续搜索。每当填充或修复尝试更改了脚本，却未能通过全局验证时，规划器就会调用归一化算子：

$$(P^\star,\ \text{opened}) := \mathsf{OpenMinimalSorries}(P).$$

直观而言，OpenMinimalSorries 会扫描失败的策略行（通常是 apply序列或 by命令），并以保留 Isar结构的方式将其替换为 sorry：

• 如果应用序列嵌套在 have/show 头部之下，它会被转换为一个局部证明——. ..sorry qed 块，而不是保留原始的 apply 行；
• 否则，失败的那一行会被替换为缩进匹配的 sorry 行。

如果打开了任何空缺，规划器会将焦点更新为旧位置附近最新创建的空缺中距离最近的一个（以确保工作连续），并从该空缺继续。

3.3.5 修复：在结构化代码块编辑上进行反例引导的归纳综合

如果填充失败（或因当前阶段而跳过），规划器会调用尝试 CEGIS 修复。此过程实现了一个有界的、采用 CEGIS 风格的循环，针对的是代码块级编辑 而非单个策略。

修复搜索状态。令 P 为当前脚本，h 为目标空缺。修复过程会维护：

- P_t：当前脚本候选项（初始为 P），
- $S_0 :=$ 状态$(P,\ h)$：初始状态块，
- 由修复预算 s 得出的时间预算函数 left()，
- 一个先前失败存储 $\mathcal{M}$，将块类型映射到 $\rightarrow$ 一个有界列表，其中包含先前尝试过的块候选项，用作大语言模型提示的禁用列表。

将目标重新定位到最早的失败锚点。尽管修复以空缺跨度为参数，但实际的语法错误通常发生在 sorry 之前。因此，修复首先计算一个锚定行：

$$\ell_{\text{anchor}} := \mathsf{EarliestFailureLine}(P, h),$$

并将修复焦点行设为 ℓ_{anchor} 附近经过限制的索引。这使修复从“恰好在空缺处编辑”转变为“编辑导致空缺变得不可证明的最早代码块”。

块类型与阶段。修复会考虑三种块类型，并与阶段升级相对应 ：

1. **阶段 1（have/show 微型块）：** 找到包围焦点行的 have/show/obtain 块，并尝试重新生成该微型块。2. **阶段 2a（case 块）：** 如果 cases/case 结构包围焦点行，则尝试重新生成 case 块。3. **阶段 2b（子证明）：** 如果 proof . . .qed 子证明包围焦点行，则尝试重新生成整个子证明。

完整证明再生成由外层驱动程序作为阶段 3 处理。

反例提示。提出新的代码块之前，修复会提取面向反例的提示：

$$\mathrm{CE} := \mathsf{CounterexampleHints}(\mathcal{I}, S_0),$$

其中可能包括 Quickcheck/Nitpick 风格诊断所建议的变量绑定和定义展开。这些提示在大语言模型提示词中被视为软约束。

大语言模型提议与严格去重。对于块区域 B（连续的行跨度 $[i, j)$），修复会构建一个提示词，其中包含：

- 有效目标 G_h，
- 当前脚本中观察到的规范化 Isabelle 错误消息，
- 代码块附近提取出的证明上下文，
- 当前代码块文本 B，
- 反例提示 CE，
- 之前的失败块列表来自 $\mathcal{M}$。

$\widehat{}$大语言模型输出一个候选替换块 B。修复会计算一个指纹

$$\mathrm{fp}(\widehat{B}) := \mathrm{SHA1}(\mathrm{normalize}(\widehat{B})),$$

$\widehat{}$ $\widehat{}$并且，如果 fp(B) 与 $\mathcal{M}$ 中为该块类型存储的任一指纹匹配，就会立即拒绝 B。这种"严格去重"可防止随机解码导致病态重复。

包装剥离与块规范化。 由于大语言模型经常包含多余的包装（例如，输出完整引理而非子块），修复会应用针对类型的剥离算子：

$$\widehat{B}' := \mathsf{StripToType}(\widehat{B}, \mathrm{type}),$$

以确保替换内容与原始代码块具有相同的句法粒度。

修复验证关卡。 通过替换来构造候选脚本

$$P' := \mathsf{ReplaceLines}(P, [i, j), \widehat{B}').$$

随后，修复会对 P'（作为完整理论）执行 Isabelle 检查。如果验证成功，则将 P' 作为已验证修复返回。如果验证失败，修复会将 B' 记录到 $\mathcal{M}$ 中（列表长度受最大列表长度限制），然后继续处理，前提是 left() 大于零。

