计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
63(5): 1117−1138, 2026

Isabelle 驱动的搜索树类算法建模及自动化验证

左正康[1,2,3] 朱小影[1] 游 珍[1] 王昌晶[1,2,3]

[1](江西师范大学人工智能学院 南昌 330022)

[2](高性能计算江西省重点实验室(江西师范大学) 南昌 330022)

[3](语言智能研究中心(江西师范大学) 南昌 330022)

(zuo803@jxnu.edu.cn)

Isabelle-Driven Modeling and Automated Verification of Search Tree Algorithms

Zuo Zhengkang[1,2,3], Zhu Xiaoying[1], You Zhen[1], and Wang Changjing[1,2,3]

[1](*School of Artificial Intelligence*, *Jiangxi Normal University*, *Nanchang* 330022)

[2](*Jiangxi Provincial Key Laboratory for High Performance Computing* (*Jiangxi Normal University*), *Nanchang* 330022)

[3](*Language Intelligence Research Center* (*Jiangxi Normal University*), *Nanchang* 330022)

Abstract Search tree algorithms are widely used in areas such as database indexing, compiler design, and file systems, where the functional correctness of these algorithms is a critical factor in ensuring software reliability. However, traditional development process faces two major challenges: first, algorithm design often lacks rigorous mathematical modeling, causing correctness verification to remain at an empirical level; second, existing verification approaches heavily rely on manual or interactive theorem proving, making the verification complexity grow exponentially with increasing algorithm scale. To address the above issues, an Isabelle-driven method for modeling and automated verification of search tree algorithms is proposed. This method builds on a formally verified functional modeling library to construct a modular modeling strategy that supports various search tree structures, and introduces a formal specification-driven framework for automatically synthesizing algorithmic code. Developers are required to supply only minimal functional expressions to complete the implementation. In addition, a reusable lemma library has been developed to support an automated block-wise verification strategy, significantly reducing verification costs. Furthermore, a method for generating functional executable programs is proposed. Experimental evaluation using a representative case study demonstrates that the proposed method substantially reduces verification costs. Compared with interactive verification approaches, the average reductions in code size and manual effort are 88.60% and 61.51%, respectively; compared with generative AI plus interactive verification methods, the average reductions in code size and manual effort are 88.09% and 41.76%, respectively. Overall, the proposed framework provides a systematic and highly automated solution for the formal verification of complex data structure algorithms in specific domains, while also offering a feasible pathway and methodological support for generating correct functional executable programs.

Key words search tree; Isabelle theorem prover; functional modeling; automated verification; program generation

收稿日期: 2025-12-15; **修回日期**: 2026-03-05

基金项目: 国家自然科学基金项目 (62462036,62462037); 江西省自然科学基金重点项目 (20242BAB26017); 江西省主要学科学术与技术带头人培养项目 (20232BCJ22013)

This work was supported by the National Natural Science Foundation of China (62462036, 62462037), the Key Project of Jiangxi Provincial Natural Science Foundation (20242BAB26017), and the Cultivation Project for Academic and Technical Leader in Major Disciplines in Jiangxi Province of China (20232BCJ22013).

通信作者: 王昌晶(wcj@jxnu.edu.cn)

摘 要 搜索树类算法广泛应用于数据库索引、编译器设计、文件系统等领域，其功能正确性验证是保障软件可靠性的关键环节。然而，传统开发流程存在2个主要问题：首先，算法设计常缺乏严格的数学建模，导致正确性验证停留于经验层面；其次，现有验证方法多依赖手工证明或交互式定理证明工具，随着算法规模的扩大，验证复杂度呈指数级增长。为解决上述问题，提出了Isabelle驱动的搜索树类算法建模及自动化验证方法。该方法基于经过形式化验证的函数式建模库，构建了涵盖搜索树类结构的模块化建模策略，并设计了形式化规约自动合成算法代码框架。开发者仅需补充少量函数表达式，即可实现完整的算法。此外，开发了可复用的验证引理库，并通过自动化分块验证策略有效降低了验证成本。进一步地，提出了函数式可执行程序生成方法。通过典型案例实验验证，结果表明所提方法显著降低了验证成本。与交互式验证方法相比，代码行数和人工占比分别平均减少了88.60%、61.51%；与“生成式AI+交互式验证方法”相比，代码行数和人工占比分别平均减少了88.09%、41.76%。该框架为特定领域复杂数据结构算法的形式化验证提供了系统性且高度自动化的解决方案，同时为生成正确的函数式可执行程序提供了可行路径与方法保障。

关键词 搜索树；Isabelle定理证明器；函数式建模；自动化验证；程序生成

中图法分类号 TP311

DOI: 10.7544/issn1000-1239.202550899 **CSTR:** 32373.14.issn1000-1239.202550899

搜索树类算法作为组织与管理程序状态、符号信息及结构化数据的关键技术，广泛应用于数据库索引、编译器设计、文件系统等领域，确保这些算法的功能正确性成为系统安全性和稳定性的关键保障[1]，任何细微的错误都可能导致数据错误、性能下降，甚至引发安全漏洞和系统崩溃。然而传统的搜索树类算法开发流程中存在2个主要问题：首先，算法设计常缺乏严格的数学建模，导致正确性验证停留于经验层面；其次，现有验证方法多依赖人工定理证明或交互式定理证明工具，导致验证复杂度随算法规模呈指数增长。

在算法设计方面，搜索树作为索引、调度、缓存与事务管理等核心模块的基础数据结构，其相关算法通常需要处理复杂的结构变动，并严格依赖有序性与平衡性等不变式约束。若缺乏形式化规约，这些隐含约束往往难以被显式刻画与验证，从而在迭代或组合过程中造成结构完整性的破坏，例如，当开发者未能通过广义代数数据类型对红黑树的颜色约束进行编译期验证[2]，可能引发违反平衡条件的运行时错误。与此同时，在实际开发中，开发者可能为了追求开发速度，忽视严格的类型检查和函数纯度，导致程序中引入隐蔽的逻辑错误或未处理的异常情况[3]。例如在递归函数设计中未能确保递归调用的终止条件，可能导致无限递归，最终导致栈溢出错误[4]，在核心建模时忽视行为回溯会导致程序在不同分支间发生状态污染[5]。搜索树类算法设计若仅依赖经验性规则而非数学建模，程序异常可能通过算法层级传递，最终在系统集成阶段表现为逻辑崩溃，造成严重后果，甚至引发灾难性事件。在算法验证方面，目前形式化验证方法，如使用Coq[6]、Isabelle/HOL[7-8]等定理证明器对算法进行机械化验证，在理论上能够提供严格的正确性保证，但在实际应用中存在以下局限：验证过程通常针对具体的算法程序进行交互式机械化验证，导致验证的可复用性较低，当验证对象或应用场景发生变化时，需要重新设计验证方案；其次，现有验证方法自动化程度低，验证过程需要大量的人为干预，耗费大量时间和人力成本。这些因素限制了形式化验证方法在工业界的广泛应用，阻碍了函数式可执行程序的大规模产业化。

