arXiv:2601.03768v1 [cs.PL] 7 Jan 2026

智能体证明自动化：案例研究

Yichen Xu[0000−0003−2089−6767] 和 Martin Odersky[0009−0005−3923−8993]

洛桑联邦理工学院，瑞士洛桑
yichen.xu@epfl.ch martin.odersky@epfl.ch

摘要。 证明工程以劳动密集著称：纸面上看似简单明了的证明，在定理证明器中往往需要编写冗长的脚本。大语言模型（LLM）的最新进展为证明自动化带来了新的机遇：现代大语言模型不仅能生成证明脚本，还具备智能体行为，能够探索代码库，并根据定理证明器的反馈迭代改进输出。这些进展催生了一种新兴方案：由基于大语言模型的智能体在人类指导下承担大部分证明工程工作。人类提供数学洞见（定义、定理、证明策略），智能体则处理证明开发中的机械性工作。我们将这一方案称为智能体证明自动化。

我们通过一个案例研究来介绍这一方案：在 Lean 4 中对复杂形式系统 System Capless 的语义类型可靠性进行形式化，该系统包含超过 14,000 行代码。借助现成的大语言模型智能体和一个轻量级证明检查工具，智能体完成了 189 项证明工程任务，成功率达到 87%，其中仅有 16% 的任务需要人工干预。该案例研究表明，智能体能够胜任证明工程师的工作，并显著提高生产力；不过，它们在创造性推理方面仍有所欠缺，在某些情况下依然需要人工指导。我们发布了一个交互式浏览器，读者可以查看所有智能体交互过程；该形式化成果也已开源，供实验和扩展使用。

关键词： 证明自动化 · 交互式定理证明 · 大语言模型。

1 引言

机器检查证明，即机械化证明，为安全关键软件提供高度可靠的保证。通过 Coq [28], 、Lean [20], 或 Isabelle/HOL [21] 等交互式定理证明器（ITP）验证的系统，其正确性属性由机器进行检查。这些保证比测试更加可靠，因为测试可能遗漏边界情况；它们也比传统的纸笔证明更加可靠，后者容易受到人为错误的影响。

然而，证明形式化一直以难度极高而闻名。人类在纸上勾勒或在脑海中构思的非形式化数学论证，与定理证明器所要求的机器检查证明之间存在巨大差距。例如，CompCert [17], 这一经验证的 C 编译器，耗费了六个人年，才通过约 100,000 行 Coq 代码完成形式化。当人类写下

“这可以通过直接归纳法得出”时，在证明助手中，要为这种直接性提供论证，往往需要数百行代码。这种转换不仅要求对问题领域和交互式定理证明器都有深入的专业知识，还需要投入大量时间。对于复杂系统，证明开发可能耗费专家数月乃至数年的精力。因此，对于大多数现实世界的软件而言，形式化验证的成本高得令人难以承受。

提出证明自动化旨在缩小这一差距，研究人员也开发了各种技术 [4,5,9,19,13]。`eauto` 和 `try` 等策略能够自动处理常规证明义务。Stainless [15] 和 Liquid Haskell [33] 等高度自动化的工具集成了 SMT 求解器，以处理验证任务。这些进展提高了生产力，但仍然需要大量人工投入。即使有了复杂的自动化工具，证明工程师有时仍需编写冗长的脚本，手动管理证明状态。此外，现有自动化工具的可恢复性和可解释性有限：当证明器或策略失败时，用户几乎无法了解原因，也难以引导工具成功完成证明。证明工程的负担依然存在。

大语言模型（LLM） [32,12,8,23,31,11] 的最新进展为应对这一挑战带来了新的机遇。现代大语言模型展现出代码生成能力，其中也包括证明脚本。这些模型经过海量语料训练，语料中既包含自然语言，也包含形式语言。更重要的是，这些模型如今支持智能体行为：它们可以自主探索代码库、生成并测试代码、从失败的尝试中回溯，并根据反馈迭代优化输出。对于证明开发而言，这意味着大语言模型可以在反馈循环中与定理证明器交互：检查证明目标、生成证明脚本、接收证明器可能返回的错误消息，并据此改进尝试。这种交互能力使大语言模型从单纯的代码生成器转变为积极参与证明工作的证明工程师。与纯粹在形式层面运行的传统证明自动化不同，大语言模型能够衔接自然语言和形式语言，有望充当人类数学直觉与机器检查证明之间的翻译者。

我们在基于大语言模型的智能体方面的经验表明，它们在证明工程中展现出令人瞩目的能力。智能体能够搜索相关模块，找到相关引理，生成证明脚本以填补证明空洞，并递归地改进尝试，直至成功。如果无法成功，它们会指出错误的假设；如果目标过于复杂，无法直接证明，则会建议进行引理分解。这些能力促成了一种新的证明开发模式：在人工指导下，大语言模型智能体承担大部分证明工程工作。在这种模式中，人类专注于关键的数学洞见：撰写定理陈述、进行引理分解，并以自然语言描述证明策略。随后，大语言模型智能体生成证明脚本，填补证明中的空洞，处理定理证明器的大部分技术细节；如果发现人类推理中存在任何缺陷，也会予以指出。我们将这种新兴模式称为智能体证明自动化。

本文旨在通过一个案例研究展示这一方案：在 Lean 4 中对一个复杂形式系统（SystemCapless [37]）的可靠性进行形式化。

该形式化包含超过 14,000 行代码。人类定义了形式系统并编写了定义，而智能体则在高层指导下负责证明开发。我们收集了人类与智能体交互的大量记录，并手动将其标注为任务。每项任务代表一个最小的人机交互单元：人类表达意图，智能体尝试满足这一意图。我们根据任务性质对其分类（例如，完成证明、修复有问题的证明、重构代码），并根据结果进行评估（成功、部分成功、失败）。我们展示了人机交互的具体示例，对智能体的贡献进行了定量分析，包括成功率和性能模式，并分享了关于如何有效利用这一新方案进行证明开发的见解。本案例研究附带一个探索工具，读者可以借此交互式地浏览原始交互记录和标注后的任务，了解人类如何提供指导，以及智能体如何在反馈循环中构建证明。该探索工具既可从开源仓库[1] 在本地部署，也可在线访问[2]。本文将多次引用该探索工具中的示例。本案例研究的形式化成果也已开源[3]。

智能体证明自动化降低了机器验证的证明开发门槛。人类可以通过自然语言指导来探索系统的形式化属性，并将底层证明细节交由智能体处理。这一转变有望让形式化方法不再局限于小众应用：随着验证对无需多年定理证明器专业经验的实践者变得触手可及，数学严谨的软件开发也将能应用于目前仅依赖测试的领域。这些潜在影响促使我们开展本案例研究，探索智能体证明自动化的实际应用，并展示其在现实世界证明开发中的可行性。

2 AgenticProof 自动化示例

本节将展示智能体证明自动化的实际运用。我们将介绍相关设置，呈现案例研究中的具体示例，分析由此显现的模式，并讨论该工作流程的特点。

2.1 设置

我们的案例研究采用配备标准软件工程工具的现成大语言模型智能体。我们没有对模型进行微调，也没有构建定制的证明生成系统。相反，我们使用通用编程智能体，并添加了一款用于集成 Lean 4 的轻量级工具。这种简单的设置有两点意义。首先，任何从业者都可以在无需专用基础设施的情况下采用这一工作流程。这项形式化工作已开源，其他人可以据此试用该工作流程，或将其作为自己项目的起点。其次，