如果候选内容更改了脚本但未通过验证，修复仍可能将其作为部分进展返回，并附带诊断标签（例如 stage=1 partial-progress ），以便外层循环应用 OpenMinimalSorries，并继续填充显式空缺。

3.3.6 外部控制：阶段上限、焦点管理与升级

驱动程序会针对每个起始阶段维护各空缺的尝试次数。令 tries[(hid s)] 表示在阶段 s 对空缺 hid 发生的验证关卡失败次数（包括无变化结果）。系统强制执行两个上限： c_1 和 c_2。

$$c_1 = 2, \qquad c_2 = 3.$$

升级过程如下：

$$\mathrm{stage}[\mathrm{hid}] \leftarrow \begin{cases} 2 & \text{if } \mathrm{stage} = 1 \text{ and } \mathrm{tries}[(\mathrm{hid}, 1)] \geq c_1, \\ 2 \text{ and trigger whole regeneration} & \text{if } \mathrm{stage} = 2 \text{ and } \mathrm{tries}[(\mathrm{hid}, 2)] \geq c_2. \end{cases}$$

焦点策略。 每当部分进展打开新的空缺时，驱动程序会围绕先前位置采用最近空缺启发式，并将焦点设置到该空缺的指纹上。这样可以形成连贯的局部搜索过程，而不是在互不相关的空缺之间来回跳转。

对 Isabelle 不稳定性的鲁棒性。 fill 和 repair 中的所有 Isabelle 调用都使用有界超时进行封装。异常（超时、由格式错误的中间理论引起的值错误，或意外失败）会触发受控 Isabelle 重启，并将相应尝试计为失败，但不会终止整体规划循环。

3.3.7 算法概要

填充+修复循环可概括为对脚本进行有界搜索的过程：

> **状态：** 当前脚本 P、空洞集合 Holes(P)、规划器状态 Π。**操作：** 通过逐步证明器执行填充空洞；通过大语言模型执行修复代码块；OPENMINI- MALSORRIES；重新生成整体。**转移：** 执行一个操作以获得 P'；若 VERIFY(P') 成功，则提交；否则通过打开空缺进行规范化并更新 Π。**目标：** 在全局时间预算内，找到一个脚本 P，使得 VERIFY(P) $=$为 true，且不含 sorry$\notin P$。

这种设计体现了逐步证明器的理念：大语言模型用于提出候选编辑，而 Isabelle 仍是唯一的正确性裁判。主要区别在于提议的粒度：逐步证明器提出单条命令；规划器修复则提出结构化的 Isar代码块，使证明空间在局部命令级搜索陷入停滞时能够实现更大的"跨越"。

4 数据处理与生成

逐步证明器和证明规划器都经过了相应的埋点设置，可生成丰富的执行轨迹，同时用于调试工件、基准测试结果以及基于学习的组件的训练数据。本节介绍 prover/ 和 planner/ 模块中实现的数据模型、日志记录流水线、数据集生成流程和基准测试基础设施。

4.1 设计原则

数据流水线遵循四项设计原则：

1. **基于验证器的数据**：所有记录的信号均源自具体的 Isabelle 执行，而非模型内部的置信度。2. **细粒度监督**：每个尝试执行的操作（策略、代码块编辑、大纲）都会被记录，而不仅仅是成功的证明。3. **阶段分离**：来自战术级证明、规划、填充和修复的数据彼此可区分，但共享一种通用模式。4. **可复现性**：日志采用仅追加方式，与版本无关，并且足以离线重放或重新评估决策。

因此，该系统能够自然地产生适用于监督学习、离线强化学习、消融研究和纵向基准测试的数据集。

4.2 运行级日志记录

每次调用证明器或规划器都会生成一条运行记录，对应一次目标尝试。设 G为一个目标陈述。运行记录 $R(G)$包含：

- 目标标识符和目标文本，
- 证明器/规划器模式及其配置（束宽、深度、温度、预算），
- 大语言模型、重排序器和前提选择器的模型标识符，
- 墙钟运行时间和超时状态，
- 成功标志，以及成功时的最终证明文本，
- 汇总统计量（达到的深度、扩展次数、修复次数、重新生成次数）。