为解决上述问题，本文在不同的搜索树结构找到其中的共性，制定高效的建模及验证方法，在快速开发程序的同时，确保可执行程序的正确性和健壮性。进一步，建立函数式验证框架，构建统一的验证模式，引入自动验证技术，以在保证验证准确性的前提下，尽可能地减少人为参与，降低验证成本。并选取Haskell为示例语言，其适用于可靠性要求高的系统开发。结合程序生成，可将经形式化验证的算法转为Haskell可执行程序，从而推动函数式程序工业化应用。本文的组织结构图如图1所示，其贡献点总结为3个方面：

1)利用经过形式化验证的函数式建模库，构建了一个包含搜索树类结构的模块化建模策略，并设计了自动合成形式化规约算法的代码框架。开发者只需补充少量的函数表达式，即可完成算法的完整实现。

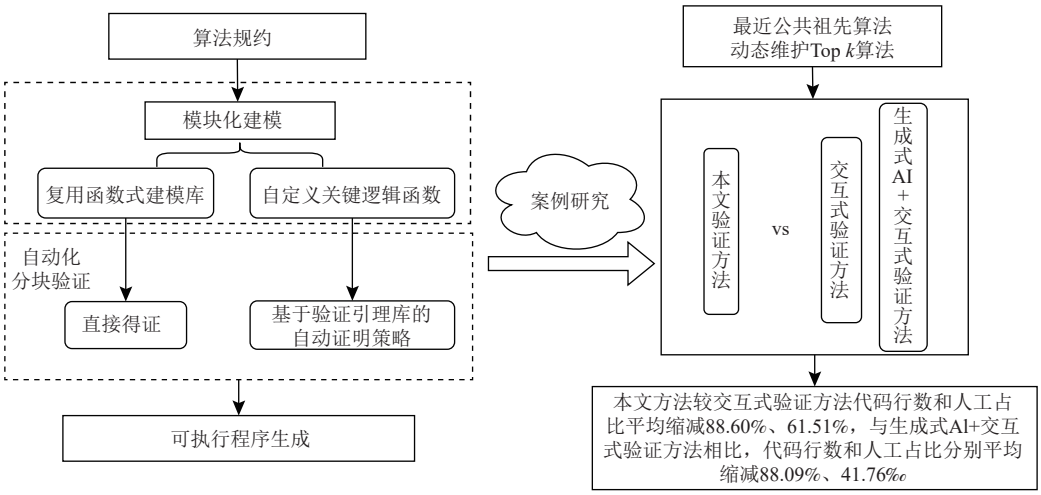


Fig. 1 Organizational structure diagram

图 1 组织结构图

2)开发了可复用的验证引理库,并通过自动化分块验证策略使形式化验证流程更加高效,提高了验证过程的模块化的同时,还将平均验证成本显著降低。

3)提出了函数式可执行程序生成机制,将建模、验证方法与 Isabelle 提供的代码生成机制相结合。通过在建模阶段规约精确定义算法行为,并在验证阶段确保功能与结构正确性,最终生成的可执行程序能够与最初的规约保持严格一致,从而保证代码的正确性与可信度。

1 相关工作

本文主要聚焦搜索树类结构算法建模及自动化验证研究,并与大语言模型验证进行实验比较。因此,本节将从搜索树类结构建模与验证框架、大语言模型验证 2 个方面进行相关工作比较。

1)搜索树类结构建模与验证框架。为突破单一算法验证模式的局限性,近年来研究逐渐转向通用验证框架的设计,旨在通过统一的方法支持多样化搜索树类结构的建模与验证。文献 [9] 基于函数式建模框架,给出了插入和删除的高阶泛化函数,对 LLRB 进行实例化建模,并对其附加性质进行细化,给出具体的验证方案。文献 [10] 提出了针对动态顺序统计树类结构的函数式建模和自动化验证框架,支持动态集合、顺序统计和搜索树等常用操作,构建完善的验证引理库并详细阐述具体的验证方法。上述文献针对具体类型的搜索树结构,其适用范围相对有限,较难在不同数据结构与算法间实现通用化迁移。

文献 [11] 提出用于验证 8 种搜索树类结构功能正确性的验证框架,通过中序遍历将树映射为列表,自动验证查找、插入以及删除操作的功能正确性,并严格证明了操作执行前后的搜索树属性保持。上述工作在构建通用验证框架方面取得了重要进展,但其构建的引理库主要集中于列表辅助函数,验证范围局限于搜索树的基本增加、删除和查找操作,对搜索树结构的实际算法应用尚未覆盖,且该研究未涉及实际算法的建模、验证与可执行程序生成的完整流程,具体对比见表 1。

Table 1 Comparison of Ref. [11] and Our Method

表 1 文献 [11] 与本文方法对比

对比项	文献 [11]	本文方法
函数式建模库		结构与操作建模库
验证引理库	列表辅助函数引理集	搜索树类功能和结构验证引理集
验证维度	底层基本操作	实际算法应用
目标	建模—验证	建模—验证—程序生成一体化

2)大语言模型验证。大语言模型辅助验证已成为提升形式化验证自动化能力的重要研究方向。随着大语言模型在数学推理领域的能力不断增强[12-14],相关研究正尝试将其能力系统地融入定理证明流程。文献 [15] 提出自玩定理证明框架,通过协同优化猜想生成与定理证明,有效克服了训练数据稀缺问题。DeepTheorem[16] 构建了专用于自然语言定理证明的框架,包含 12.1 万条国际数学奥林匹克(IMO)级别定理的数据集,并采用 RL-Zero 强化学习方法,直接激励模型进行数学推理。DeepSeek-Prover[17] 通过大

规模自动形式化方法生成 800 万条基于 Lean 4 的定理证明数据，结合质量过滤与迭代训练策略，在多个基准测试上显著超越现有方法。上述方法在形式化数学领域取得了显著进展，但在处理需要深层次、多步骤推理以及高度抽象的问题时仍存在一定局限性，特别是在包括搜索树等数据结构算法的泛化能力方面仍有待提升。SYNVER 是一种面向 C 程序的综合与验证框架，在大语言模型生成程序的过程中施加语法与语义偏置，从而使生成的程序更易于自动化验证[18]。但该方法目前尚难以应对复杂算法设计中数据结构维持不变量的验证任务。

2 搜索树类结构 Isabelle 函数式建模库与验证引理库

为简化算法设计和验证及提高代码复用性，本文构建了基于 Isabelle/HOL 的搜索树类结构函数式建模库与验证引理库，涵盖二叉搜索树、AVL 树、红黑树及变体、动态顺序统计树类、区间树以及简单多叉搜索树等典型结构，旨在为算法设计提供建模基础并降低验证复杂度。本节围绕搜索树类结构的函数式定义展开，设计了统一的建模策略。对于搜索树类结构对应的通用操作，开发了通用的操作模板，且这些操作已通过形式化方法严格验证其正确性。

