arXiv:2602.04846v1 [cs.LO] 4 Feb 2026

CSLib：Lean 计算机科学库

Clark Barrett[1,2], Swarat Chaudhuri[3,4],Fabrizio Montesi[5],Jim Grundy[1],Pushmeet Kohli[3], Leonardo de Moura[6,1], Alexandre Rademaker[7,8],和 Sorrachai Yingchareonthawornchai[9]

[1]亚马逊[2]斯坦福大学[3]Google DeepMind[4]德克萨斯大学奥斯汀分校[5]FORM，南丹麦大学[6]Lean FRO[7]Atlas Computing[8]Getulio Vargas Foundation[9]苏黎世联邦理工学院理论研究所¨

摘要

我们推出 CSLib，这是一个开源框架，可用于在 Lean 证明助手中证明计算机科学相关定理并编写形式化验证代码。CSLib 旨在成为计算机科学领域的 Mathlib，正如 Lean 的 Mathlib 服务于数学领域一样。Mathlib 影响深远：它是 Lean 在数学研究界广受欢迎的关键原因，也在训练用于数学推理的人工智能系统方面发挥了重要作用。然而，目前 Lean 中计算机科学知识的基础仍相当有限。CSLib 将大幅扩充这一知识库，并提供基础设施，以便在现实世界的验证项目中运用这些知识。为此，CSLib 将（1）推动 Lean 在计算机科学教育与研究中的广泛应用；（2）促进大规模形式化验证系统的人工和人工智能辅助工程开发。

1 引言

表面上看，Mathlib [1] 是一个库，其中收录了用 Lean [32]证明助手形式化的数学结果和技术。实际上，它远不止于此——这是世界各地数学家共同参与的一项社区工作，旨在以一种便于审查和共享的形式汇集全世界的数学知识。Mathlib 为一种新的数学研究方式提供了平台，让许多人能够满怀信心地协作完成规模庞大且富有挑战性的项目 [38]。此外，通过以机器可检查的形式表示广泛的数学定义和证明，它为人工智能系统辅助数学发现过程开辟了新的机会 [44, 19]。新一代数学家正在接受这种新的数学研究方式，并且它也赢得了许多知名拥护者 [41, 24]。Mathlib 是这一范式转变的核心，因此其影响怎么强调都不为过。

计算机科学始终与数学密切相关。许多计算机科学系最初都是从数学系分支出来的，时至今日，计算机科学理论与数学之间的界限仍然模糊。此外，计算机科学家早已认识到形式化数学

是一种强大的工具，能够助力设计可靠且安全的系统 [4, 9, 18]。然而，迄今为止，还没有人尝试建立一个类似 Mathlib 的形式化计算机科学知识通用仓库。

在本白皮书中，我们介绍 CSLib，[1] 一个基于 Lean 的形式化验证与计算机科学研究库，旨在改变这一现状。CSLib 有两大“支柱”：

1. 在 *Lean* 中形式化所有重要的计算机科学概念。这些形式化内容包括但不限于： λ-演算和资源受限图灵机等计算模型；附有正确性与复杂度证明的众多算法与数据结构；并发理论；编程语言基础；以及目前尚未收录于 Mathlib、但与计算机科学相关的数学工具。

2. 构建基础设施，以便基于 *Lean* 对日常命令式代码进行推理。该基础设施包括一种名为 Boole 的中间编程语言，允许将经典命令式结构与用 Lean 编写的规格说明交错使用；用于从 Boole 代码生成 Lean 语言验证条件的工具；以及对支柱 1 所涵盖算法与数据结构的、经过验证的 Boole实现。

总的来说，CSLib 中的 Lean 代码与证明，使 Lean 能够成为开展具有数学基础的计算机科学研究与开发的媒介。例如，我们可以设想：理论研究者利用 CSLib 内置的抽象，快速搭建一种新颖的近似算法，并提出刻画其基本性质的定理；程序语言研究者利用这一框架构建经过验证的编译器；或者系统研究者利用该库为新的网络协议建模，并给出最坏情况保证。Boole 框架为 Lean 与日常使用的编程语言搭建了一座重要桥梁。具体来说，我们设想这样一个未来：Rust 和 C++ 等主流语言中的代码能够大规模自动转换为 Boole。CSLib 内置的基础设施将验证这些 Boole 语言程序的任务转化为证明 Lean 语言定理的任务。随后，便可借助 Lean 语言的证明机制完成这项证明工作。

CSLib的影响。 我们认为，CSLib 的影响体现在两个方面：满足对可靠软件系统的迫切实际需求，以及满足对更严谨、更具可扩展性的计算机科学的长期需求。

首先，软件可靠性和安全性方面的进展比以往任何时候都更加刻不容缓，因为人工智能的发展给全球计算基础设施带来了新的风险。如今，人工智能系统拥有出色的代码分析能力，使人工智能驱动的恶意行为者更容易发现并利用漏洞。与此同时，人工智能编程代理正在编写越来越多的全球代码。虽然这提高了整体生产力，但人工智能也可能在代码中引入正确性错误和安全漏洞 [36]。形式化方法可以通过数学证明，确保代码——无论由人编写还是由人工智能生成——不存在程序错误和漏洞，从而降低这些风险。

然而，形式化方法的影响力长期以来受到其所需巨大人力投入的限制。以 seL4 [23]，为例，它是世界上首个经过形式化验证的操作系统内核。该系统影响深远：在 DARPA HACMS计划中，一架由 seL4 驱动的无人机曾在精锐红队持续六周的攻击下安然无恙，因此被誉为“世界上最安全的无人机”。但 seL4 也凸显了传统形式化验证所伴随的成本代价。该系统耗费了 20 多人年的证明工作，并在 Isabelle [34] 框架中包含约 48 万行形式化规格说明和证明。在大多数商业环境中，这样的成本难以证明其合理性。

[1] CSLib 主网站位于 https://cslib.io。当前 CSLib 代码库位于 https://github.com/ leanprover/cslib/。

CSLib 的愿景从两个方面着手解决这一瓶颈。CSLib 建成后，开发者便能够通过组合经过预先验证的 CSLib 组件，以组合式方式构建可靠且安全的系统。其次，CSLib 能够简化人工智能在形式化验证中的应用。人工智能系统在纯数学领域的形式化证明能力已经展现出超越人类的水平 [19]。然而，要将这些能力用于广泛领域的代码验证，我们需要一种能够描述代码性质的综合性语言。Lean 和 CSLib 将提供这种语言。此外，CSLib 还将作为高质量数据存储库，用于训练人工智能系统进行形式化和形式化证明。