运行记录按每行一个 JSON 对象（JSONL）的形式序列化，因此无需将完整日志加载到内存中，即可进行流式写入和事后聚合。

4.3 尝试级日志记录

更细粒度的监督信息会在尝试级别记录。一次尝试对应于由 Isabelle 评估的一项拟议操作。尝试的确切含义取决于上下文：

- 在逐步证明器中：将一条候选命令应用于一个证明状态；
- 在规划器填充阶段：将一个候选证明片段插入一个空缺；
- 在规划器修复阶段：替换结构化代码块或子证明。

每条尝试记录包括：

- 目标 G 以及当前脚本前缀或大纲，
- 拟议操作文本，
- 尝试类型（步骤、收尾命令、填充、修复、重新生成），
- Isabelle 验证下的成功标志，

- 尝试前后的子目标数量（如适用），
- Isabelle 的执行耗时，
- 缓存命中指示器，
- 当前搜索深度或修复阶段，
- 辅助特征（检索分数、重排序器分数、提示使用情况）。

形式化地，每次尝试都会生成一个带标签的元组

$$(x, y, \Delta),$$

其中，x 是该次尝试的特征表示，$y \in \{0, 1\}$ 是验证器结果，而 Δ记录状态变化指标，例如子目标减少量。

这些尝试日志是策略重排序器、前提编码器和修复启发式方法的主要训练数据来源。

4.4 持续学习的数据集生成

日志记录的尝试经过后处理，被整理成适用于不同学习任务的结构化数据集。

重排序器数据集。对于策略和代码块重排序，每次尝试都会生成一个监督示例（x，y），其中 x 是数值特征向量，y 表示验证器接受结果。根据训练目标，标签可以转换为：

- 二元分类目标，
- 近似于 Q值的回归目标，
- 用于离线策略学习的优势加权目标。

还可以重构对应单次证明尝试的回合，从而得到轨迹

$$(s_0, a_0, r_0, s_1), \ldots, (s_T, a_T, r_T, s_{T+1}),$$

其中，奖励 r_t 源自子目标减少量或证明完成。

前提选择数据集。对于前提选择，成功的尝试提供了弱监督：在成功步骤中明确引用的任何引理，都会被视为相应目标或状态的正前提。负前提从检索池中采样，但不会用于证明。由此可得到对比样本对（g，p^+）和（g，p^-），用于训练双编码器和交叉编码器。

规划器修复数据集。规划器日志还编码了：

- 块类型（have/show、分情况、子证明），
- 空缺的有效目标，
- 反例诊断，
- 失败历史和禁用候选项。

The这些信号为未来研究学习块级修复策略或重新生成策略提供了支持 .

通过将证明搜索视为数据生成过程，而非黑盒求解器，该系统能够通过学习持续改进，同时保持形式化验证的可靠性保证。

5 与 Isabelle/jEdit UI 集成

为支持交互式使用，系统与 Isabelle/jEdit 用户界面进行了轻量级集成。用户可以在编辑器中当前引理所在的位置直接调用逐步证明器和证明规划器，并将生成的证明文本插入缓冲区。该设计刻意保持用户界面层精简：所有推理和验证都由现有的证明器和规划器组件完成，用户界面则纯粹作为便捷的操作界面。

5.1 整体架构

该集成由两部分组成：本地 HTTP 服务器和一组 jEdit 宏。HTTP 服务器作为长期运行的进程运行，并提供一个封装证明器和规划器入口的小型 API。它还会维护一个持久的 Isabelle 服务器会话，因此多次调用用户界面时可以复用同一个 HOL 会话，避免重复启动带来的开销。对于用户界面而言，该服务器是所有证明自动化请求的唯一联系点。

在编辑器端，Isabelle/jEdit 通过 BeanShell 宏进行了扩展。这些宏会提取插入符附近引理的目标，将其发送到本地服务器，并将返回的证明文本粘贴回编辑器。通信仅在本地进行，并通过 HTTP 传输 simpleJSON 载荷。

5.2 服务器端功能

UI 服务器使用一个小型 FastAPI 应用程序实现。启动时，它会初始化一个持久运行的 I sabelle 服务器，并打开一个供后续所有请求复用的 HOL 会话。该服务器提供两个主要端点。