2.1 函数式建模库

函数式建模库的核心目标是通过严格的数学抽象，将搜索树类结构的语义约束与操作逻辑转换为可机械验证的形式化模型。本库基于 Isabelle/HOL 定理证明器，构建了结构定义层与操作模板层的双层建模体系：前者通过高阶逻辑精确刻画搜索树结构的本质特征，后者则提供已验证的通用操作模板，为后续代码生成与验证奠定基础。

2.1.1 搜索树类结构的函数式建模

函数式建模库基于 Isabelle/HOL 的代数数据类型系统，通过递归定义和参数化设计，实现搜索树类结构的统一建模与灵活扩展。通过多态类型参数与高阶逻辑约束，既支持搜索树结构的严格数学建模，又能快速实例化为特定领域结构，以适应不同的应用需求。

搜索树类的基础模型采用递归定义，搜索树的性质主要体现为每个节点的左子树所有键值小于该节点键值，右子树所有键值大于该节点键值，这一结构约束将在第 2.2.2 节结构正确性中进一步详细说明。其类型参数与结构约束定义为：

```
datatype 'a treestr =
Leaf | Node [info] "'a treestr" 'o "'a treestr"
```

其中，Leaf 代表叶子节点（即节点为空），Node 代表非叶子节点（即节点非空），*info* 为可选变量，在节点中存储额外的信息以支持特定操作，普通搜索树无需该信息位，例如顺序统计树类可通过该信息位记录子树规模。参数'a 和'o 为多态类型参数，'a 是数据结构的类型参数，'o 代表节点存储的信息类型，'o 可以泛化为'a（单一数据）以及'a*extra（键值对）等情形，根据需求实例化为不同的具体类型。

普通的二叉搜索树节点值无附加性质，可根据搜索树的基础模型将 treestr 实例化为 $\text{treestr}_{\text{bst}}$。针对需含有其他附加信息的场景，如 AVL 树的高度信息、红黑树的颜色信息等，则利用参数封装复杂的数据类型，可将 treestr 分别实例化为 $\text{treestr}_{\text{avl}}$、$\text{treestr}_{\text{rbt}}$。对于多叉搜索树结构，可通过扩展搜索树类模型的参数以支持多路分支。搜索树模型实例化如图 2 所示。

2.1.2 搜索树类函数式通用操作模板

搜索树类通用操作模板旨在通过预先设计好的代码框架，让开发者在保持形式化严谨性的前提下，同时快速实现不同变种搜索树结构的核心功能。通用操作模板不仅支持跨结构的实例化与扩展，还避免重复编写和验证基础逻辑，从而简化开发流程。

对于不涉及外部输入参数的操作，其核心只需关注 2 类情形：空树情形，即如何处理叶节点；节点情形，即如何将左右子树的计算结果与当前节点值或结构组合，整个过程自顶向下、逐层递归，每一步都依赖于对其子结构先前计算结果的调用，最终得出全树的整体结果。搜索树类结构中涉及输入参数的一系列操作基于比较函数（*cmp*）实现，比较元素必须满足可比较性和对称性，进而构建搜索树类结构相关操作的高阶复合函数：

```
fun <opname> :: " 'a treestr ⇒ <inputtype> ⇒
<outputtype> " where
"<opname> Leaf x = [adjustfuncLeaf] <endLeaf >"
"<opname> (Node l a r) x = [pre-operation]
    case <cmpoperation> of
    LT ⇒ <[adjustfuncLT] endLT> |
    EQ ⇒ <[adjustfuncEQ] endEQ> |
    GT ⇒ <[adjustfuncGT] endGT>"
```

其中，[] 表示可选部分，<>表示实现功能需要补充的逻辑部分，opname 是具体操作名称，决定函数的核心行为，'a treestr 是泛化的搜索树结构，节点值类

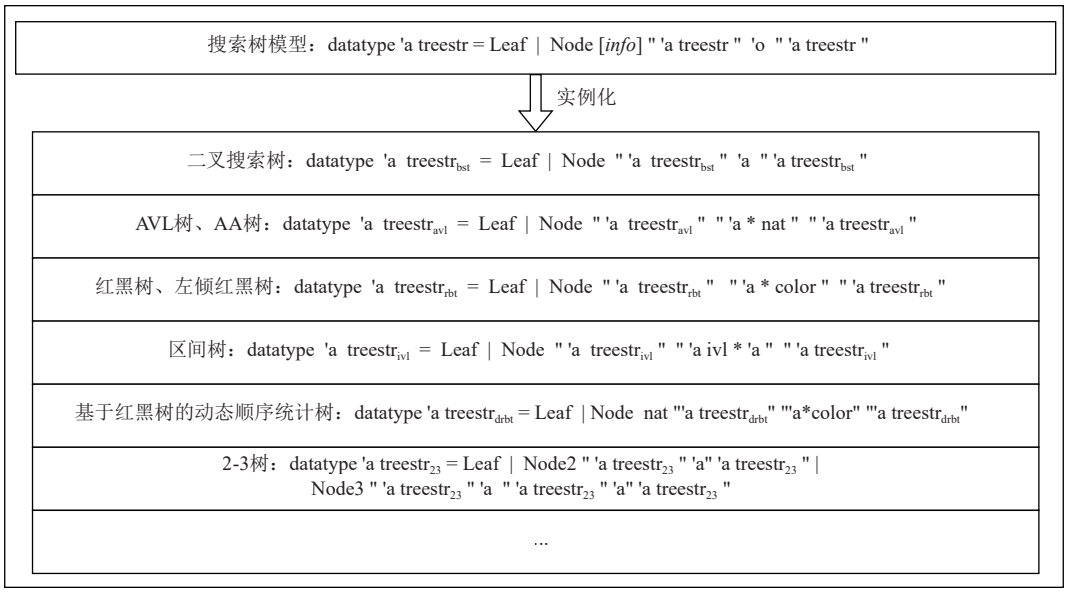


Fig. 2 Instantiation of the search tree model

图 2 搜索树模型实例化

型为'a, 其对应的元素需满足全序关系, inputtype 和 outputtype 分别表示操作所需的输入参数类型和操作结果类型。在函数体中, 对于树为空的情况, 根据函数功能返回最终结果(endLeaf), 如需处理可调用调整函数(adjustfuncLeaf)。对于树非空的情况, 可进行预处理操作(pre-operation)后再调用比较函数判断目标节点与当前节点的大小关系。若比较结果为小于(LT)时, 则对当前节点的左子树进行函数的递归操作, 在左子树中找到目标节点后返回最终结果(endLT)。若结果为等于(EQ), 则说明当前节点是需要处理的目标节点, 可直接对当前节点执行相关操作, 返回最终结果(endEQ)。若比较结果为大于(GT), 则对当前节点的右子树进行函数的递归操作, 在右子树中找到目标节点返回结果(endGT)。