除了对实际形式化验证的影响之外，CSLib 还将成为 21 世纪计算机科学研究的平台。CSLib 所支持的形式化验证将增强人们对各项主张的信心，并让基础概念得以大规模复用。我们还可以设想，基于 CSLib 训练的人工智能工具能够发现新颖算法并解决尚未解决的主张，从而让人工智能代理开始在数学研究中发挥的作用也延伸至计算机科学领域。

2 技术方案

现在，我们将进一步介绍 CSLib 技术方案的两大支柱。

2.1 支柱 1：在 Lean 中形式化计算机科学

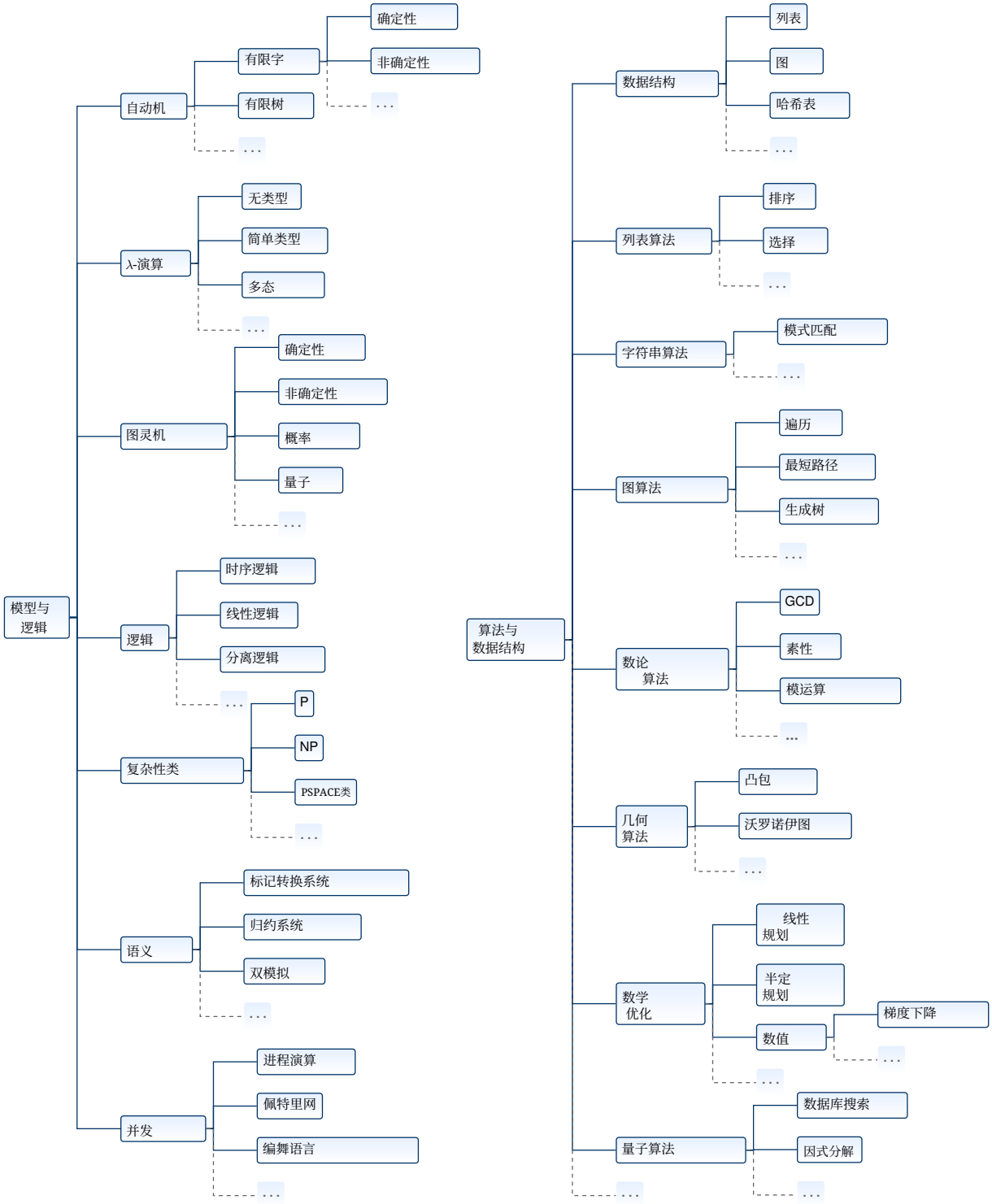
CSLib 的第一项发展支柱是利用 Lean 对一套全面的计算模型、算法和数据结构进行形式化，并形式化这些对象的性质及其证明。这些形式化旨在构成一个连贯、集成的框架，而不是一组互不相关的模块。这种统一使整体大于各部分之和，也是 Mathlib 产生重大影响的关键。

这一支柱与 Mathlib 具有协同关系。一方面，它大量使用 Mathlib 的模块，例如用于大O推理和概率论的模块。另一方面，它还会形式化一些核心数学内容——例如，分析近似算法所需的某些组合数学引理，或证明优化算法收敛性所需的不等式——这些内容在计算机科学中很有用，但目前尚未出现在 Mathlib 中。根据具体情况，我们可能会将这些形式化成果贡献给 Mathlib，也可能将其保留在 CSLib 代码库中。

现在，我们进一步介绍这一支柱的目标。

模型与逻辑。 多年来，计算机科学家设计了种类繁多的计算模型。其中包括确定性、非确定性、概率和量子计算模型；函数式计算、在线算法和交互式协议模型；用于解决判定问题的模型；以及用于近似求解优化问题的模型。这些模型中有许多从未经过形式化；已经形式化的模型往往散落在彼此孤立的代码库中。CSLib 支柱 1 的一个关键目标是创建一个统一的形式化模型体系。

另一项工作是将模型和算法的规约表示法形式化。例如，时序逻辑适合描述有状态系统的性质；霍尔逻辑是描述过程式代码的标准机制；分离逻辑简化了低级命令式计算的规约；线性逻辑适用于推理资源使用和并发行为。目前，还没有包含此类规约表示法的统一代码库，CSLib 将致力于解决这一问题。