[1] https://github.com/linyxus/apa-explorer [2] https://apa.univalence.xyz/ [3] https://github.com/Linyxus/semantic/tree/case-study

该工作流程具有通用性：它可以迁移到其他证明助手，并随着前沿模型的改进而扩展。

模型与框架。 在整个项目中，我们使用了多种前沿大语言模型：Claude Sonnet 4.5、Claude Opus 4.5、Claude Opus 4.1 和 GPT Codex 5.1。这些模型运行于两种智能体框架中：Claude Code（用于 Claude 模型）和 Codex-CLI（用于 GPT 模型）。这两种框架提供了相似的能力：它们为底层大语言模型配备了文件系统访问、命令执行以及根据工具输出进行迭代改进的工具。

智能体工具。 智能体可以使用四类工具：

– **代码探索**：按模式搜索文件、查找定义和用法，以及读取文件内容。– **文件修改**：创建和编辑文件，并能够对特定位置进行有针对性的修改。– ***Lean 4* 构建**：一个名为 `lean4check` 的自定义工具，可编译 Lean 4 模块并呈现结果。该工具会报告成功，或整理错误消息并附上相关源代码上下文，以便于理解。

`lean4check` 工具是唯一的 Lean 专用扩展。它会在命令行上调用 Lean 编译器，收集其输出，并将错误与其源代码上下文一并呈现，以便智能体更轻松地解读。该工具无需与证明助手的内部组件集成。例如，当证明脚本出现类型不匹配错误时，`lean4check` 会生成：

```
ERROR: Semantic/CC/Eval/SmallStep.lean:35:45
33 |   (hv : Exp.IsVal v) ->
34 |   h l = none ->
35 |   Step h (.letin v e) (h.extend l v) e
   |                                      ^
   | Application type mismatch: argument
   |   e
   | has type
   |   Exp ({},x)
   | but is expected to have type
   |   Exp {}
```

输出同时显示了错误消息和周围的源代码，为智能体理解问题提供了更充分的上下文。

工作流程。典型的交互过程如下：人类编写定义、陈述定理，并可选择在注释中提供非正式的证明策略。随后，智能体探索代码库以构建上下文：搜索相关定义、阅读相关文件，并找出可能有用的引理。准备就绪后，智能体通过修改文件生成证明脚本，并调用 `lean4check`。如果证明失败，智能体会读取错误消息、加以解读，并改进尝试。这个循环会持续进行，直到证明成功

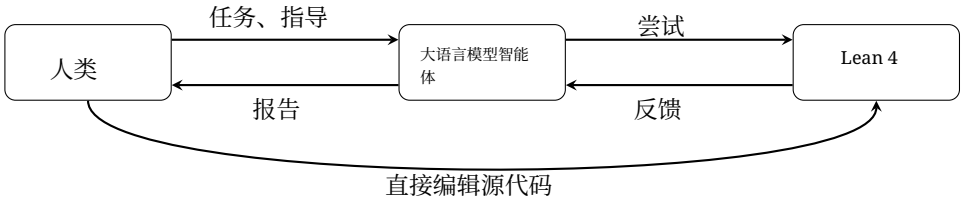


图 1. 人类—智能体—证明器工作流。人类提供任务和指导。智能体生成证明脚本，并根据 Lean 的反馈对其进行改进，再向人类报告成功或发现的障碍。

或者智能体判断目标需要人工干预：例如，缺少必要的引理，或证明策略存在缺陷。图 1 展示了这一交互过程。

2.2 SystemCapless 简介

我们的案例研究聚焦于对 System Capless（一种用于基于捕获的效应跟踪的演算）的类型健全性进行机械化证明。我们在此简要介绍该系统。

System Capless [37] 是一种基础演算，用于捕获跟踪 [7], ，这是一种通过记录某个值可能捕获哪些能力来跟踪效应和资源的类型系统化方法。

捕获跟踪的核心概念是捕获类型，它通过捕获集扩展普通类型。捕获类型 $S \wedge C$ 由一个形状类型S 和一个捕获集 C 组成，后者对该值可能引用的能力作出过近似。例如，执行控制台输入输出的函数可能具有类型 (字符串 $\to$ 单元类型) $\wedge$ {控制台}，表示它捕获了控制台能力；换句话说，调用时可能执行控制台操作。

该形式化工作采用逻辑关系方法，建立了 System Capless 的语义类型可靠性 [30] 。类型被解释为关于内存状态和表达式的谓词：如果一个值在所有可能的堆扩展下都按照该类型的规范运行，则该值满足此类型。核心结果是基本定理，它表明句法类型推导蕴含语义类型判定。结合类型指称的适切性，这便确立了

– 类型正确的程序在求值过程中不会陷入停滞，这就是经典的类型安全性；而–执行期间访问的能力不会超出类型系统所预测的捕获集范围。

形式化的复杂性源于两个方面。首先，System Capless 本身就是一种表达力强且复杂精妙的演算（完整定义见附录 A）。它支持对捕获集进行全称量化和存在量化。它还具有依赖函数类型，其中函数的结果类型可以提及其参数。例如，一个根据文件能力创建日志函数的方法

的类型为 $\forall(f:$ 文件$).($字符串 $\rightarrow$ 单元类型$)\ \hat{}\ \{f\}$：返回的日志记录器恰好捕获传入的文件 f。这种依赖关系对捕获跟踪至关重要，因为函数结果所捕获的能力往往取决于传入的参数。其次，语义类型健全性方法要求深入推理类型系统、运行时语义和类型指称（即将类型解释为谓词的逻辑关系）。这种相互交织的推理使证明变得错综复杂。

该形式化工作包含超过 14,000 行 Lean 4 代码，围绕德布鲁因索引语法组织，其中有三类变量（项变量、类型变量和捕获变量）、带有显式堆模型的大步操作语义，以及将类型解释为由能力集合和内存状态参数化的逻辑关系的指称语义。

2.3 示例

现在，我们将介绍本案例研究中出现的几类不同示例，每个示例都展现了人机协作的不同方面。

证明脚本开发 最常见的交互模式是证明脚本开发：人类陈述一个定理，智能体则给出证明。我们用 System Capless 形式化工作中的一个示例来说明。该形式化工作定义了两种操作语义。`Eval C m e Q` 是大步求值：表达式 `e` 在内存 `m` 中最终产生一个满足后置条件 `Q`的结果，并使用能力 `C`。`Step C m1 e1 m2 e2` 是小步归约：表达式 `e1` 执行一个原子步进，变为 `e2`，并可能将内存从 `m1` 更新为 `m2`。读者无需理解这些定义的全部细节，也能看懂这个示例。有兴趣的读者可参阅随附的资料，其中包含完整文档化的定义。