通过模板参数化的形式, 我们只需提供少量元素即可快速生成类型安全的操作函数。这一过程自然地继承了高阶复合函数的模块化特性, 使得对其他需要输入参数的操作也能同样以少量替换完成实例化。

区域(locale)是 Isabelle/HOL 中提供的一种程序模块化和参数化机制, 可以有效表达函数式程序结构之间复杂的依赖关系[19]。基于通用操作模板, 我们利用 locale 定义高阶泛化函数, 用于刻画搜索树类结构的一系列实例化操作。在此基础上, 针对特定类型的搜索树结构, 可通过 locale 的继承机制引入更具结构特性的约束与操作定义, 进一步构造特定变体扩展操作, 以对实例化操作进行补充和增强, 图 3 展示了函数式建模库中搜索树类结构的通用区域及部分继承关系。

2.2 验证引理库

验证引理库是本文框架实现高效形式化验证的核心基础, 由功能正确性验证引理集与结构正确性

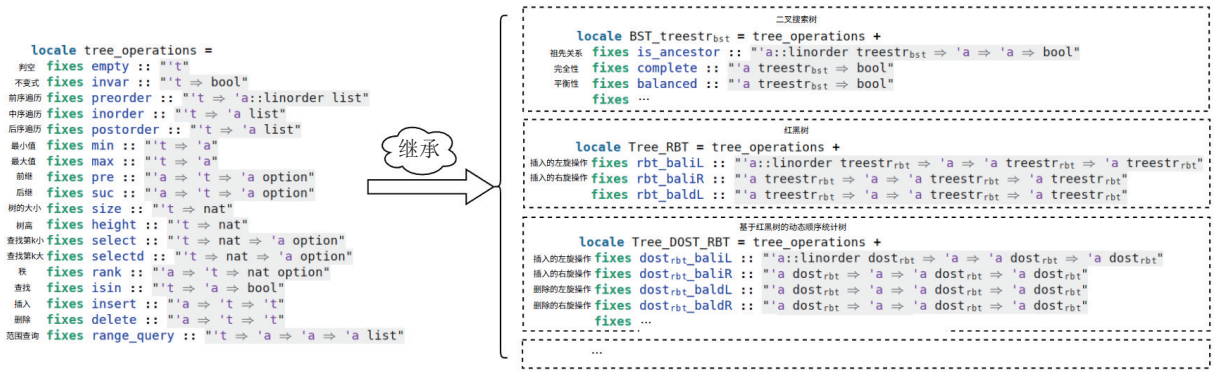


Fig. 3 Generic locale of search tree structures and partial inheritance relationships

图 3 搜索树类结构通用区域及部分继承关系

验证引理集组成，将搜索树类算法的功能正确性与结构正确性验证解耦，显著降低验证复杂度。本节从功能正确性与结构正确性2个维度出发，分别阐述引理库的设计与实现。

验证引理库包含了二叉搜索树、AVL树、红黑树及变体、区间树及变体、动态顺序统计树类、2-3树、2-3-4树、AA树、splay树在内的14种搜索树结构。

在功能验证引理集中共包含224条功能正确性主引理以及大量辅助引理，保证搜索树结构各类操作在执行过程中均严格符合既定的功能正确性要求，并为后续验证不同搜索树算法提供了坚实的逻辑支撑。

在结构验证引理集共包含66条子结构不变式及大量辅助引理，覆盖了搜索树在修改类操作中的关键约束条件。通过对这些子结构不变式的严格刻画与验证，保证了操作执行前后搜索树结构属性的维持，从而有效避免因局部调整而引发的全局结构失效。

2.2.1 功能正确性验证引理集

功能正确性验证算法操作是否满足输入-输出的语义规约，搜索树结构与列表线性表示具有严格的数学等价关系[11]，搜索树操作的语义可通过列表操作间接描述，引入列表辅助函数及其对应引理作为中介，进而验证搜索树类通用操作的功能正确性。

通过中序遍历函数，我们可以将任意搜索树映射成一个有序列表。具体地，我们在Isabelle/HOL中定义了通用的中序遍历函数：

```
fun inorder :: " 'a treestr => 'a list" where
"inorder Leaf = []" |
"inorder Node l (a, [_]) r=inorder l @ a # inorder r"
```

*inorder*函数主要提取每个节点的核心值信息，忽略所有的附加信息。考虑到节点可能包含多个附加信息位，采用占位符机制统一处理这些附加信息。

在此基础上，我们为每种操作在列表域中设计了对应的辅助函数，通过对列表进行模式匹配和条件分支来刻画理想的语义规范。功能正确性证明的关键在于建立搜索树操作和列表辅助函数之间的行为等价，即执行搜索树操作后再中序遍历得到对应的列表，与先中序遍历原树再在列表上执行对应的辅助函数得到的结果始终保持一致。对于特定搜索树结构的扩展操作，部分函数由于其依赖节点层次或结构关系，难以直接转化为列表操作进行验证。在此类情形下，可通过构造与该操作语义等价的辅助函数，借助其与通用函数之间的等价关系，将验证任务转化为对这些连接性引理的证明。图4展示了通用操作功能正确性的逻辑规约，并呈现了部分结构变体在功能正确性层面的继承关系，用以说明不同搜索树在复用通用操作逻辑规约基础上如何引入自身特性，并通过继承机制拓展验证路径。

2.2.2 结构正确性验证引理集

在机械化验证过程中，还需要通过验证数据结构在操作过程中的不变性来保证其正确性。不变性是指操作执行前后某些性质或者特征始终保持不变。对数据结构进行不变性证明时，可通过证明在执行操作前后不破坏其依赖的不变式。不变式是数据结构要符合的逻辑规则式[20]。

搜索树类结构的性质可被定义为：

1)搜索树性质不变式(简称$invar_{sort}$)，即树中节点值的中序遍历序列呈现线性递增的顺序。*inorder*函数以中序遍历的方式将搜索树结构中的节点转换为列表。转换完成后利用*sorted*函数判断该列表是否按升序排列。$invar_{sort}$根据2个函数组合定义为：

definition "$invar_{sort}\ t \equiv sorted\ (inorder\ t)$"。

2)附加性质不变式(统称$invar_{extra}$)是可选约束项，根据具体搜索树结构的特性选择性补充。当实例化到某一具体搜索树结构且需保持附加性质不变时(如颜色信息、树的高度、树的大小等)，应结合具体需求引入相应的附加不变式。

根据1)、2)统称为搜索树结构的不变式(简称*invar*)，其定义为：

definition "$invar\ t \equiv invar_{sort}\ t \wedge invar_{extra}\ t$"。

对于修改操作(插入、删除)的函数，在执行过程中可能会改变节点信息，则需要引入额外的逻辑规则来判断其是否保持结构特性。

在形式化验证过程中，不同类型的搜索树结构其不变式(*invar*)各不相同，这些不变式在定义和约束数据结构行为方面起着核心作用。因此，需要针对每种搜索树结构的特性对其进行拆分和调整，同时构建相应的辅助引理集以支持证明流程。图5列举了部分典型搜索树结构需要满足的结构不变式，旨在说明结构不变式在具体建模过程中的多样性。