(a) 模型与逻辑　　(b) 算法与数据结构

图 1：CSLib 的支柱 1：所涵盖主题的一种可能组织方式。

(a)

```
结构体 LTS
    (状态: 类型u)
    (标签: 类型v) 其中
  /-- 迁移关系。 -/
  Tr: 状态 →标签 → 状态 → 命题
```

(b)

```
定义 LTS.双模拟关系
    (lts : 标记迁移系统 状态 标签)
    (r : 状态 → 状态 → 命题): 命题 :=
∀ {s1 s2|},   r s1 s2   → ∀ μ, (
   (∀ s1',   lts.Tr s1   μ s1' → ∃ s2',   lts.Tr s2
    μ s2' ∧ r s1' s2')
  ∧
   (∀ s2',   lts.Tr s2 μ s2' → ∃ s1',   lts.Tr s1
    μ s1' ∧ r s1' s2')
)
```

(c)

```
定理 LTS.IsBisimulation.inv
    (h : LTS.双模拟关系 r) :
  lts.IsBisimulation (flip r): = 证明

  简化 仅 [是双模拟] 于 h
  简化 仅 [双模拟关系]
  介绍 s1 s2 hrinv μ

  constructor
  case left =>
    引言 s1' htr
    特化 h hrinv μ
    假设 h' : = h.2 s1' htr
    获得 ⟨s2', h'⟩ := h'
    存在 s2'
  情形 right =>
    引入 s2' htr
    特化 h hrinv μ
    设 h' : = h.1 s2' htr
    获取 ⟨s1', h'⟩ := h'
    存在 s1'
```

图 2: （a）标记转换系统（“LTS”）的定义，该系统用于对离散计算系统可能状态的可观测行为进行建模。（b）两个状态之间双模拟的定义。（c）定理：双模拟的逆关系也是双模拟。

虽然我们将在这一支柱中构建的具体形式化集合必然会随着时间推移而演变，但图 1-（a）列出了我们目前计划涵盖的一些模型和逻辑。图 2-（a）和图 2-（b）给出了形式化的具体示例。具体而言，图 2-（a）展示了标记迁移系统 [43], 的 Lean 定义的一部分，这是一种有状态系统的经典模型；图 2-（b）则定义了双模拟，这是系统状态之间等价关系的一个关键概念。如果两个状态 s_1 和 s_2 通过双模拟相关，那么 s_1 就能通过相应的迁移来模拟 s_2 的所有迁移，反之亦然；通过这些原始迁移和模拟迁移到达的状态仍然由该双模拟保持相关。

迁移系统与双模拟在计算机科学中有众多应用，涵盖从硬件设计 [8] 到控制 [13] 再到机器学习 [45]等领域。因此，这些概念的形式定义及相关定理（如图 2-(c) 中的定理）无论从理论还是实践角度都具有价值。然而，此类形式化并不自然地属于 Mathlib 的范围，因此体现了 CSLib 将提供的独特能力。

算法与数据结构。 支柱 1 还将使用 Lean 构建一个全面的、经过形式验证的算法与数据结构仓库。我们的长期目标是让该库覆盖整个计算机科学。中期而言，我们希望该库覆盖典型计算机科学毕业生可能接触到的所有算法与数据结构。图 1-(b) 列出了我们计划覆盖的部分算法类别。

本讨论串的一个关键目标是对算法分析进行形式化，尤其是对算法的 复杂度 进行形式化。我们注意到，此前已有大量关于程序自动化、轻量级复杂度分析的工作 [15, 39], , 以及一些关于特定算法的基于证明助手的复杂度验证[7, 6] 和在 Isabelle 中进行形式化的工作 [33]。然而，据我们所知，只有一项工作 [20] 致力于在 Lean 中系统地形式化算法复杂度。该形式化 [5]专注于随机预言机

```
结构体 TimeM (α : 类型*) 其中
  返回值: α
  时间: ℕ

命名空间 TimeM

受保护的定义 纯函数 {α} (a: α) : TimeM α :=
  ⟨a, 0⟩

定义绑定 {α β} (m: TimeM α) (f: α →TimeM β) :TimeM β :=
  设 r := f m.ret
  ⟨r.ret, m.time + r.time⟩

实例 : 单子 TimeM 其中
  纯 := 纯
  绑定 := 绑定

/-- 将时间开销增加 'c'（默认值为 1），而不改变值。 -/
定义 tick (c: ℕ := 1) : TimeM PUnit := ⟨.unit, c⟩

宏 "✓ [" c:term "]" body:doElem : doElem => `(doElem| do TimeM.tick c; $body:doElem)
/-- '✓ x' 是 'pure x'，增加一个 tick。 -/
宏 "✓" body:doElem : doElem => `(doElem| ✓[1] $body)


-- Simp 引理
@[simp] 定理 ret_纯值 {α} (a : α) : (纯值 a : TimeM α).ret = a := rfl
@[simp]定理ret_bind{α β}(m:TimeM α)(f: α → TimeM β):
  (m >>= f).ret = (f m.ret).ret := rfl
@[simp] 定理 ret_tick (c: ℕ) : (tick c).ret = () := rfl
@[simp] 定理 time_bind {α β} (m : TimeM α) (f : α → TimeM β) :
  (m >>= f).时间 = m.时间 + (f m.ret).时间 := rfl
@[simp]定理time_pure{α}(a: α) :(purea:TimeM α).time = 0:=rfl
@[简化] 定理 时间_滴答 (c : ℕ) : (滴答 c).时间 = c := 反身性
-- 附加 Simp 引理，用于 <*与 *> 以及 <*>
(...)

结束 TimeM
```