定理 `step_preserves_eval` 联系了这两种语义：如果 `e1` 求值为 `Q`，且 `e1` 步进到 `e2`，那么 `e2` 也求值为 `Q`。人类给出了这个定理，并留下了空证明：

```
theorem step_preserves_eval
    (heval : Eval C m1 e1 Q)
    (hstep : Step C m1 e1 m2 e2) :
    Eval C m2 e2 Q := by sorry
```

提示很简单：“证明定理 `step_preserves_eval`。”

智能体首先读取文件并检查相关定义。随后，它确定了证明结构：“让我先对求值推导进行归纳法证明。”第一次编辑用证明框架替换了 `sorry`：

```
theorem step_preserves_eval ... := by
  induction heval generalizing m2 e2 with
  | eval_val hv hQ =>
    -- Values cannot step: contradiction
    exact absurd hstep (step_val_absurd hv)
```

```
  | eval_var hQ =>
    exact absurd hstep step_var_absurd
  | eval_apply => sorry
  | eval_invoke => sorry
  ...
```

智能体正确识别出，基础情形（`eval_val`、 `eval_var`）会导致矛盾：值和变量都是无法继续步进的答案。它通过探索代码库，发现了引理 `step_val_absurd` 和 `step_var_absurd`，这两个引理指出，值和变量不可能进行归约步进。

接下来是一个编辑—检查—细化循环，其中包括对证明脚本进行 23 次编辑，以及调用 Lean 编译器 18 次。前 17 次检查均返回错误，每个错误都为下一步细化提供了指引。来看这个 `eval_letin` 案例，它需要进行嵌套的分类讨论。当 let 表达式 `let x = e1 in e2`步进时，会出现三种情况：绑定表达式 `e1` 步进（应用归纳假设）、 `e1` 是一个变量（重命名并继续），或 `e1` 是一个值（将该值分配到内存中）。智能体逐一处理了这些情况：

```
| eval_letin hpred heval_e1 h_nonstuck h_val h_var ih =>
  cases hstep with
  | step_ctx_letin hstep_e1 =>
    have heval' := ih hstep_e1
    have hsub := step_memory_monotonic hstep_e1
    apply Eval.eval_letin hpred heval' h_nonstuck ...
  | step_rename => ...
  | step_lift => ...
```

在第 18 次调用编译器时，证明编译通过，总计 240 行。从提示到完成证明，整个交互只进行了一轮，期间没有人类干预。智能体找到了相关引理（`step_memory_monotonic`、`eval_ans_holds_post`），确定了归纳原理，并使用恰当的策略处理了所有情况。

重构后的证明修复 定义发生变化时，下游证明往往会失效。智能体可以系统地修复这些证明，有时还能在此过程中发现更深层的问题。在这个例子中，人类为演算扩展了可变引用，在语法和语义中加入了 `read` 和 `write` 操作。这导致代码库中的许多证明失效。人类请智能体修复 `Semantics/Props.lean`:

> “修复此次可变引用重构导致的 Props.lean 中的所有错误。”

智能体系统地处理了整个文件，进行了 27 次编辑，并在 14 分钟内调用 Lean 编译器 18 次。它修复了十个不同的问题，主要是为新增操作添加分支，并更新证明以匹配变更后的定义。

然而，智能体遇到了无法解决的情况。有一个定理指出，当我们隐藏未使用的能力时，归约仍然成立：

```
theorem step_masked (hstep : Step C m1 e1 m2 e2) :
    Step C (m1.masked_caps C) e1 (m2.masked_caps C) e2
```

通俗地说：如果一个步进只使用 C 中的能力，那么隐藏 C 之外的所有内容后，它仍应有效。在为 read 和 write 添加情况后，智能体发现这个证明无法完成：

> “要让 step_masked 与 step_read/write 一起工作，可变单元必须属于能力集合 C。但当前的步进规则并未要求这一点！”

原始语义允许读取任意可变单元，但屏蔽操作会隐藏 C 之外的单元。智能体用 sorry 标记了这些情况，并报告了该问题。人类确认了这一诊断，并指示智能体修复语义，在 read 和 write 中加入缺失的成员资格要求。

这个例子表明，证明修复并非纯粹机械的过程。智能体发现，某些证明并非只是出了问题，而是在当前定义下无法证明，并指出了所需的概念性修正。

回答有关形式化的问题智能体还可以帮助浏览和理解代码库。以下是一个具体示例。人类提出了一个关于形式化的问题：

> “是否存在一个定理，指出只要 Eval C m1 e1Q，就存在 m2 和 e2，其中 e2 是一个答案，使得 Q m2 e2？”

人类想知道，是否已经对这样一个充分性定理进行了形式化：该定理指出，求值总会终止，并得到一个满足后置条件的答案。

智能体通过 10 次工具调用（6 次 grep、3 次读取文件、1 次 glob）搜索了代码库，耗时 52 秒，随后报告：

> “根据我对 CC 模块的搜索，**代码库中目前不存在这样的定理**。我找到的最接近的相关定理有：eval_ans_holds_post，它指出，如果 e 已经是一个答案且存在求值，那么后置条件就对它成立。不过，该定理要求 e 已经是一个答案。 [...] 你询问的定理应该是：
>
> ```
> th
> ```
>
> 在这样的定理。”

智能体回答了问题，找出了相关定理，并表述了人类心中的命题。

陈述并证明独立性质接着上一个示例，人类要求智能体陈述并证明缺失的定理：

> “在 Semantics/Props.lean 中陈述并证明这个定理。”

智能体将陈述形式化（在先前回复版本的基础上进行完善，加入了内存包含），并给出了证明：

```
theorem eval_exists_answer (heval : Eval C m e Q) :
```

证明通过对求值推导进行归纳来完成。基础情形（值、变量、能力调用）显而易见。顺序情形（`letin`,`unpack`, `cond`）应用归纳假设得到一个中间答案，然后利用续延得出最终答案。智能体在不到 5 分钟内完成了这项任务，共进行了 21 次工具调用（6 次编辑、5 次编译器调用）。

这个例子展示了智能体如何将非形式化猜想形式化。人类对应该成立的结论有一种直觉。智能体将这种直觉精确化，并通过形式化证明予以验证。

其他示例 我们观察到其他类型的交互，但出现频率较低。一个例子是理想化[4]，：人类在思考形式系统或证明策略的设计时，会把智能体当作思考伙伴，一起集思广益。在我们的案例研究中，我们发现，尽管智能体能很好地跟进对话，却无法提出新颖见解，而且总是盲目认可人类的想法。这体现了智能体在创造性推理任务上的当前局限：它是一位出色的证明工程师，却不是原创理论家。另一个例子是日常杂务与维护：人类会把繁琐的例行任务委托给智能体，例如添加文档注释[5]、、重构模块结构[6], 以及修复样式一致性问题[7]。智能体很好地完成了这些任务：对于那些与形式化的数学内容无关的日常维护工作，它是一个可靠的副驾驶。

2.4 模式与局限

通过案例研究中的这些互动，我们观察到人类引导智能体的反复出现的模式，以及智能体在哪些方面能够成功或遇到困难。这些互动揭示了一种自然分工：人类提供结构、上下文和数学洞见，而智能体则负责证明工程中的机械性工作。我们将讨论其中形成的引导模式以及遇到的局限。