3 搜索树类算法的建模及自动化验证

为将搜索树类算法的形式化验证成果应用于工业场景，本文提出了搜索树类算法的建模及自动化验证方法。利用算法规约精确定义输入-输出，确保算法设计的正确意图；在模块化建模阶段，通过复用

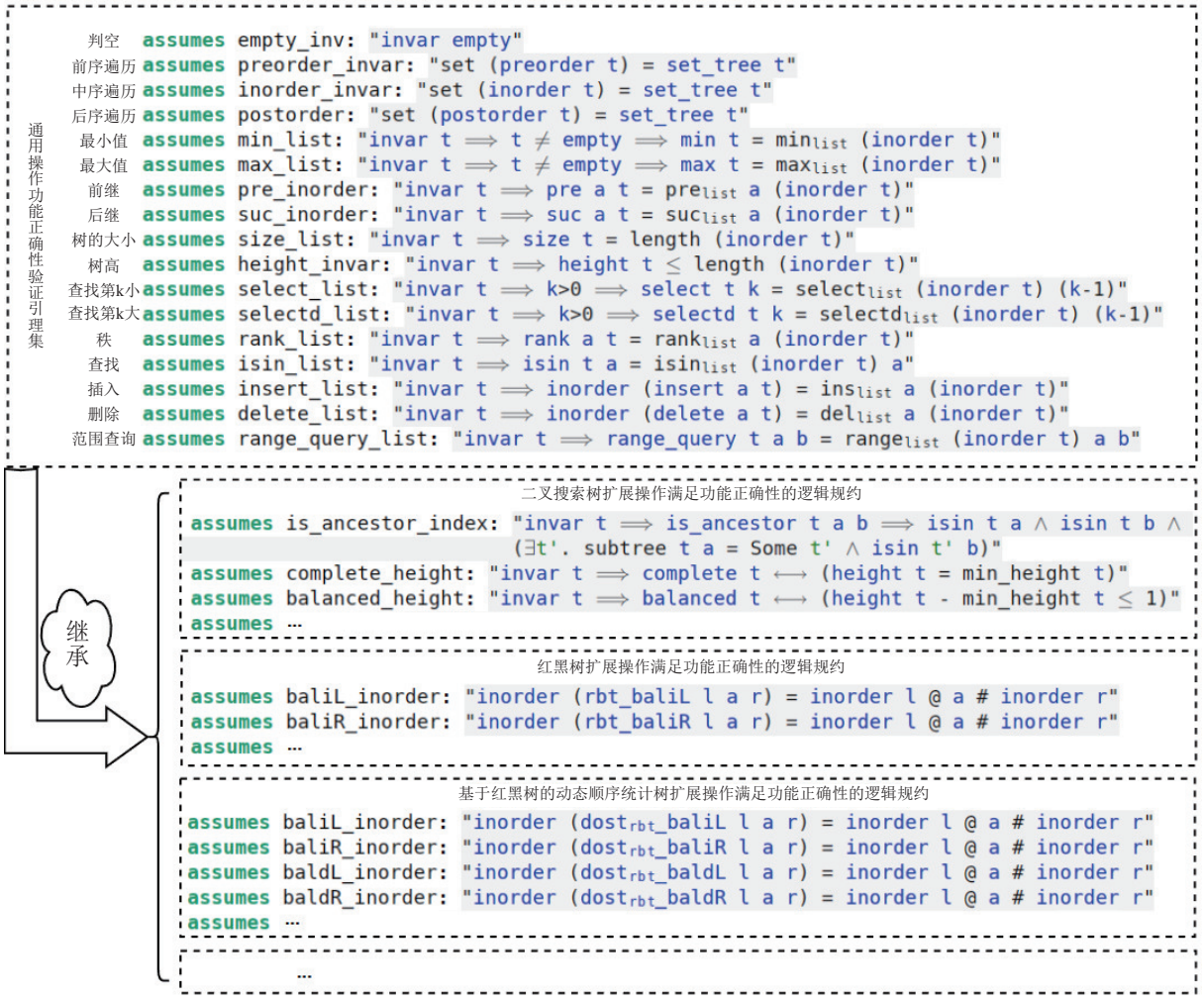


Fig. 4 Logical specifications of functional correctness for generic operations and partial inheritance relationships

图 4 通用操作功能正确性的逻辑规约及部分继承关系

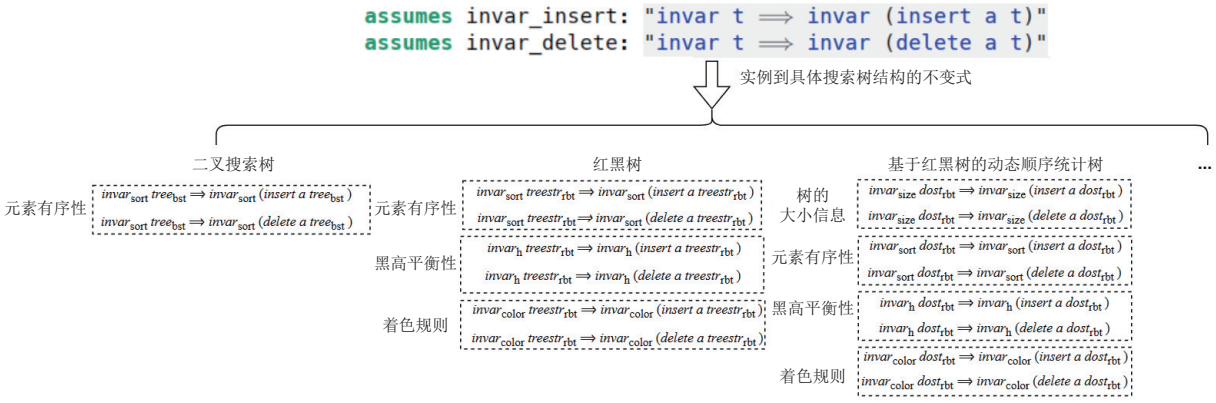


Fig. 5 Partial instantiation of structural invariants

图 5 结构不变式的部分实例化

已验证的函数式建模库并在此基础上注入核心算法逻辑，构建严谨可测的算法代码框架；自动化分块验证过程中，借助验证引理库和自动化证明方法，对算法的功能正确性和结构正确性进行验证；最后，通过代码生成机制将验证无误的算法导出为结构化、可读的函数式可执行程序。