图 3：用于基本算法复杂度分析的单子式 API。类型为 `TimeM` α的对象包含 `ret`（一个类型为 α的值）以及计算该值所产生的 `time` 代价。`bind` 运算将两个计算链接在一起，并将复合运算的时间代价定义为各个运算的时间代价的和。辅助函数 `tick` 用于指定时间代价。记号 ✓是调用 `tick`的语法糖。最后一节陈述了一些基本引理，用以帮助化简涉及 `TimeM`的表达式。

```
导入 ComplexityAPI
变量 {α :类型} [线性序 α]

定义 merge:     列表 α → 列表 α     →TimeM（列表 α）
  | [], ys => 返回 ys
  | xs, [] => 返回 xs
  | x::xs, y::ys' => do
    令 c := x ≤ y
    若 c 则
      let rest ← merge xs' (y::ys')
      返回 (x :: rest)
    else
      令 剩余 ←合并 (x::xs')          ys'
      返回 (y :: rest)

/-- 归并排序算法，返回一个 'TimeM (列表 α)'，其中时间表示总
    比较次数。 -/
定义 归并排序 (xs : 列表 α) : TimeM (列表 α) := do
  如果 xs.length < 2 那么 返回 xs
  else
    令 half := xs.length / 2
    令 L := xs.取 一半
    令 R := xs.drop half
    令 L' ←归并排序 L
    令 R' ←归并排序 R
    令 结果 ←      归并 L' R'
    返回 result

/-- 归并排序在功能上是正确的。 -/
定理归并排序_正确（xs : 列表 α）:
  已排序（归并排序 xs）。ret ∧ 列表置换（归并排序 xs）。ret xs      := 证明
  -- 证明省略

/-- 归并排序的时间复杂度，以比较次数计数。 -/
定理 归并排序_时间（xs : 列表 α）:
  令 n := xs.length
  (归并排序 xs).time ≤ n *clog2 n := 由
  -- 证明略
```

图 4：使用图 3 中的应用程序编程接口实现的归并排序。请注意，每次调用排序例程都会返回一个有序列表及其计算成本。有了这些定义，我们便可以正式陈述并证明定理，确立归并排序的正确性及其最坏情况下的比较次数。

计算；相比之下，我们旨在建立一个通用的复杂度分析框架 s.

我们所提出框架的精确定义还需要进一步完善，并将在后续不断演进。目前，我们在图 3 中展示了一个简单的单子式 API，类似于 Mathlib 的 Writer 单子，并受到此前关于函数式算法分析的工作的启发 [10, 12], ；该 API 将成为 CSLib 复杂度分析框架的一部分。该 API 将过程的返回值与计算这些值的成本打包在一起；其使用示例见图 4。将时间复杂度视为一种计算效应，可以使正确性与复杂度分析彼此正交。此外，通过让用户控制 tick 的放置位置，该框架支持对不同成本模型进行实验，从而实现多样化的复杂度分析。

与此同时，该 API 也存在多项局限。例如，它依赖于手动添加的 tick 注释，用户可能会不小心出错，而不是对执行成本进行自动验证。此外，它无法直接证明“没有任何算法能够以快于 $f(n)$ 的速度解决问题 X” 这类陈述。然而，由于它轻量便捷，并且能够证明许多关于算法复杂度的有趣定理，因此是我们开展工作的良好起点。从长远来看，我们计划采用更重量级的方法来补充这一方案，通过显式的 RAM 模型和查询模型对复杂度进行形式化。

2.2 支柱 2：为推理日常代码提供基础设施

CSLib 的第二大支柱将开发一套强大的全新基础设施，用于推理日常命令式代码。Lean 最近新增了对命令式程序定义 [42] 和验证[2] 的大力支持，我们计划在支柱一的工作中充分利用这些进展。本支柱所采用的方法与之互补，因为它们将基于 Lean 的形式化推理与 Rust、Python 和 C++等主流语言中的代码验证连接起来。

我们的方法建立在演绎验证技术和工具的深厚传统之上， [11, 17], 尤其是建立在中间验证语言（IVL）之上的系统。最广为人知的或许是 Boogie IVL， [26, 28], 许多系统都在使用它（例如 Dafny、 [27], Viper、 [21], CIVL、 [25], 以及 Move Prover [46]）。CSLib 的支柱 2 将基于一种名为 *Boole* 的新 IVL；它部分受到 Boogie 的启发，同时也充分利用了 Lean 的能力与优势。

图 5 展示了基于 CSLib 的代码推理的长期愿景。我们设想构建一个支持多种前端语言的框架。用户可以用代码编写程序，并附上 Lean 中的规格说明。随后，这些代码和规格说明会被转译（通过编译进行转换）为 Boole。我们计划构建一个以 Boole 为核心的演绎式验证平台，利用各种算法和后端生成 Lean 中的验证条件。然后，可以使用各种 Lean 自动化技术来证明这些条件。

这一愿景遵循若干面向未来的设计原则，旨在避免以往工作中暴露出的不足和陷阱：(i) Boole 应当看起来像伪代码，便于人类理解；(ii) Boole 及其生态系统将嵌入 Lean 中，转译步骤和验证条件生成步骤都应以尽量缩小可信计算基（TCB）为目标来实现；(iii) 验证条件应生成为 Lean 目标，便于人类直观阅读，并能轻松关联到其来源代码；(iv) 代码规格说明和验证条件应能够利用 CSLib 支柱 1 中定义的丰富形式化体系。

Boole 的验证条件生成将基于其他开源项目的基础设施和理念构建，例如 Loom [14] 和 Strata。Loom[3] 提供了在 Lean 中定义浅嵌入式语言的机制（包括一种名为 Velvet、类似 Dafny 的语言），并对正确的验证条件生成提供了强有力的保证。Strata[4] 是一个旨在促进创建领域特定语言的开源项目，称为

[2]https://lean-lang.org/doc/reference/latest/The--mvcgen--tactic/[3]
https://github.com/verse-lab/loom[4]https://github.com/strata-org/Strata