人类引导模式 人工指导分为两个层面：系统级和任务级。

系统级指令会持久保存在配置文件中（通常命名为 `CLAUDE.md` 或 `AGENTS.md`）。这些文件编码了智能体

[4] https://apa.univalence.xyz/#/tasks/fe72733 2 5 3 34 0 820 28 8 64 f- fb - a d- f- cd eaa d https://apa.univalence.xyz/#/tasks/8efffdc3-91f0-45dd-b523-fe1ef496c79c [6] https://apa.univalence.xyz/#/tasks/53af846a-0e1e-49e4-8499-0af57bf266db [7] https://apa.univalence.xyz/#/tasks/1e4e9d0a-c3e3-4e25-b0b8-e0b323fc0982

应始终应用的领域知识：工具偏好（“使用 `lean4check` 而非 `lakebuild`”）、策略惯用法（“对等式使用 `calc` 模式”）、调试技巧（“使用 `trace_state` 检查目标上下文”），以及对语义不明显的策略的说明（例如，记录 `rename_i` 从目标上下文的底部而非顶部命名变量）。

任务级提示用于指导单次交互。
这些提示可分为几类：

– 指令式提示直接陈述目标：“证明定理 `step_preserves_eval`”或“修复 `Semantics/Props.lean` 中的所有错误。”– 策略性提示建议证明方法：“对求值推导使用归纳法”或“应用单调性引理。”– 渐进式提示将工作分解为多个阶段：“首先陈述定理并概述各个情形，留出 `sorry` 占位符”，或“我们一步一步来：先探索相关定义。”– 引导式提示帮助智能体熟悉代码库：“阅读模块文档以了解上下文”，或“尝试证明之前先探索相关文件。”– 自主式提示委派较长时间的工作：“继续修复错误，直到模块编译通过”，或“我暂时离开时，请处理剩余的情形。”

这些提示类型的具体示例见第 2.3 节的示例。

智能体局限性在我们的案例研究中，智能体表现出一些反复出现的局限性模式。下面逐一讨论。

微妙的证明状态处理。 智能体难以掌握证明器特定的惯用法，因为这些惯用法的语义并不显而易见。一个反复出现的例子[8] 是 Lean 4 中的 `rename_i` tactic，它会为目标上下文中的匿名假设（标记为 `†`）命名。该 tactic 从上下文的底部 开始为变量命名，这有悖直觉： `rename_i x y z` 会为最后三个匿名变量命名，而非前三个。尽管经过多次纠正并收到系统级指令，智能体仍反复误用这一 tactic。

上下文检索失败。 智能体有时无法找到代码库中已有的相关引理。如果引理名称与目标没有明显关联，或者引理位于意想不到的模块中，就会发生这种情况。于是，智能体会尝试重新证明已有的内容，或者干脆陷入困境。指出具体引理名称的策略性提示可以解决这个问题，但这种失败暴露了代码库理解能力的局限。

长期规划。 需要多步分解的复杂证明会给智能体带来挑战。当证明需要引入非平凡的辅助引理时，如果没有人工指导，智能体往往会迷失方向。将证明分解为多个阶段的渐进式提示会有所帮助，但智能体往往无法自行启动这种分解。

[8] https://apa.univalence.xyz/#/tasks/e69d9c6d-759a-413e-8927-43848ac726f0

创造性推理。 正如第 2.3 节所述，智能体无法提出新的数学洞见。当作为设计构思的思考伙伴时，智能体会跟进对话并认可各种想法，但无法提出原创且合理的想法。它们是有能力的证明工程师，但不是原创理论家。

2.5 讨论

我们的观察表明，正在形成两种工作流程：形式化的快速原型开发，以及机械化证明的潜在自然语言接口。

形式化的快速、可靠原型开发。 开发机械化形式化通常需要反复迭代：更新定义、修复因此失效的证明、发现原始设计中的问题，然后重复这一过程。瓶颈在于证明修复。单次定义更改就可能导致数十条引理失效，而在证明助手中手动修复这些引理既繁琐又耗时，会延迟对设计是否合理的反馈。

智能体证明自动化缩短了这一周期。人类更新定义；智能体修复受影响的证明；如果修复成功，设计便得到验证；如果没有成功，智能体通常会指出原因（例如第 2.3 节中的 `step_masked` 示例，智能体诊断出问题出在语义而非证明上）。无论如何，几分钟内就能获得反馈，而不必等待数小时。这种快速迭代促成了一种开发方式，使形式化验证成为设计过程的一部分，而不是事后强加的负担。

机械化证明的自然语言接口。 在整个案例研究中，我们观察到，人类与智能体之间的大多数交流都使用自然语言：陈述目标、勾勒策略、解释障碍。智能体将这些非正式描述转换为形式化证明脚本，并用自然语言报告结果。这种模式表明，人类或许可以纯粹通过自然语言与机器验证的证明进行交互，而智能体则充当人类数学直觉与机器检验的证明之间的转换层。

目前仍需要人工干预，这要求人类具备使用证明助手的专业知识。但随着模型不断改进，这一差距可能会缩小。数学家最终或许只需用通俗自然语言陈述定理，提供大致的证明思路，就能获得完整的机械化证明。如果推理存在缺陷，智能体也可以指出问题，甚至用自然语言解释反例。

3 定量评估

本节对人类与大语言模型智能体之间的交互进行定量分析，以刻画智能体证明自动化的性能模式。

3.1 数据收集与标注

原始数据由人类与智能体之间的对话历史组成。每段对话构成一个会话，其中包含双方发送的多条消息。筛除空白及无实质内容的交互（例如用于测试连接的消息，或不包含证明相关工具调用的会话）后，我们共得到 58 个会话，包含 513 条消息。

然后，我们手动将连续的消息分组为任务。一个任务代表一项独立的工作：由人类提出请求，随后由智能体给出一个或多个响应，直至完成或放弃。当人类提出多个请求时，一个会话可能包含多个任务。

每个任务都会标注所涉及工作的类别：

– **证明**：人类陈述一个定理；智能体对其进行证明。– **修复**：人类进行更改，导致现有证明失效；智能体修复这些问题。– **重构**：根据自然语言指令更新定义和证明。– **陈述并证明**：人类以自然语言描述一个性质；智能体对其进行陈述并证明。– **查询**：回答有关代码库或 Lean 概念的问题。– **例行维护**：执行修复样式警告或添加导入项等常规任务。

每个任务还会标注其结果：

– **成功**：智能体无需人工干预即可完成任务。– **经干预后成功**：智能体在获得人工指导（自然语言提示、代码修改或两者兼有）后完成任务。– **部分完成**：智能体取得进展，但仍有一些证明义务（`sorry`s）未完成。– **失败**：即使经过人工干预，智能体仍未能完成任务。– **发现问题**：智能体发现规范或推理中存在问题，并将其上报给人类。

我们总共对这 58 个会话中的 189 项任务进行了标注。所有原始会话和已标注的任务均可在在线浏览器中查看。

3.2 结果

表 1 总结了结果。智能体完成了 189 项任务中的 164 项（87%），仅有 9 项失败（5%），另有 16% 的任务需要人工干预。不同类别的表现有所差异：查询和例行维护的成功率均为 100%，因为它们不涉及证明合成；而四个与证明相关的类别成功率较低。