第一个端点提供对逐步证明器的访问。它接受一个目标字符串以及基本搜索参数，例如时间预算和模型选择，调用证明器，并返回生成的证明命令序列。返回结果之前，服务器会筛选输出，只保留可直接用于 Isabelle 证明的行，这些行通常以 apply 或 by 开头。这样可以确保粘贴的输出在语法上立即有效。

第二个端点提供对证明规划器的访问。根据请求模式，它会返回证明大纲，或尝试完整的规划并填充运行。该端点会转发特定于规划器的参数，例如大纲多样性、修复选项和 micro-RAG 配置；如果省略参数，则使用服务器端默认值。响应包含规划器生成的当前最佳证明脚本，无论所有空缺是否都已填充。

该服务器旨在抵御 Isabelle 不稳定性。必要时，服务器会通过重启 Isabelle 会话来处理超时和意外故障，而不会导致服务器进程本身崩溃。

5.3 编辑器宏

jEdit 宏实现了统一的编辑器端工作流程。调用宏时，它会从插入符处向上扫描，定位最近的引理或定理声明，并提取带引号的目标字符串。随后，它会判断插入符位于相应证明块的内部还是外部。此判断决定生成的文本将插入何处：插入符所在位置，或紧接在引理头部之后。

提供了三个宏。第一个宏调用逐步证明器，并插入建议的证明命令。第二个宏以仅生成大纲的模式调用规划器，并插入结构化的证明骨架。第三个宏以完整的规划并填充模式调用规划器，并插入生成的脚本；该脚本可能已经完整，也可能仍包含明确标出的空缺。

插入之前，规划器的输出会在编辑器端进行轻度规范化。具体来说，会去除多余的引理头部，并且只插入 proof–qed 块。缩进会调整为与编辑器缓冲区中的上下文相匹配。

5.4 预期用途

此用户界面集成旨在作为交互式辅助工具，而非批量评估的替代方案。典型的工作流程是在 Isabelle/jEdit 中编写或导入引理陈述，将插入符放在引理内，然后调用其中一个宏。随后，证明器或规划器会尝试构造证明，并将生成的文本直接插入编辑器，供用户检查、编辑或重新运行。