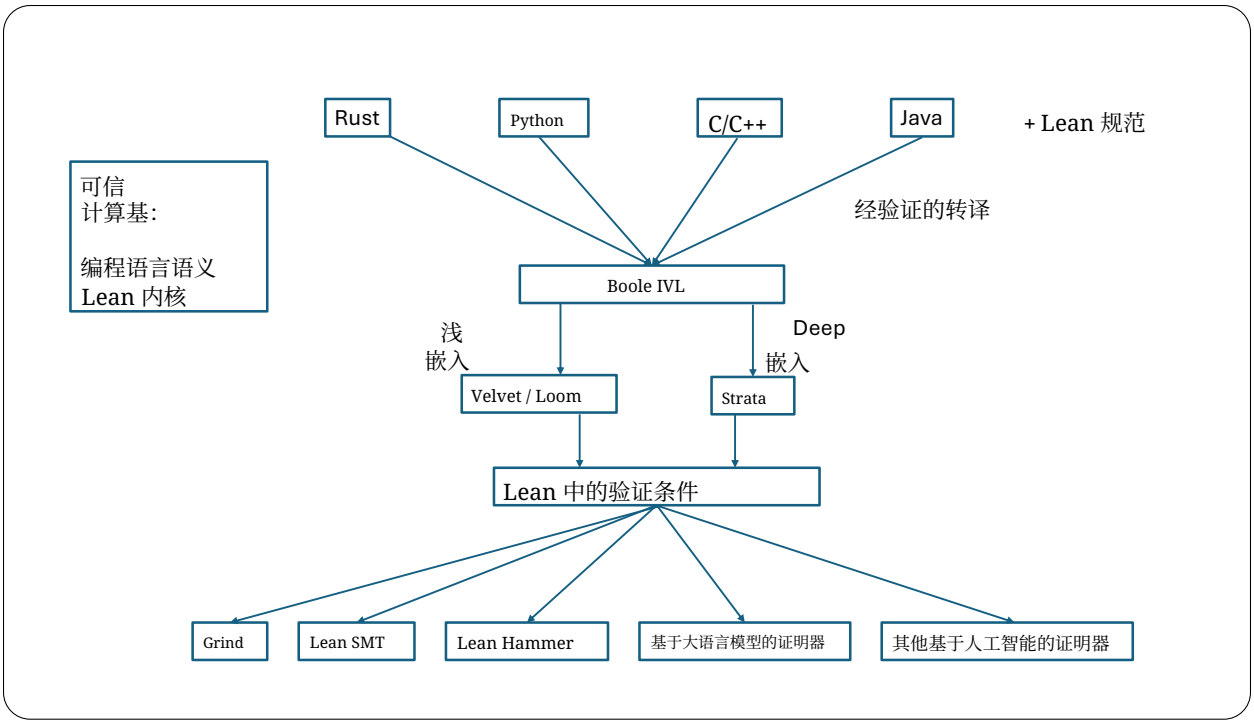


图 5：CSLib 代码推理愿景。

「方言」，这些语言被深度嵌入 Lean 中 .

图 6 展示了一段简单的 Lean 代码片段，其中包含一个用 Boole 编写的程序。Boole 是作为 Strata 的 Core 方言扩展实现的，而 Strata 在很大程度上仿照了经典的 Boogie IVL。在导入并打开相关软件包的前导代码之后，程序本身可以用易于阅读的形式编写。请注意 `spec` 关键字，它表示一项规范。在此例中，`requires` 关键字表明有一项要求（前置条件）： n 应为非负数。我们还可以使用 `invariant` 关键字指定循环不变式，也可以使用 `assert` 关键字指定更一般的程序不变式，这些不变式应在代码中的特定位置成立。

我们可以通过将规格说明和代码转换为验证条件来验证 这些断言。具体做法是为程序中的每个构造赋予 Lean 语义，然后遵循标准的演绎验证方法：对于程序中的某些无循环路径，证明沿该路径所有可能的执行中，该路径的前置条件都能确保其后置条件。示例中的`#prove_vcs` 命令会调用一个原型验证条件生成器，在 Lean 中生成目标。图 7 展示了一组生成的目标。例如，第一个目标表示一条从过程入口到循环开始处的路径。程序的前置条件要求$n \geq 0$，而循环条件必须为真，这意味着 $i < n$。这些条件应能推出循环不变式。我们的验证条件生成器会进行一些基本化简——在此例中，它知道进入循环时 $i = 0$，因此将 i 替换为 0 并进行化简。由此得到图中所示的目标。同样，第二个目标表示一个验证条件，说明循环不变式在一次迭代前后保持成立；另外两个目标则对应那两条断言。

Lean 中的目标可以手工证明，也可以通过调用自动证明器来证明；自动证明器是用于自动完成 Lean 目标证明的通用策略。图 5 底部展示了若干潜在的自动证明器。对于

```
导入 Strata.MetaVerifier

导入 Smt
导入 Smt.Rat

打开 Strata

定义 loopSimple : Strata.Program :=

#strata
程序 Boole;

过程 loopSimple (n: 整数) 返回 (r: 整数)
规范 {
  前置条件 (n  >= 0);
}
{
  变量 和 : 整数;
  变量 i : 整数;

  和 : = 0;
  i := 0;
  当 (i  < n)
    不变式 (i  <= n && ((i *(i-1)) div  2 == 和));
  {
    和 := 和  + i;
    i : = i  + 1;
  }
  断言  [和_断言]: ((n *(n-1)) div  2 == 和);
  断言  [否定_条件]: (i  == n);
  r : = 和;
};
#end

#prove_使用 loopSimple 证明 vcs
  所有_目标 （尝试 smt）
```

图 6：一个使用 Strata 的 Boole 方言编写的简单程序，用于计算前 n个正整数的和。

我们的示例，一种名为`smt` 的策略可以解决所有这些目标。该策略使用 Lean-SMT 工具 [30]将目标转换为 SMT 公式，调用求解器，获取证明证书，然后在 Lean 中逐步回放该证明。此类自动证明器的数量与能力一直在迅速增长，未来还将出现更强大的自动证明器，包括基于人工智能的自动证明器。

CSLib 的长期路线图将包括逐步扩展 Boole，以增加更多功能，包括代价语义（用于支持对计算复杂性的推理和定理证明）、支持使用 Mathlib 和 CSLib 支柱 1 中定义的规格说明、支持对并发进行推理，以及支持对指针等底层构造进行推理。我们还将考虑现代编程抽象和验证技术，例如编舞式编程 [31, 40] 和分离逻辑 [22, 35]研究中所采用的技术。