现有上下文的价值。 在与证明相关的任务中，智能体在修复任务上的表现最佳：成功率为 90%（43/48），干预率仅为 8%，且迭代次数较少（检查次数为 4.9 次，而证明任务为 8.3 次）。现有的有缺陷证明提供了宝贵的上下文。智能体能够看到预期的方法，通常只需进行局部修复，无需从头合成证明。相比之下，证明任务从一个简明的定理陈述开始，要求智能体独立发现证明策略。

表 1. 智能体证明自动化任务的定量评估。成功包括在人类干预下完成的任务。检查和行数分别显示 `lean4check` 调用次数和更改行数的平均值。干预显示需要人工指导的任务百分比。

类别	总计	成功	部分成功	失败	检查	行数	干预
证明	51	41	6	4	8.3	82.6	20%
修复	48	43	2	1	4.9	40.8	8%
重构	35	28	4	3	6。6	61。8	23%
状态+证明	23	20	2	1	8.3	129.3	35%
查询	21	21	0	0	0.3	0.1	0%
例行维护	11	11	0	0	2.1	5.9	9%
总计	189	164	14	9	5.9	60.2	16%

形式化很难。 陈述并证明任务的干预率最高（35%），产出的代码也最多（平均 129 行）。这些任务要求智能体先将非正式自然语言描述转换为形式化定理陈述，再进行证明。这个形式化步骤会引入歧义：人类可能需要澄清预期的规范，或纠正误解。一旦陈述正确，证明工作量便与证明任务相当（两者的检查次数平均都是 8.3 次）。

迭代带来成功。 `lean4check` 调用次数反映了任务难度。证明和陈述与证明任务需要最多的迭代（8.3 次），重构和修复需要较少迭代（分别为 6.6 次和 4.9 次），而查询任务几乎不需要迭代（0.3 次）。智能体与类型检查器之间的反馈循环至关重要：智能体提出证明步骤，接收错误消息，并不断完善其方法。越复杂的任务需要越多迭代。

4 相关工作

有三个研究领域与我们的案例研究相关：经典证明自动化与证明修复、神经定理证明，以及自主编码智能体。我们将在以下各节中分别综述这些领域。

4.1 证明自动化与证明修复

交互式定理证明器（ITP）的自动化传统上依赖符号搜索算法和启发式方法。在 Isabelle/HOL 生态系统中，Sledgehammer [6] 代表了“hammer”系统的先进水平。它将高阶逻辑目标转换为一阶逻辑问题，并将其交由外部自动定理证明器（ATP）处理。如果外部求解器找到证明，Sledgehammer 会在 Isabelle 的可信内核中重建该证明。在 Coq 中，Tactician [3] 等工具专注于从用户编写的策略

脚本中进行学习。Tactician 使用 k近邻（k-NN）算法，根据当前证明状态预测下一步策略，并从现有库中检索知识。与外部“hammer”系统不同，它直接在交互式定理证明器的策略语言中运行。对于 Lean 4，Aesop [18] （显然证明的自动化可扩展搜索）提供了一个白盒自动化框架。Aesop 会在用户可配置的一组规则上执行最佳优先树搜索。虽然它在解决常规目标方面非常有效，但从根本上说，Aesop 是一种由固定启发式方法驱动的搜索算法。

与证明自动化相关的是证明修复：在规格发生变化时调整证明。Pumpkin Patch [25] 开创了基于示例的补丁修复，随后 Pumpkin Pi [24], 将其推广，通过在类型等价关系之间迁移证明，支持复杂的重构，包括代数装饰 [26] 和商类型 [34]。Sisyphus [14] 则通过动态分析来合成循环不变量，以补充这些静态方法。这些工具擅长处理结构化、可预测的变更，但缺乏灵活性，无法应对任意修改，也不能以自然语言解释失败原因。

4.2 基于大语言模型的交互式定理证明

大语言模型与交互式定理证明器（ITP）的结合通常称为神经定理证明（NTP），其发展十分迅速。早期研究（如 GPT-f [22] ）表明，在形式化库上训练的 Transformer 能够生成有效的证明步骤。Llemma [2] 通过在数学代码上进行预训练，进一步提升了这一能力。LeanDojo [39]实现了与 Lean 的细粒度交互，而 ReProver [39] 则引入了检索增强生成，从 Mathlib 中选取相关前提。

近期方法已从静态的下一步预测转向通过交互学习的智能体工作流。Copra [29] 在有状态的回溯搜索中使用 GPT-4，并读取错误消息以剪除错误路径。DeepSeek- Prover [36] 和 InternLM-StepProver [35] 将强化学习与蒙特卡洛树搜索相结合。AlphaProof [1] 通过在数百万个合成问题上训练，取得了奥林匹克竞赛级表现。其他系统则侧重于人类参与或持续学习工作流：Lean Copilot [27] 以交互方式建议策略，而 LeanAgent [16] 使用课程学习逐步攻克难度更高的定理。

我们的案例研究在范围和设置上与这些系统不同。大多数神经定理证明（NTP）研究针对数学基准测试，其中每个问题都是一个独立的定理陈述。我们的工作聚焦于大型且不断演进的形式化开发中的上下文内证明工程，在这种开发中，证明依赖于周围的定义、引理和设计决策。此外，我们不使用后训练：我们的设置完全依赖现成的大语言模型智能体，并仅使用一个工具来获取编译器反馈。这种简洁性让从业者能够立即采用这一工作流程，也使其随着前沿模型的进步而不断改进。

4.3 用于软件工程的智能体大语言模型

我们的案例研究也与自主软件工程智能体的发展相关；这类智能体采用相同范式，即探索结构化环境（代码库）以实现目标。

SWE-agent [38] 表明，简化文件导航和编辑命令，能显著提升大语言模型自主解决 GitHub 问题的能力。与我们的领域更接近的是，ProofWright [10] 将智能体工作流应用于 CUDA 内核的形式化验证，同时生成代码和正确性证明。然而，ProofWright 专注于高性能内核，而我们的案例研究则考察了通用编程智能体在编程语言语义复杂元理论形式化中的应用。

5 结论

我们介绍了一项智能体证明自动化案例研究：借助集成简单证明检查器的方式，使用现成的大语言模型智能体，在 Lean 4 中将 System Capless 的语义类型可靠性形式化，代码超过 14,000 行。对 189 项任务进行的定量求值显示，成功率为 87%，仅有 16% 的任务需要人工干预。该案例研究揭示了自然的分工方式：人类提供数学洞见（定义、定理和证明策略），智能体则通过依据编译器反馈进行迭代改进来处理证明工程。全部 58 个会话和 189 项带注释的任务均可通过交互式浏览器查看，读者可以检查原始交互，并了解指导模式如何转化为证明尝试。该案例研究的形式化成果也已开源，可供试验和扩展。

我们的案例研究所展示的智能体证明自动化方案并不局限于 Lean，且会随着前沿模型的改进而扩展。随着智能体能力不断增强，机械化证明的门槛随之降低，让缺乏多年定理证明器经验的从业者和理论家也能更轻松地开展经机器检验的证明开发。