3.1 模块化建模

在搜索树算法的开发中，模块化建模不仅是实现自动验证和程序生成的基础，更是保证整个开发流程具有可维护性、可重用性和可扩展性的关键。

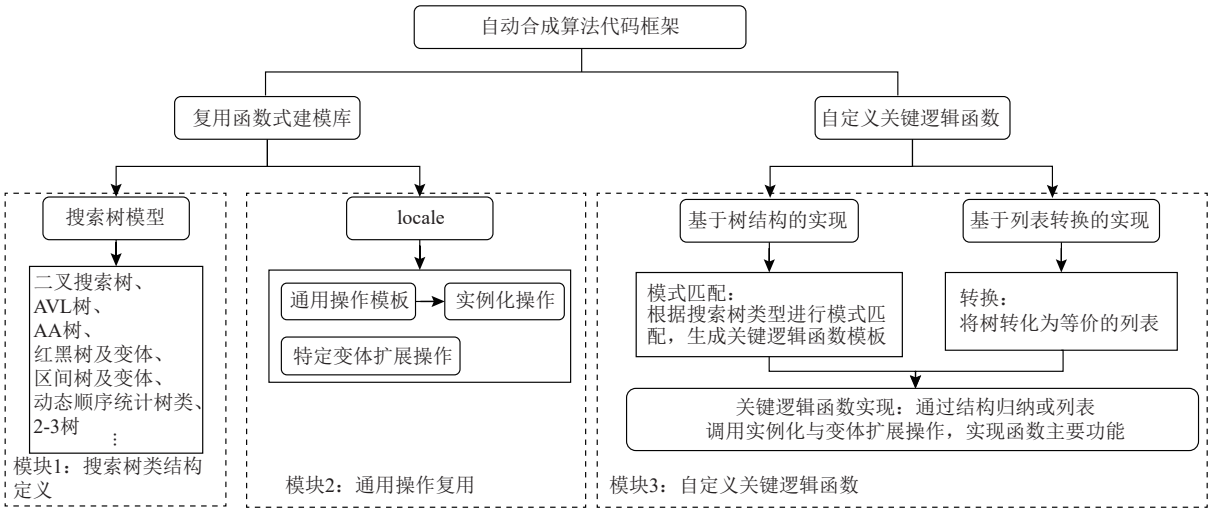


Fig. 6 Framework for automatic synthesis of algorithm code

图 6 自动合成算法代码框架

其建模策略是将算法设计拆解为 3 个模块：模块 1 是搜索树类结构定义模块，用于明确结构特性并确保类型安全；模块 2 是通用操作复用模块，通过复用已验证的操作函数降低算法建模与验证的复杂度；模块 3 是自定义关键逻辑函数模块，设计人员基于树结构或列表转换的实现方式，调用实例化与变体扩展操作实现算法整体功能。图 6 展示了基于模块化建模策略的自动合成算法代码框架。

算法的形式规约用于精确定义算法的行为和期望结果，是设计和编制程序的出发点，也是验证程序是否正确的依据。我们使用霍尔三元组 $(\{P\}, C, \{Q\})$ 明确算法的语法边界：前置条件 $\{P\}$ 描述在执行算法之前必须满足的条件；程序体 C 是算法具体的实现逻辑；后置条件 $\{Q\}$ 描述算法执行完成后应满足的条件。通过这一规范，为编码与形式化证明提供参考标准。

基于算法规约可直接或间接明确所需的搜索树结构，从而确定搜索树类结构定义模块（模块 1），该模块负责构建满足特定需求的搜索树结构。若规约中提及“平衡”“节点统计”等概念时，会相应地映射到子树高度、节点权重或统计信息的需求。形式规约通过刻画访问模式与性能目标，限定所选用的搜索树结构必须具备的关键组织特性，为后续的建模与实现提供明确指引。函数式建模库提供了丰富的搜索树类结构，开发者可根据搜索树模型实例化直接复用已定义的结构，从而进一步调用相关操作和引理。通过 imports 语句导入相关程序，即可在建模算法和证明定理时直接调用辅助函数与引理。图 7 展示了算法在 Isabelle 中复用程序的脚本模块化过程。

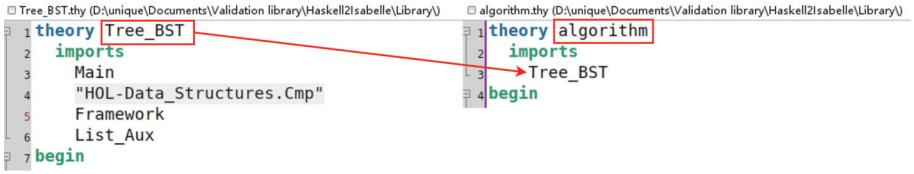


Fig. 7 Script modularization of algorithms in Isabelle

图 7 算法在 Isabelle 中的脚本模块化

通用操作复用模块（模块 2）依托已验证的操作函数简化建模过程。函数式建模库中为搜索树类结构封装了共有的基本操作，还针对特定变体提供额外的扩展操作。借助 locale 机制，开发者在编写算法时可检索相应 locale 中是否有可复用的操作，无需重写或重新验证。通过复用已有函数，最大限度地减少算法的构造和验证工作量，提升建模效率和一致性。

自定义关键逻辑函数模块（模块 3）是模块化建模的核心阶段。该模块实现通常分为 2 类方式：一种是基于树结构的实现，根据搜索树的数据类型生成关键逻辑函数的基础模板，从而覆盖所有可能的构造形式并确保分支完整性，这种方式能够直接反映搜索树的层次化语义，适用于需要紧密依赖树结构性质的算法；另一种是基于列表转换的实现，将

树操作转化为等价的列表操作，适用于功能逻辑更偏向序列化处理的场景。而后根据结构归纳或列表，调用实例化与变体扩展操作，从而得到满足特定功能需求的实现。该过程不仅确保了函数逻辑与数据结构特性的紧密契合，还在设计层面提高了代码的可维护性与可扩展性，为后续的自动化证明奠定基础。

3.2 自动化分块验证

完成算法的函数式建模后，为确保其实现严格符合形式规约，我们采用自动化分块验证策略对算法进行形式化验证，将验证过程划分为 2 个模块以提高验证效率。验证模块 1 针对算法中复用的通用函数，这些函数在定义时已通过形式化验证，其正确性得到充分保证，无需二次证明。验证模块 2 聚焦于算法的核心逻辑部分，将最大化复用已有验证引理库，并结合自动化验证策略加速证明。图 8 展示了自动化分块验证策略。

针对建模过程中复用的搜索树结构和操作（验证模块 1），验证引理库已提供功能和结构正确性证明，确保它们在输入输出层面与形式规约完全一致，同时保证任何基于该结构的修改类操作都不会破坏既有不变式。复用这些函数时，无需重新对其内部逻辑进行验证，只需信赖对应的引理集即可。

对于主算法中每一个待证明的定理（验证模块 2），验证过程中可借助验证引理库中的引理集。对于待证主定理 T，可采用以下自动化证明策略进行验证：

1）目标拆分。将主定理中各个独立的结论或条件子句拆分成若干互不相交的子目标，每个子目标对应一条子引理，每条子引理只聚焦单一性质或一组相关的条件。通过这种方式，子目标更简洁、更易于使用结构归纳和自动化方法完成证明。主定理只需将已证明的引理添加到证明过程，即可完成推导。此步骤可完全自动化。