更为重要的是，我们还打算将 Boole 用作标准编程语言中真实系统验证的真正中间 语言。计划是在 Lean 中形式化真实编程语言的语义，利用这些语义将这些语言中的代码和规格说明翻译为 Boole 代码和规格说明，然后使用我们正在为 Boole 构建的验证工具来验证翻译后的代码。Aeneas 项目 [16], 将 Rust 的相当大子集翻译为若干证明

```
情况 入口_不变量_0
⊢ ∀ (n : 整数), 0 < n → n ≥ 0 → 0 ≤n ∧ 真

情况 任意_迭代_维持_不变量_0
⊢ ∀(n i sum : 整数) ,
  0 < n →
    i < n →
      我 ≤ 不 ∧ 我* (我 - 1) / 2 = 和 →
        n ≥ 0 → i + 1 ≤ n ∧ (i + 1) *(i + 1 - 1) / 2 = 和 + i

情形 和 _断言
⊢ ∀ (n i 和 i' sum' : 整数) ,
  n ≥ 0 →
    (若 0 < n 则 0 < n 否则 真) →
      (若 0 <n则i <n否则真) →
        (若 0 < n 则 i ≤ n ∧ i* (i - 1) / 2 = 和 否则 真) →
          (若 0 < n 则 ¬i' < n 否则 真) →
            (若 0 < n 则i' ≤ n ∧i' *(i' - 1) / 2 = 和' 否则 真) →
              (如果 如果 0 < n 那么 False 否则 True 那么 如果 0 < n 那么 False 否则 True
    否则 真) →
                n* (n - 1) / 2 = 如果 0 < n 那么 和' 否则 0

情况 负_条件
⊢ ∀ (n i 和 i' sum' : 整数) ,
  n ≥ 0 →
    (如果 0 < n 那么 0 < n 否则 真) →
      (如果 0 <n那么i <n否则True) →
        (如果 0 < n 那么i ≤ n ∧ i* (i - 1) / 2 = 和否则True) →
          (如果 0 < n 那么 ¬i' < n 否则True) →
            (如果 0 < n 那么i' ≤ n ∧i' *(i' - 1) / 2 = 和' 否则 真) →
              (如果如果 0 < n 那么 假 否则 真 那么如果 0 < n 那么 假 否则 真
    否则 真) →
                (如果 0 < n 那么 i' 否则 0) = n∀(n : 整数), 0 < n → n ≥ 0 → 0 ≤n ∧
     True
```

图 7：由图 6 中的程序生成的 Lean 验证条件。

助手（包括 Lean），为这样一条流水线提供了一种范例。

这一支柱的最后一个具体目标，是汇集一个大型代码库，其中的代码均已使用 CSLib 验证基础设施完成认证。我们计划免费开放该库，并将其与 VeriLib 等协同项目整合。[5]

3 人工智能的作用

CSLib 社区的这项工作受到人工智能代码生成持续变革的推动，原因有二。一方面，我们相信 CSLib 能成为我们共同应对人工智能风险的一部分：由人工智能驱动的恶意行为者可能破坏全球软件，而使用人工智能编程工具的工程师也可能在不知情的情况下编写出存在漏洞的代码。这些风险使严格的软件验证和形式化验证的代码生成比以往任何时候都更加重要，而 CSLib 将助力实现这一目标。另一方面，AlphaProof [19], Deepseek-Prover [37], 和 Aristotle [2] 等人工智能项目展示了人工智能解决高难度形式化证明任务的能力。然而，基于人工智能的证明器受到底层证明助手所提供的抽象

[5] https://verilib.org/

能力的限制。通过扩展 Lean 中可用的、与计算机科学相关的抽象范围，CSLib 也将大幅提升此类证明器在形式化验证场景中的能力。

在 CSLib 中完成的形式化将成为基于人工智能的形式化和定理证明系统的高质量训练数据 [19, 37, 29]。与此同时，我们预计，面向 Lean 证明的基于人工智能的工具将大幅加快 CSLib 的开发。理想的结果是形成一个“飞轮”：基于人工智能的证明工具和人类专家都能在生成新知识方面不断提高效率。

在 CSLib 这样的复杂开发工作中使用人工智能，一个风险是人工智能生成的形式化可能存在缺陷。此外，尽管针对形式化陈述的人工智能生成的 Lean 证明可能是可靠的，但它们未必能够被人类理解。我们将通过确保人工智能工具在 CSLib 的开发中仅发挥辅助作用，并确保所有提交至 CSLib 的形式化和证明都须经过基于 GitHub 的人工代码审查流程，来降低这些风险。

4 构建 CSLib 社区

一个库的影响力取决于其用户群体。尽管形式化验证是计算机科学中一个充满活力的分支领域，我们希望 CSLib 的影响力能够超越这一研究领域。我们希望各类计算机科学研究人员——从理论研究者到系统构建者——都能在日常工作中使用 CSLib。我们希望 CSLib 能大幅降低形式化验证的成本，使众多行业的工程师愿意采用形式化技术。我们希望 CSLib 能在广泛的大学中得到应用，用于教授计算机科学的数学基础。

幸运的是，我们可以借鉴 Mathlib [3]，以此作为实现目标的蓝图。Mathlib 社区证明，开放讨论和社区活动对于让我们这样的大规模形式化项目取得成功至关重要。我们效仿他们，在 Lean Zulip 聊天上建立了 CSLib 频道 [6]。该频道已经相当活跃，我们鼓励所有对这个项目感兴趣的人加入。我们也请大家留意接下来几个月我们发布的、针对 CSLib 各个特定组件的贡献征集。未来几个月乃至几年，我们还计划举办许多关于 CSLib 的研讨会和教程。如果你有兴趣参与或主持此类活动，请与我们联系！

另一个方面是采用严格的代码审查实践，并创建能够降低 CSLib 开发和维护成本的工具。尤其是，过去几年间，用于自动形式化和形式化定理证明的人工智能工具取得了飞速进展。正如第 3 节所述，CSLib 项目的目标之一是充分利用并改进这类工具。我们预计，这些工具将大幅降低开发和使用 CSLib 的门槛。

依赖类型语言历来与主流命令式语言分属两个世界，这也是它们普及有限的一个关键原因。CSLib 的支柱 2 旨在改变这一现状。将 Rust 和 C++ 等主流语言翻译为 Boole，是实现这一愿景的关键。我们鼓励社区成员迎接挑战，着手构建这些翻译工具。

最后，我们将优先为 CSLib 构建高质量、可搜索的文档。这些文档将包括专为计算机科学不同领域的研究人员编写的教程，以及入门级教科书。如果您有意牵头编写或参与编写此类教科书，请联系我们。

[6] https://leanprover.zulipchat.com/#narrow/channel/513188-CSLib

5 治理模式

CSLib 采用双机构结构进行治理，旨在平衡战略方向与技术执行。指导委员会由学术界和工业界的领军人士组成，负责争取资金支持并指导项目的整体愿景。委员会的创始成员包括 Clark Barrett（斯坦福大学和亚马逊云科技）、Swarat Chaudhuri（Google DeepMind 和德克萨斯大学奥斯汀分校）、Jim Grundy（亚马逊云科技）、Pushmeet Kohli（Google DeepMind）、Fabrizio Montesi（南丹麦大学）和 Leonardo de Moura（Lean FRO 和亚马逊云科技）。维护者团队与委员会协同工作，负责管理代码库的技术方向、质量标准和日常开发。截至本文发表之日，团队由首席维护者 Fabrizio Montesi 领导，负责协调整体工作；团队还包括负责长期跨领域开发的技术负责人 Alexandre Rademaker 和 Sorrachai Yingchareonthawornchai，以及负责特定领域的领域维护者 Chris Henson 和 Kim Morrison，其负责的领域包括 λ-演算、元编程和 CI/CD 基础设施。