我们的案例研究考察了 Lean 4 中的单个形式化开发。未来一个自然的研究方向是为智能体证明自动化建立标准化基准测试。不同于评估孤立定理证明任务的现有基准测试，这类基准测试将评估在不断演进的代码库中、结合上下文开展的证明工程。

参考文献

1. AlphaProof 团队、AlphaGeometry 团队：AI 达到国际数学奥林匹克竞赛银牌水平。Google DeepMind 博客（2024年），https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/，发表于2024年7月25日 2. Azerbayev, Z., Schoelkopf, H., Paster, K., Santos, M.D., McAleer, S., Jiang, A.Q., Deng, J., Biderman, S., Welleck, S.：Llemma：面向数学的开放语言模型。CoRR **abs/2310.10631**（2023），https://arxiv.org/abs/2310.10631

3. Blaauwbroek, L., Urban, J., Geuvers, H.: The tactician——一款无缝、交互式的 Coq 策略学习器与证明器。载于：智能计算数学——第 13 届国际会议，CICM 2020，意大利贝尔蒂诺罗，2020年7月26—31日，会议论文集。计算机科学讲义，第 12236 卷，第 271–277 页。施普林格（2020）。https://doi.org/ 10 1007 978 3 030 53518 6_17 。/ - - - - . Blaauwbroek, L., Urban, J., Geuvers, H.: The Tactician：一款无缝、交互式的 Coq 策略学习器与证明器。载于：智能计算数学（CICM）。LNCS，第 12236 卷，第 271–277 页。施普林格（2020）。https://doi.org/10.1007/ 978-3-030-53518-6 17_5. Blanchette, J.C., Böhme, S., Paulson, L.C.：使用 SMT 求解器扩展 Sledgehammer。自动推理杂志 **51**(1)，109–128（2013）。https://doi.org/ 10 5 6 1007 10817 013 9278 . /s - - - . Böhme, S., Nipkow, T.: Sledgehammer：审判日。载于：自动推理——第 5 届国际联合会议，IJCAR 2010，英国爱丁堡，2010年7月16—19日，会议论文集。计算机科学讲义，第 6173 卷，第 107–121 页。施普林格（ 2010 10 1007 978 3 642 14203 1_9 ）。https://doi.org/ . / - - - - . Boruch-Gruszecki, A., Odersky, M., Lee, E., Lhoták, O., Brachthäuser, J.：捕获类型。ACM Trans. Program. Lang. Syst. **45**(4)，21:1–21:52（2023）。https: //doi.org/10 3618003 8 1145 3618003, 10 1145 . / https://dl.acm.org/doi/ . / . Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., 等：语言模型是少样本学习者。载于：神经信息处理系统进展。第 33 卷，第 1877–1901 页（2020）9. Böhme, S., Nipkow, T.: Sledgehammer：审判日。载于：自动推理国际联合会议（IJCAR）。LNCS，第 6173 卷，第 107–121 页。施普林格（ 2010 10 1007 978 3 642 14203 1_9 ）。https://doi.org/ . / - - - - . Chatterjee, B., 等：ProofWright：迈向智能体式 CUDA 形式化验证。CoRR **abs/251**112294 11**2294** 2025 2511.( )，https://arxiv.org/abs/ . . Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H.W., Sutton, C., Gehrmann, S., 等：PaLM：利用路径扩展语言建模（2022）12. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.：BERT：用于语言理解的深度双向 Transformer 预训练。arXiv 预印本 arXiv:1810.04805（2018）13. Gauthier, T., Kaliszyk, C., Urban, J.: TacticToe：学习使用策略进行证明。自动推理杂志 **65**(2)，257–286（2021）。https://doi.org/10.1007/ s10817-020-09580-x 14. Gopinathan, K., Keoliya, M., Sergey, I.: 经验证的库的大部分自动化证明修复。载于：第 44 届 ACM SIGPLAN 编程语言设计与实现国际会议（PLDI）论文集。第 25–49 页。ACM（2023 3591239 15 10 1145）。https://doi.org/ . / . Hamza, J., Voirol, N., Kuncak, V.：System FR：Stainless 验证器的形式化基础。载于：ACM 编程语言论文集，OOPSLA。第 3 卷，3360592 16 148 1 148 30 2019 10 1145 pp. : – : ( )。https://doi.org/ . / . Kumarappan, A., Tiwari, M., Song, P., George, R.J., Xiao, C., Anandkumar, A.：LeanAgent：面向形式化定理证明的终身学习。CoRR **abs/2410.06209**（2024 06209 17 2410），https://arxiv.org/abs/ . . Leroy, X.：现实编译器的形式化验证。《ACM 通讯》**51**(7)，107–115（2008）。https://doi.org/10.1145/1378704.1378709，https://doi.org/ 10.1145/1378704.1378709

18. Limperg, J., From, A.H.：Aesop：用于 Lean 的白盒最佳优先证明搜索。载于：第12届 ACM SIGPLAN国际认证程序与证明会议（CPP 2023）论文集，美国马萨诸塞州波士顿，2023年1月16日至17日。第253–266页。ACM（2023）。https://doi.org/10.1145/3573105.3575671。Limperg, J., From, A.H.：Aesop：用于 Lean 的白盒最佳优先证明搜索。载于：认证程序与证明（CPP）。第253–266页。ACM（2023）。https://doi.org/10.1145/3573105.3575671。de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.：Lean定理证明器（系统描述）。载于：Ollinger, N., Kirchner, H.（编）自动演绎——CADE 26。第378–388页。施普林格（2017）。https://doi.org/10.1007/978-3-319-61407-3_26_21. Nipkow, T., Paulson, L.C., Wenzel, M.：Isabelle/HOL：高阶逻辑证明助手，计算机科学讲义，第2283卷。施普林格（2002）。https://doi.org/10.1007/3-540-45949-9。Polu, S., Sutskever, I.：用于自动定理证明的生成式语言建模。CoRR **abs/2009**03393 2**303393** 2020 2009。（），https://arxiv.org/abs/。Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I.：语言模型是无监督多任务学习器。OpenAI博客 **1**（4），9（2019）24. Ringer, T., Porter, R., Yazdani, N., Leo, J., Grossman, D.：跨类型等价的证明修复。载于：第42届 ACM SIGPLAN 编程语言设计与实现国际会议（PLDI）论文集。第112–127页。ACM（2021）。https://doi.org/10.1145/3453483.3454033。Ringer, T., Yazdani, N., Leo, J., Grossman, D.：调整证明自动化以适配证明。载于：第7届 ACM SIGPLAN国际认证程序与证明会议（CPP）论文集。第115–129页。ACM（2018）。https://doi.org/10.1145/3167080。Ringer, T., Yazdani, N., Leo, J., Grossman, D.：Coq中的证明复用装饰。载于：第10届国际交互式定理证明会议（ITP 2019）。LIPIcs，第141卷，第26:1–26:19页（2019）。https://doi.org/10.4230/LIPIcs.ITP.2019.26。27. Song, P., Yang, K., Anandkumar, A.：让大语言模型成为 Lean 定理证明的副驾驶。CoRR **abs/2404.12534**（2024），https://arxiv.org/abs/2404.12534。Team, T.C. D.：Coq证明助手参考手册。INRIA、CNRS、巴黎综合理工学院（2020），https://coq.inria.fr，版本8.12。Thakur, A., Wen, Y., Chaudhuri, S.：COPRA：用于形式化定理证明的上下文学习。CoRR **abs/2310**04353 3**004353** 2023 2310。（），https://arxiv.org/abs/。Timany, A., Krebbers, R., Dreyer, D., Birkedal, L.：类型健全性的逻辑方法。《ACM期刊》**71**（6）（2024年11月）。https://doi.org/10.1145/3676954，https://doi.org/10.1145/3676954。Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Zoph, B.等：Llama：开放且高效的基础语言模型（2023）。32. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.：注意力机制就是你所需要的一切。神经信息处理系统进展**30**（2017）。33. Vazou, N., Jhala, R., Horn, D.V., Kong, S.：LiquidHaskell：真实世界中的细化类型实践经验。载于：ACM SIGPLAN函数式编程国际会议（ICFP）。第39–51页。ACM（2014）。https://doi.org/10.1145/2633357.2633366