2）归纳法。结构归纳针对递归定义的搜索树结构，系统自动生成归纳规则，将任意搜索树结构的证明任务分解为平凡与非平凡 2 种情况。平凡情况对应树为空，证明空树情况满足目标性质；非平凡情况则假设各个子树已满足该性质，再证明由它们构造出的更复杂场景也同样满足。induct 和 induction 都可表示对项使用归纳法，使用归纳规则前，可用全称量词约束每个标记为 arbitrary 的变量，rule 选项可用于指定归纳规则。

3）auto 方法。Isabelle 定理证明器中常用的自动化证明方法结合了化简和经典逻辑推理，能自动处理许多简单的证明目标。simp 表应用化简行为，需要添加验证引理库中的辅助引理集，从而化简子目标或直接得证。子目标中如果含有 if、case 等表达式，可配合修饰符 split 进行分解，使 auto 方法可以分别处理分解后的情况，从而完成目标的证明。

4）try 命令。尝试多种自动化方法证明当前命题而不改变证明状态。该命令会调用 Sledgehammer[21] 等用于自动化证明的组件，按顺序尝试多个证明方法。整个过程中，try 命令自动利用当前上下文中的验证引理库，检索并复用已知的辅助引理，依次尝试

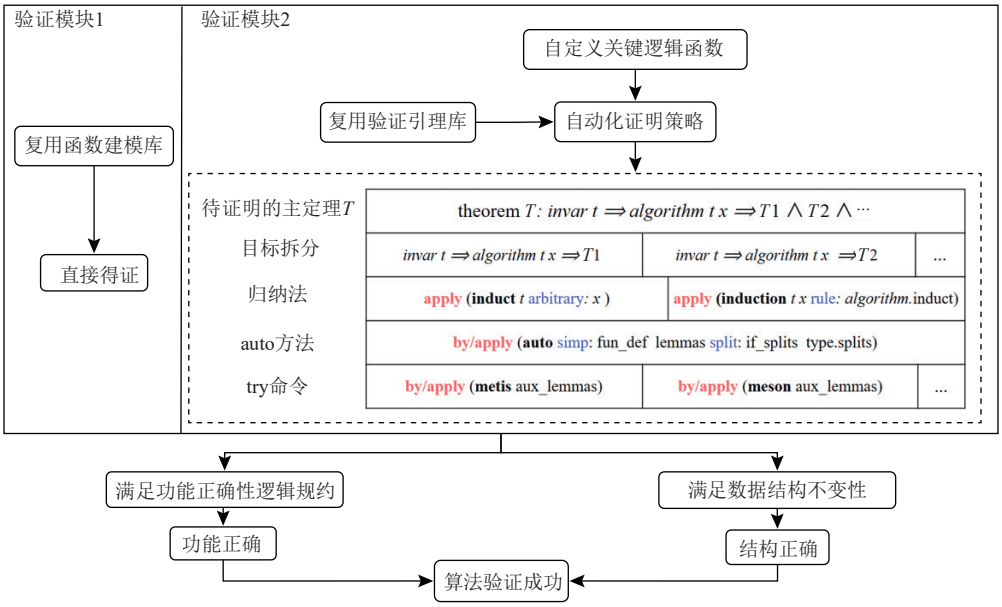


Fig. 8 Automated block-wise verification strategy

图 8 自动化分块验证策略

多个方法直到成功或全部失败。值得注意的是，在自动化验证机制的性能方面，try 命令会在可行范围内自动搜索与目标适配的证明策略。随着引理库规模的增加，耗时确实会有所增加，但由于本文仅构建了与搜索树类结构相关的引理，其规模远小于 Isabelle 内置的引理集，因此整体搜索空间仍然有限。且 try 命令内部设置了搜索深度与时间限制，该增长趋势得到有效抑制，并未呈现指数级膨胀，从而保证了整体验证效率与可用性。

结合 Nipkow 等人[11,21]提出的 Isabelle 自动验证方法，本文提出的自动化证明策略整合了归纳法、auto 方法与 try 命令，实现了较高程度的证明自动化。在该策略中，归纳法用于分解复杂问题，auto 方法完成基本的自动推导，而 try 命令则借助自动化组件，搜索验证引理库中的相关引理，探索可行的证明路径。通过这种组合方式，本文的自动化验证能够充分利用构建的验证引理库，将复杂问题分解成小的子问题，在保证辅助引理可靠性和通用性的前提下，高效处理不同场景下的证明任务。

3.3 程序生成

本文所提出的程序生成方法具有较强的普适性，能够支持多种函数式可执行程序的生成。为展示方法的有效性与可行性，本文选择 Haskell 作为主要示例语言。Haskell 具备高度抽象、类型安全与可验证性强等特性，适用于开发可靠性要求高的软件系统[22]。通过将经形式化验证的算法自动转化为 Haskell 程序，可实现工业级应用。尽管以下讨论主要围绕 Haskell 程序生成展开，但所提方法并不限于此，其应用范围可进一步推广至其他函数式编程语言。

以算法规约为核心驱动，通过将算法意图精确定义为输入输出之间的逻辑约束，确保程序设计阶段即具备明确的正确性目标。在建模过程中，数据结构通过复用建模库中已验证的数据结构与通用函数模板，显著降低开发负担。关键逻辑的实现基于类型系统，借助定理证明器的静态检查机制，在定义阶段规避不合理分支与非法操作，保障建模质量。验证阶段依托定制的引理库，对功能正确性与结构正确性进行验证，通过自动化分块验证以及基于验证引理库的证明策略，大幅提高自动化验证能力，减少验证成本。

已经过验证的 Isabelle 代码具有类型约束明确、结构表达清晰、逻辑语义严谨等特点。通过预处理、翻译、序列化 3 个阶段，可以转换为 Haskell 程序代码[23]。图 9 展示了程序生成流程。

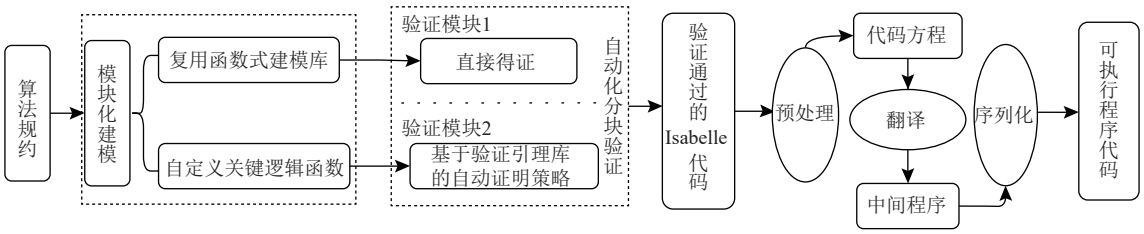
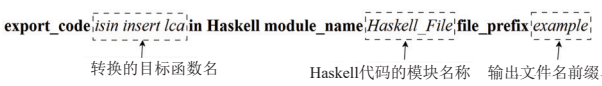