随着社区的发展，维护者团队和指导委员会自然也会有所变化。具体而言，我们计划根据贡献绩效（尤其是贡献和审查活动）以及项目需求，定期邀请新的维护者。我们的目标是通过指导委员会的指导确保战略方向的一致性，并凭借维护者团队的专业知识实现技术卓越，同时营造一个欢迎广大贡献者参与的环境。

6 路线图

现在，我们简要介绍一下 CSLib 项目在 2026 和 2027 年的发展规划。鉴于项目目前成立仅数月，且我们计划根据社区意见进一步完善项目范围，这份路线图暂定。更详细且定期修订的路线图可在 CSLib网站上查看：https://cslib.io。

2026. CSLib 代码库已包含操作语义、程序等价性、若干自动机模型、一种线性逻辑，以及一些基础排序和搜索算法的初步 Lean 形式化。我们预计，到 2026 年底，CSLib 将涵盖典型本科算法与数据结构课程中许多算法的形式化，以及典型本科计算理论课程中涉及的大多数模型和逻辑。此外，社区成员也可贡献其研究专门领域中的形式化成果。请注意，这项支柱一任务的进展不取决于 Booleto 是否准备就绪。

关于支柱 2，我们已经基于 Strata框架开发了 Boole 的初始版本。预计在未来一年，框架将大幅成熟。到 2026 年底，Boole框架将具备强大的能力，能够生成 Lean语言验证条件，并与基于SMT的自动证明器交互。届时，CSLib 代码库将包含大量基本算法和数据结构的 Boole语言表示，以及所有相关验证条件的 Lean规范和证明。

2027年。 项目的第二年将重点关注规模化与统一化。具体而言，我们的支柱一工作将转向复杂性理论、并发、安全编译以及随机化算法和量子算法等技术上具有挑战性的主题。我们预计，到2027年年底，CSLib将为计算机科学领域至少一项重要的基础性发现提供实质性助力。在支柱二工作中，我们将

扩展 Boole，以支持分离逻辑等机制；这些机制对于验证低级系统代码是必需的。我们还将广泛使用支柱一机制，编写更高级的系统规格说明。我们预计，到2027年年底，CSLib将实现至少一个规模相当大的现实世界系统的端到端验证。

致谢

我们感谢 CSLib[7] 所有现任及未来的贡献者为本项目所做的工作，并感谢 Eric Wieser 和 Tom Kalil 对本论文提出的深思熟虑的反馈。

参考文献

[1] Lean 数学库。*CoRR*，abs/1910.09336，2019年。

[2] T. Achim、A. Best、A. Bietti、K. Der、M. F´ed´erico、S. Gukov、D. Halpern-Leistner、K. Henningsgard、Y. Kudryashov、A. Meiburg、M. Michelsen、R. Patterson、E. Rodriguez、L. Scharff、V. S hanker、V. Sicca、H. Sowrirajan、A. Swope、M. Tamas、V.Tenev、J. Thomm、H. Williams 与 L. Wu。Aristotle：国际数学奥林匹克级自动定理证明。*arXiv* 预印本 *arXiv:2510.01346*，2025年。

[3] A. Baanen、M. R. Ballard、J. Commelin、B. G. ge Chen、M. Rothgang 和 D. Testa。《发展 Mathlib：大型数学库的维护》，2025年。[4] R. S. Boyer 和 J. S. Moore。证明检查、定理证明与程序验证。技术报告，计算科学研究所，德克萨斯大学奥斯汀分校，1983年。[5] J. Brown-Cohen、G. Irving 和 G. Piliouras。通过双重高效辩论实现可扩展的人工智能安全。*arXiv* 预印本 *arXiv:2311.14125*，2023年。[6] H. Chan 和 M. Norrish。AKS 算法的机械化。《自动推理杂志》，65(2)：205–256，2021年。[7] A. Chargu´eraud 和 F. Pottier。使用带有时间信用的分离逻辑，验证并查集实现的正确性和摊销复杂度。《自动推理杂志》，62(3)：331–365，2019年。[8] G. Chehaibar、H. Garavel、L. Mounier、N. Tawbi 和 F. Zulian。PowerScale™ 总线仲裁协议的规范与验证：一项 LOTOS 工业实验。载于形式描述技术 *IX*：理论、应用与工具，第435–450页。施普林格，1996年。[9] E. M. Clarke。模型检验。载于软件技术与理论计算机科学基础国际会议，第54–56页。施普林格，1997年。[10] N. A. Danielsson。纯函数式数据结构的轻量级半形式化时间复杂度分析。载于*POPL*，第133–144页。ACM，2008年。[11] R. W. Floyd。为程序赋予含义。载于计算机科学的数学方面：应用数学专题讨论会论文集，第19–32页。美国数学学会，1967年。