通过保持用户界面层精简，并将所有实质性推理交由底层系统处理，该集成在不损害可靠性、也不重复证明器和规划器中已有逻辑的前提下，提供了流畅的交互体验。

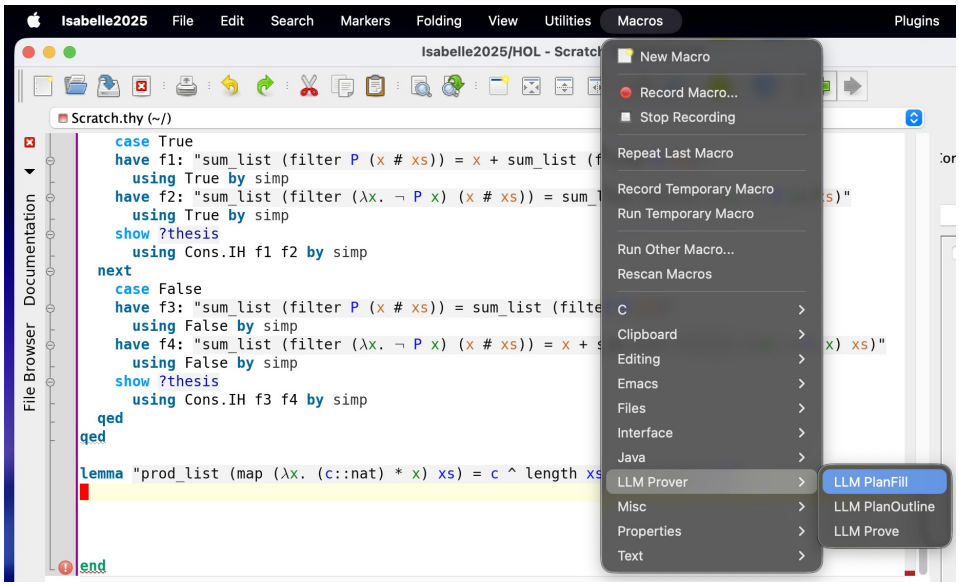

图 1：与 Isabelle/jEdit 集成的示意图。

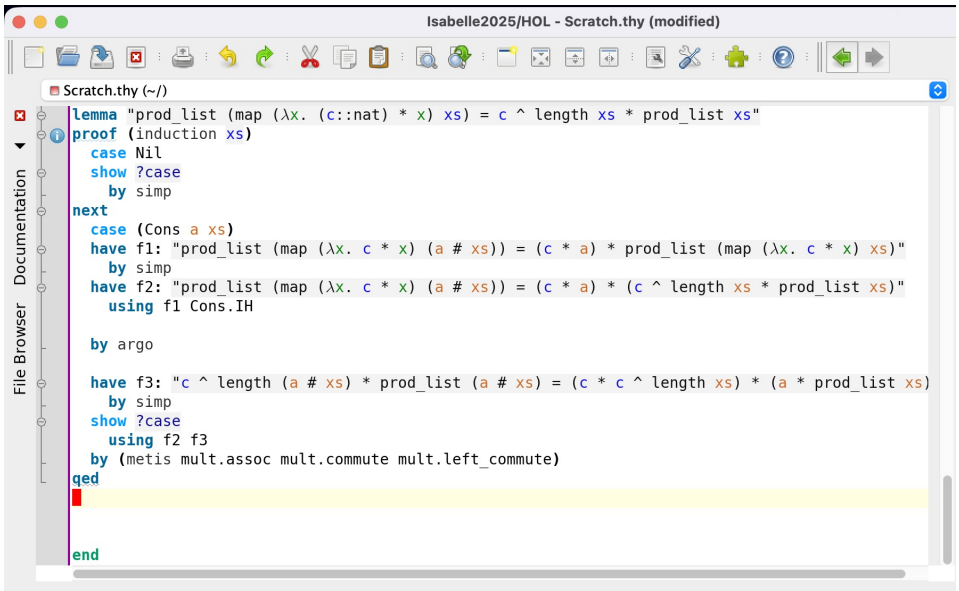

图 2：大语言模型证明器插入的证明示例。

例如，假设用户已按上述说明启动本地服务器，便可将光标移至某个证明状态，然后从宏菜单中点击 LLM → PlanFill，如图 1 所示。计算完成后，如果大语言模型证明器能找到有效证明，就会将其插入文档中的正确位置，如图 2 所示。

6 结论

本研究介绍了一种面向 Isabelle/HOL 的大语言模型驱动定理证明系统，该系统将验证器引导的逐步搜索与更高层次的证明规划器相结合，后者基于 Isar 大纲、空缺填充和修复。该系统旨在完全自动运行：给定一个目标，它会尝试合成完整的、经内核验证的证明，无需用户交互。在整个设计中，Isabelle

始终是正确性的唯一权威，而大语言模型仅用作提案机制，其输出会在严格的语法和语义约束下经过筛选、验证和修复。

本项目的一项关键贡献在于证明：在特定情形下，大语言模型驱动的证明搜索已能超越经典自动化方法。具体而言，即使给予 Sledgehammer 充足的超时时间，逐步证明器仍能解决某些 Sledgehammer 无法企及的高阶目标。图 2 展示了一个典型示例：某个引理无法由 Sledgehammer 证明，却能由大语言模型证明器通过一系列基于语义理解的证明步骤加以证明。这表明，基于 LLM 的指导并非只是现有自动化方法之上的便利层，还能探索与传统启发式方法所编码的策略有本质区别的证明策略。

与此同时，本项目也揭示了明显且根本的局限性。最值得注意的是，证明规划器的填充与修复组件远不如最初预期的有效。尽管我们精心设计了有效目标提取、分阶段修复、CEGIS 式循环和完整证明再生成，但即使是 GPT 5.2 Extended Thinking 和 Gemini 3 Pro 等最先进的大语言模型，也难以可靠地实现第 3.3 节所述的填充与修复特征设计。实际使用中，这些模型往往无法生成算法设计如此复杂的代码；尤其当代码库超过 20 个文件，且新代码需要遵循现有 API 时，更是如此。因此，规划器的修复阶段很少能在简单情形之外取得成功，而大多数成功的证明要么直接来自初始大纲，要么由未获规划器协助的逐步证明器完成。