34. Viola, C., Fan, M., Ringer, T.：跨商类型等价的证明修复。《ACM 编程语言论文集》**9**（OOPSLA2）（2025），即将发表。35. Wu, Z., Huang, S., Zhou, Z., Ying, H., Wang, J., Lin, D., Chen, K.：Internlm2.5-stepprover：通过在大规模 Lean 问题上进行专家迭代，推进自动定理证明。CoRR **abs/2410.15700**（2024），https://arxiv.org/abs/2410.15700。36. Xin, H., Ren, Z.Z., Song, J., Shao, Z., Zhao, W., Wang, H., Liu, B., Zhang, L., Lu, X., Du, Q., Gao, W., Zhu, Q., Yang, D., Gou, Z., Wu, Z.F., Luo, F., Ruan, C.：Deepseek-prover-v1.5：利用证明助手反馈进行强化学习和蒙特卡洛树搜索。CoRR **abs/2408.08152**（2024），https://arxiv.org/abs/2408.08152。Xu, Y., Bračevac, O., Pham, C.N., Odersky, M.：《盒子里有什么：在泛型数据结构上实现符合人体工学且富有表现力的捕获跟踪》。《ACM 编程语言论文集》**9**（OOPSLA2），1726–1753（2025年10月）。https://doi.org/10.1145/3763112。Yang, J., Jimenez, C.E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., Press, O.：SWE-agent：智能体-计算机接口助力自动化软件工程。载于：神经信息处理系统进展37：神经信息处理系统年度会议2024，NeurIPS 2024，加拿大不列颠哥伦比亚省温哥华，2024年12月10日至15日（2024），https://arxiv.org/abs/。Yang, K., Swope, A.M., Gu, A., Chalamala, R., Song, P., Yu, S., Godber, S., Prenger, R., Anandkumar, A.：LeanDojo：结合检索增强语言模型的定理证明。载于：神经信息处理系统进展36：神经信息处理系统年度会议2023，NeurIPS 2023，美国路易斯安那州新奥尔良，2023年12月10日至16日（2023），https://arxiv.org/abs/2306.15626

A System Capless 的形式化定义

本附录介绍 System Capless 的形式化定义。System Capless 是本文的案例研究中用 Lean 4 形式化的目标系统。System Capless 是一种用于捕获检查的核心演算，它通过捕获多态扩展了 System $\mathrm{CC}_{<:\Box}$ [7] 。

A.1 语法

x, y, z 变量 $\lambda[X <: S]t$ $a := x \mid v$ 答案X 类型抽象 $\theta := x \mid c$ 捕获类型变量 $\lambda[c <: B]t$ 捕获抽象 $C, D := \{\theta_1, \cdots, \theta_n\}$ 捕获集 捕获变量$\overset{c}{\text{量}}\langle C,x\rangle$ 打包 $B := * \mid Cs$, t, u:= 项 $E, F :=$ 捕获界限a 答案 存在类型 $T, U :=$ 类型 $x\,y$ 应用 $\exists c.\,T$存在 $S \wedge C$捕获$x[S]$ 类型应用 T 类型 S 纯$x[c]$ 捕获应用 $R, S :=$ 形状类型 $\top$ 顶层 $\Gamma, \Delta :=$ 上下文 $\emptyset$ 空 let $x = t$ in u X 类型变量 $\Gamma, x : T$ let $\forall(x : T)E$ 项绑定 let $\langle c, x\rangle = t$ in u 项函数 $\Gamma, X <: S$存在 let $\forall[X <: S]E$ 类型绑定 $v :=$ 值 $\lambda(x : T)t$ 项抽象 类型函数 $\forall[c <: B]E$ 捕获函数 $\Gamma, c <: B$ 捕获绑定

图 2. System Capless 的抽象语法。

图 2 展示了 System Capless 的抽象语法。相较于 $\mathrm{System\ CC}_{\mathsf{formal}}$ ，其关键扩展是增加了捕获变量 c，从而能够通过捕获抽象 $\lambda[c <:$、捕获应用 $x[c]$ 和存在类型 $\exists c.\,T$ 实现捕获多态： $B]t$。

A.2 类型系统

图 3 展示了 System Capless 的类型规则。类型判断的形式为 $C; \Gamma \vdash t : E$，其中 C 是项的使用集， Γ 是类型上下文，t 是待进行类型判断的项， E 是导出类型。使用集 对该项在求值为一个答案时可能使用的能力集合进行过度近似。

类型

$$\frac{C;\Gamma\vdash t:\ E \qquad x:\ S\mathbin{\wedge}C\in\Gamma}{\{x\};\ \Gamma\vdash x:\ S\mathbin{\wedge}\{x\}}\ \text{(变量)}$$

$$\frac{C';\ \Gamma\vdash x:\ (\forall[X<:\ S]E)\mathbin{\wedge}C}{C';\ \Gamma\vdash x[S]:\ [X:=S]E}\ \text{（应用）}$$

$$\frac{C';\Gamma\vdash x:\ [c:=C]T}{\{\};\Gamma\vdash\langle C,\ x\rangle:\ \exists c.\,T}\ \text{(打包)}$$

$$\frac{C;\ (\Gamma,\ c<:\ B)\vdash t:\ E \qquad \Gamma\vdash C\ \text{工作流}}{\{\};\ \Gamma\vdash\lambda[c<:\ B]t:\ (\forall[c<:\ B]E)\mathbin{\wedge}C}\ \text{(cabs)}$$

$$\frac{C;\Gamma\vdash t:\ E \qquad \Gamma\vdash E<:\ F \qquad \Gamma\vdash C<:\ C' \qquad \Gamma\vdash C',F\ \mathsf{wf}}{C';\ \Gamma\vdash t:\ F}\ \text{（子）}$$

$$\frac{C';\ \Gamma\vdash x:\ (\forall[c<:\ D]E)\mathbin{\wedge}C}{C';\ \Gamma\vdash x[D]:\ [c:=D]E}\ \text{(capp)}$$