[7]当前 CSLib 贡献者名单请见 https://github.com/leanprover/cslib/graphs/ contributors。

[12] J. Gibbons 和 R. Hinze。《Just do it：简单的单子等式推理》。载于 *ICFP*，第 2–14 页。ACM，2011 年。[13] A. Girard 和 G. J. Pappas。《近似双模拟：计算机科学与控制理论之间的桥梁》。《style id='5'>欧洲控制杂志，17(5-6):568–578，2011 年。[14] V. Gladshtein、G. Pˆ ırlea、Q. Zhao、V. Kurin 和 I. Sergey。《基础性多模态程序验证器》。《style id='8'>ACM程序设计语言会议录，第 10 卷，2026 年。[15] S. Gulwani、K. K. Mehra 和 T. Chilimbi。《Speed：程序计算复杂度的精确高效静态估算》。《style id='11'>《ACM SIGPLAN 通知》，44(1):127–139，2009 年。[16] S. Ho 和 J. Protzenko。《Aeneas：通过函数式翻译验证 Rust》。《style id='14'>《ACM编程语言会议录》，6(ICFP):711–741，2022 年。[17] C. A. R. Hoare。《计算机编程的公理基础》。《style id='17'>《ACM通讯》，12(10):576–580，1969 年。[18] T. Hoare。《验证编译器：计算研究面临的一项重大挑战》。《style id='20'>《ACM杂志》(JACM)，50(1):63–69，2003 年。ˇ[19] T. Hubert、R. Mehta、L. Sartran、M. Z. Horv´ath、G. Zuˇzi´c、E. Wieser、A. Huang、J. Schrittwieser、Y. Schroecker、H. Masoom 等。《利用强化学习进行奥林匹克级形式化数学推理》。《style id='25'>《自然》，第 1–3 页，2025 年。[20] G. Irving。《辩论：辩论协议的 Lean 形式化》。https://github.com/girving/ debate，2025 年。[21] U. Juhasz、I. T. Kassios、P. M¨uller、M. Novacek、M. Schwerhoff 和 A. J. Summers。《Viper：用于基于权限推理的验证基础设施》。技术报告，苏黎世联邦理工学院，2014 年。[22] R. Jung、R. Krebbers、J. Jourdan、A. Bizjak、L. Birkedal 和 D. Dreyer。《Iris 从基础构建：高阶并发分离逻辑的模块化基础》。《style id='31'>《函数式编程杂志》，28:e20，2018 年。[23] G. Klein、K. Elphinstone、G. Heiser、J. Andronick、D. Cock、P. Derrin、D. Elkaduwe、K. Engelhardt、R. Kolanski、M. Norrish 等。《seL4：操作系统内核的形式化验证》。载于第*22*届*ACM SIGOPS*操作系统原理研讨会，第 207–220 页，2009 年。[24] A. Kontorovich。《数学的未来形态》。《style id='37'>arXiv 预印本 arXiv:2510.15924，2025 年。[25] B. Kragl 和 S. Qadeer。《Civl 验证器》。载于计算机辅助设计中的形式化方法，*FMCAD 2021*，美国康涅狄格州纽黑文，*2021* 年 *10* 月 *19–22* 日，第 143–152 页。IEEE，2021 年。[26] K. R. M. Leino。《这就是 Boogie 2》，2008 年。手稿 KRML 178。[27] K. R. M. Leino。《Dafny：用于函数正确性的自动程序验证器》。载于国际编程逻辑、人工智能与推理会议，第 348–370 页。施普林格，2010 年。[28] K. R. M. Leino 和 P. R¨ummer。《多态中间验证语言：设计与逻辑编码》。载于 J. Esparza 和 R. Majumdar 编，*TACAS*，第 312–327 页，柏林、海德堡，2010 年。施普林格·柏林·海德堡。

[29] Y. Lin, S. Tang, B. Lyu, J. Wu, H. Lin, K. Yang, J. Li, M. Xia, D. Chen, S. Arora 等。Goedel-prover：用于开源自动定理证明的前沿模型。*arXiv* 预印本 *arXiv:2502.07640*，2025年。[30] A. Mohamed, T. Mascarenhas, H. Khan, H. Barbosa, A. Reynolds, Y. Qian, C. Tinelli 和 C. Barrett。LEAN-SMT：一种用于完成 Lean 中证明目标的 SMT 策略。载于 R. Piskac 和 Z. Rakamari´c 编，《37th 国际计算机辅助验证会议（*CAV '25*）论文集，第15931卷《计算机科学讲义》，第197–212页。施普林格，2025年7月。克罗地亚，萨格勒布。[31] F. Montesi。《编舞导论》。剑桥大学出版社，2023年。[32] L. d. Moura 和 S. Ullrich。Lean 4 定理证明器与编程语言。载于《国际自动演绎会议》，第625–635页。施普林格，2021年。[33] T. Nipkow, J. C. Blanchette, M. Eberl, A. G´omez-Londo˜no, P. Lammich, C. Sternagel, S. Wimmer 和 B. Zhan 编。《函数式数据结构与算法：证明助手方法》。ACM图书，2025年。[34] T. Nipkow, M. Wenzel 和 L. C. Paulson。《Isabelle/HOL：高阶逻辑证明助手》。施普林格，2002年。[35] P. O'Hearn。分离逻辑。《*ACM*通讯》，62(2):86–95，2019年1月。[36] N. Perry, M. Srivastava, D. Kumar 和 D. Boneh。使用人工智能助手会让用户编写更多不安全代码吗？载于《*2023*年*ACM SIGSAC*计算机与通信安全会议论文集，第2785–2799页，2023年。[37] Z. Ren, Z. Shao, J. Song, H. Xin, H. Wang, W. Zhao, L. Zhang, Z. Fu, Q. Zhu, D. Yang 等。Deepseek-prover-v2：通过强化学习进行子目标分解，推进形式化数学推理。*arXiv* 预印本 *arXiv:2504.21801*，2025年。[38] T. Ringer。证明与对话。《美国数学会通告》，71(9):1179–1182，2024年。[39] M. Rosendahl。自动复杂度分析。载于《第四届函数式编程语言与计算机体系结构国际会议论文集》，第144–156页，1989年。[40] R. Rubbens, P. van den Bos 和 M. Huisman。Veymont：基于编舞生成使用共享内存的正确并发程序。载于 N. Kosmatov 和 L. Kov´acs 编，《集成形式化方法——第*19*届国际会议（*IFM 2024*），英国曼彻斯特，*2024*年*11*月*13*日至*15*日，会议论文集，第15234卷《计算机科学讲义》，第217–236页。施普林格，2024年。[41] T. Tao。机器辅助证明。《美国数学会通告》，即将出版，2024年。[42] S. Ullrich 和 L. de Moura。"do"解链：在纯函数式语言中拥抱局部命令式特性（函数式珍珠）。《*ACM*编程语言会议录》，6(ICFP):512–539，2022年。[43] G. Winskel 和 M. Nielsen。《并发模型》，第4卷，第1–148页。牛津大学出版社，美国，1995年。

[44] K. Yang, G. Poesia, J. He, W. Li, K. E. Lauter, S. Chaudhuri 和 D. Song。立场论文：形式化数学推理——人工智能的新前沿。载于第四十二届国际机器学习会议立场论文专题，2025年。[45] A. Zhang, R. McAllister, R. Calandra, Y. Gal 和 S. Levine。无需重构的强化学习不变表示学习。*ICLR*，2021年。[46] J. E. Zhong, K. Cheang, S. Qadeer, W. Grieskamp, S. Blackshear, J. Park, Y. Zohar, C. Barrett 和 D. L. Dill。Move Prover。载于 S. K. Lahiri 和 C. Wang 编，《32nd 国际计算机辅助验证会议（*CAV '20*）论文集，第12224卷《计算机科学讲义》，第137–150页。施普林格国际出版公司，2020年7月。