这些局限性并非只是实现层面的瑕疵，而是反映了更深层的挑战。修复 Isar 证明需要同时推理全局结构、局部证明状态以及 Isabelle 对上下文敏感的证明模式，而且必须遵循严格类型化、对缩进敏感的语法。当前的大语言模型即使拥有较长的上下文和先进的推理能力，似乎仍缺乏足够精确的内部模型来理解这些约束，因此难以大规模可靠地完成修复。这表明，要改进填充与修复，可能需要新形式的训练数据、更严谨的符号抽象，或让规划器与逐步证明器实现更深层的集成，而不是单纯依赖更强大的语言模型。

尽管存在这些不足，当前实现已经是 Isabelle/HOL 用户的实用工具。仅逐步证明器就能证明一些现有自动化工具无法解决的目标，而规划器则为今后开展结构化证明综合研究奠定了基础。更广泛地说，该项目表明，将定理证明视为一种生成数据、验证器在环的过程，便能开展系统化实验并逐步改进，即使部分组件尚未达到最终目标。我们认为，这项工作是迈向易用且完全自动化的表达力强的逻辑定理证明的一步，也是一个具体案例研究，展现了基于大语言模型的推理在形式化验证中的潜力与当前局限。

参考文献

[1] Jasmin Christian Blanchette 和 Tobias Nipkow。Nitpick：一种基于关系模型查找器的高阶逻辑反例生成器。载于交互式定理证明国际会议（*ITP*），计算机科学讲义。Springer，2010。数字对象标识符：10.1007/978-3-642-14052-5 11。[2] Jasmin Christian Blanchette、Sascha B¨ohme 和 Lawrence C. Paulson。使用可满足性模理论求解器扩展 sledgehammer。载于国际自动演绎会议（*CADE*），计算机科学讲义。Springer，2011。数字对象标识符：10.1007/978-3-642-22438-6 11。[3] Edward First、Michael Kleiner 等。Baldur：使用大语言模型生成和修复完整证明。载于 *ACM* 欧洲软件工程会议与软件工程基础研讨会联合会议（*ESEC/FSE*），2023。[4] Albert Q. Jiang、Wenda Li、Jiawei Han 和 Yuhuai Wu。LISA：Isabelle 证明语言模型。载于人工智能与定理证明会议（*AITP*），2021。[5] Albert Q. Jiang、Wenda Li、Sebastian Tworkowski、Konrad Czechowski、Tomasz Odrzyg´o´zd´z、Piotr Miło´s、Yuhuai Wu 和 Mateja Jamnik。Thor：运用锤子集成语言

模型和自动定理证明器。载于神经信息处理系统进展（*NeurIPS*），2022。[6] Guillaume Lample、Marie-Anne Lachaux、Thibaut Lavril、Xavier Martinet、A. Hayat、Gabriel Ebner、Pierre Rodriguez 和 Timoth´ee Lacroix。面向神经定理证明的超树证明搜索。*arXiv* 预印本 *arXiv:2205.11491*，2022。[7] Tobias Nipkow、Lawrence C. Paulson 和 Markus Wenzel。*Isabelle/HOL*：高阶逻辑证明助手，计算机科学讲义第 2283 卷。Springer，2002。国际标准书号 978-3-540-43376-3。数字对象标识符：10.1007/3-540-45949-9。[8] Han Wang、Haotian Xin、Zhenwen Liu、Wenya Li、Yufan Huang、Jing Lu、Ziyi Yang、Jie Tang、Jian Yin、Zhixun Li 和 Xiaodan Liang。递归地证明定理。*arXiv* 预印本 *arXiv:2405.14414*，2024。[9] Kaiyu Yang、Alexander M. Swope、Albert Gu、Rohan Chalamala、Peng Song、Shuo Yu、Sachin Godil、Ryan Prenger 和 Anima Anandkumar。LeanDojo：使用检索增强语言模型进行定理证明。载于神经信息处理系统进展（*NeurIPS*），2023。[10] Longhui Yu、Weisen Jiang、Han Shi、Jincheng Yu、Zhengying Liu、Yu Zhang、James T Kwok、Zhenguo Li、Adrian Weller 和 Weiyang Liu。MetaMath：为大语言模型构建自己的数学问题。*arXiv* 预印本 *arXiv:2309.12284*，2023。[11] Kun Zheng、Jiawei Han 和 Stanislas Polu。MiniF2F：形式化奥赛级数学的跨系统基准测试。*arXiv* 预印本 *arXiv:2109.00110*，2021。