$$\frac{C;\ (\Gamma,\ x:\ T)\vdash t:\ E \qquad \Gamma\vdash T\,\mathsf{wf}}{\{\};\ \Gamma\vdash\lambda\ (x:\ T)\ t:\ (\forall\ (x:\ T)\ E)\mathbin{\wedge}(C\setminus x)}\ \text{(abs)}$$

$$\frac{C;\Gamma\vdash t:\ T \qquad C;(\Gamma,\ x:\ T)\vdash u:\ E \qquad \Gamma\vdash C,E\ \mathsf{wf}}{C;\Gamma\vdash\text{令}\ \ x=t\ \ \text{于}\ \ u:\ E}\ \text{(令)}$$

$$\frac{C';\ \Gamma\vdash x:\ (\forall(z:\ T)E)\mathbin{\wedge}C \qquad C';\Gamma\vdash y:\ T}{C';\Gamma\vdash x\,y:\ [z:=y]E}\ \text{(应用)}$$

$$\frac{C;(\Gamma,\ X<:\ S)\vdash t:\ E \qquad \Gamma\vdash S\,\mathsf{wf}}{\{\};\ \Gamma\vdash\lambda[X<:\ S]t:\ (\forall[X<:\ S]E)\mathbin{\wedge}C}\ \text{（制表符）}$$

$$\frac{C;\Gamma\vdash t:\ \exists c.\,T \qquad C;(\Gamma,c<:\ *,x:\ T)\vdash u:\ F \qquad \Gamma\vdash C,\ F\ \text{良构}}{C;\ \Gamma\vdash\text{令}\ \ \langle c,\ x\rangle=t\ \text{于}\ \ u:\ F}\ \text{(let-e)}$$

子捕获 $\Gamma\vdash C_1<:\ C_2$

$$\frac{\Gamma\vdash C_1<:C_2 \qquad \Gamma\vdash C_2<:C_3}{\Gamma\vdash C_1<:C_3}\ (\text{SC-TRANS}) \qquad \frac{x:S\mathbin{\wedge}C\in\Gamma}{\Gamma\vdash\{x\}<:C}\ (\text{SC-VAR}) \qquad \frac{c<:C\in\Gamma}{\Gamma\vdash\{c\}<:C}\ (\text{SC-BOUND}) \qquad \frac{C_1\subseteq C_2}{\Gamma\vdash C_1<:C_2}\ (\text{SC-ELEM}) \qquad \frac{\Gamma\vdash C_1<:C \qquad \Gamma\vdash C_2<:C}{\Gamma\vdash C_1\cup C_2<:C}\ (\text{SC-SET})$$

界限子类型 $\Gamma\vdash B_1<:\ B_2$ 与子捕获相同，另加 $\Gamma\vdash B<:\ *$

Subtyping $\Gamma\vdash E_1<:E_2$

$$\Gamma\vdash S<:\top\,(\text{TOP}) \qquad \frac{\Gamma\vdash E_1<:E_2 \qquad \Gamma\vdash E_2<:E_3}{\Gamma\vdash E_1<:E_3}\ (\text{TRANS}) \qquad \frac{X<:S\in\Gamma}{\Gamma\vdash X<:S}\ (\text{TVAR}) \qquad \frac{\Gamma\vdash S_1<:S_2 \qquad \Gamma\vdash C_1<:C_2}{\Gamma\vdash S_1\mathbin{\wedge}C_1<:S_2\mathbin{\wedge}C_2}\ (\text{CAPT})$$

$$\Gamma\vdash E<:E\,(\text{REFL})$$

$$\frac{(\Gamma,c<:*)\vdash T_1<:T_2}{\Gamma\vdash\exists c.\,T_1<:\exists c.\,T_2}\ (\text{EXIST}) \qquad \frac{(\Gamma,X<:S_2)\vdash E_1<:E_2 \qquad \Gamma\vdash S_2<:S_1}{\Gamma\vdash\forall[X<:S_1]E_1<:\forall[X<:S_2]E_2}\ (\text{TFUN})$$

$$\frac{(\Gamma,x:T_2)\vdash E_1<:E_2 \qquad \Gamma\vdash T_2<:T_1}{\Gamma\vdash\forall(x:T_1)E_1<:\forall(x:T_2)E_2}\ (\text{FUN}) \qquad \frac{(\Gamma,c<:B_2)\vdash E_1<:E_2 \qquad \Gamma\vdash B_2<:B_1}{\Gamma\vdash\forall[c<:B_1]E_1<:\forall[c<:B_2]E_2}\ (\text{CFUN})$$

图 3. System Capless 的类型规则。

A.3 求值规则

语法

$$
\begin{array}{lll}
\Sigma ::= & & \textbf{存储} \\
\quad \emptyset & & \text{空} \\
\quad \Sigma,\ \textbf{值}\, x \mapsto v & & \text{值绑定} \\
\Psi ::= & & \textbf{求值上下文} \\
\quad [] & & \text{hole} \\
\quad \text{令 } x = \Psi \text{ in } t & & \text{let} \\
\quad \text{令 } \langle c,\ x\rangle = \Psi \text{ in } t & & \text{例如 let}
\end{array}
$$

归约 $\langle \Sigma \mid t\rangle \longrightarrow \langle \Sigma' \mid t'\rangle$

$$\frac{\Sigma(x) = \lambda(z : T)t}{\langle \Sigma \mid \Psi[x\ y]\rangle \longrightarrow \langle \Sigma \mid \Psi[[z := y]t]\rangle} \quad (\textsc{apply})$$

$$\frac{\Sigma(x) = \lambda[X <: S]t}{\langle \Sigma \mid \Psi[x[S']]\rangle \longrightarrow \langle \Sigma \mid \Psi[[X := S']t]\rangle} \quad (\textsc{tapply})$$

$$\frac{\Sigma(x) = \lambda[c <: B]t}{\langle \Sigma \mid \Psi[x[C]]\rangle \longrightarrow \langle \Sigma \mid \Psi[[c := C]t]\rangle} \quad (\textsc{capply})$$

$$\langle \Sigma \mid \Psi[\mathsf{let}\ x = y\ \mathsf{in}\ t]\rangle \longrightarrow \langle \Sigma \mid \Psi[[x := y]t]\rangle \quad (\textsc{rename})$$

$$\langle \Sigma \mid \Psi[\mathsf{let}\ \langle c, x\rangle = \langle C, y\rangle\ \mathsf{in}\ t]\rangle \longrightarrow \langle \Sigma \mid \Psi[[x := y][c := C]t]\rangle \quad (\textsc{rename-e})$$

$$\langle \Sigma \mid \Psi[\text{令 } x = v \text{于 } t]\rangle \longrightarrow \langle \Sigma,\ \textbf{值}\, x \mapsto v \mid \Psi[t]\rangle \quad (\textsc{lift})$$

图 4. System Capless 的求值规则。

图 4 展示了 System Capless 的求值规则。这些规则遵循小步操作语义，使用存储 Σ 存放值绑定，并使用求值上下文 Ψ 指定求值顺序。相较于 System $\mathrm{CC}_{\mathsf{formal}}$，其关键新增内容是用于捕获应用的 (capply) 规则和用于存在量词解包的 (rename- e) 规则。