Fig. 9 Program generation process

图 9 程序生成流程

由 datatype 和 definition/primrec/fun 声明的等式可直接用于代码生成，export_code 命令是 Isabelle/ML 提供的顶层命令[23]，用于从当前理论中导出可执行代码，可接收多个函数名称，并为这些函数及其所有依赖自动生成目标语言的实现代码。命令格式为：



4 案例研究及适用范围

为评估本文所提方法的有效性，我们在最近公共祖先、动态维护 Top k、日程安排冲突检测、滑动窗口统计等算法中成功实验。限于篇幅，本文选取最近公共祖先(LCA)算法及动态维护 Top k 算法为实例，展示所提方法适用性，并明确了方法的适用范围。

4.1 LCA 算法案例

LCA 问题是计算机科学中的经典算法问题之一，对于社交网络、生物信息学以及互联网路由算法等应用场景具有重要意义。在一棵无环树上，任意两节点总能找到它们的公共祖先，而“最近”公共祖先即指那些公共祖先中深度最大的节点，亦即位于路径交叉处最靠近两者的节点。

4.1.1 LCA 算法模块化建模

在对二叉搜索树的 LCA 算法进行形式化开发时，采用霍尔三元组严格描述算法的行为规约：

$$\{bst\ t\}lca := lca_bst(t,p,q)\{isin(t,p)\wedge isin(t,q)\wedge isin(t,lca)\wedge is_ancestor(t,lca,p)\wedge is_ancestor(t,lca,q)\wedge(\forall v.is_ancestor(t,v,p)\wedge is_ancestor(t,v,q)\longrightarrow is_ancestor(t,v,lca))\}$$

前置条件{*P*}要求输入的树 *t* 是一棵有效的二叉搜索树。程序体 *C* 执行 *lca_bst*(*t*, *p*, *q*) 函数，计算节点 *p*、*q* 在树 *t* 中的 LCA，并返回节点 *lca*。后置条件{*Q*}则保证 *p*、*q*、*lca* 节点存在于树 *t* 中，*lca* 是 *p*、*q* 的 LCA，即 *lca* 是 *p*、*q* 的 LCA，且在所有这样的祖先中 *lca* 的深度最大。

根据以上的算法规约描述，我们可以直接明确该算法使用的数据结构为二叉搜索树，该结构已存在于构建的函数式建模库（Tree_BST）中，即可直接复用建模库中二叉搜索树结构的定义，将相关的文件导入到当前文件（LCA_basedBST）中，图 10 为复用二叉搜索树结构的示例图。

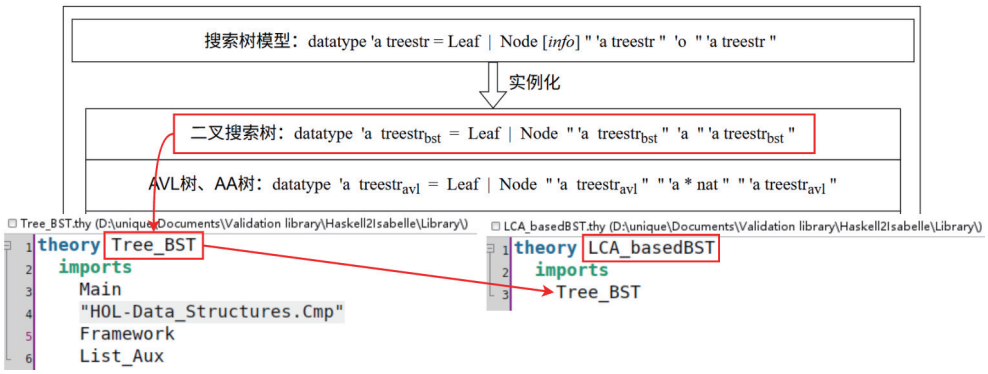


Fig. 10 Reuse of binary search tree structures

图 10 复用二叉搜索树结构

区域继承机制允许我们在已有 locale 的逻辑框架之上，扩展出更具结构特征的新区域。对于二叉搜索树，在搜索树类结构通用区域（tree_operations，定义见图 3）的基础上进一步引入区域继承，通过显式指定局部参数与逻辑规约，构建出刻画二叉搜索树行为的新 locale（BST_treestr$_{bst}$）。图 11 展示了用 locale 刻画的二叉搜索树通用操作，并利用 interpretation 命令将 locale 及其内部信息实例化传递到当前上下文。LCA 算法实现过程中需要判断节点是否存在于树中，二叉搜索树的通用操作包含的 $isin_{bst}$ 函数满足所需功能，且二叉搜索树变体扩展操作 $is_ancestor_{bst}$ 函数可用于辅助验证算法功能正确性，因此我们可以直接调用建模库中的这 2 个函数。

在上述组件的基础上，用户只需专注于 LCA 算法的关键逻辑实现，根据二叉搜索树结构进行模式匹配，覆盖所有分支，即 Leaf 和 (Node *l v r*)，并在此基础上通过结构化归纳，调用实例化操作（$isin_{bst}$ 函数）填充具体细节。实现中需判断节点 *p*、*q* 是否存在于当前树 *t* 中，若都存在，再根据当前节点值与 *p*、*q* 的大小关系决定递归方向，最终在值处于中间范围时返回当前节点作为 LCA。依据算法的关键逻辑，可以得到如下 Isabelle 的最终朴素实现：

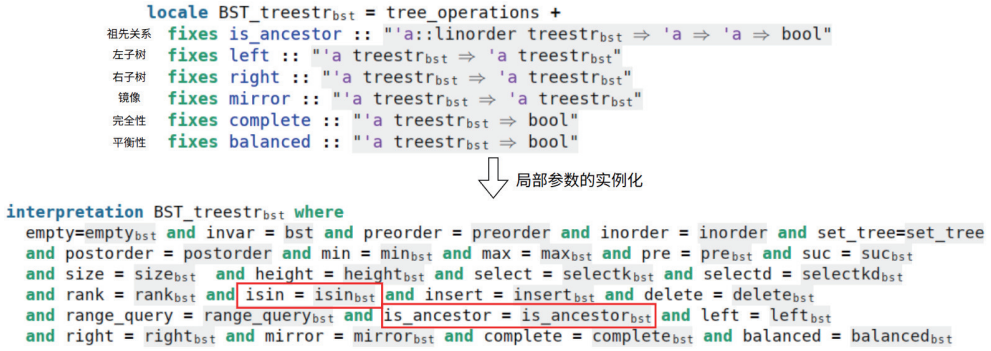


Fig. 11 Locale and parameter instantiation of binary search tree

图 11 二叉搜索树区域及参数实例化

```
fun lca_bst::"'a::linorder treestr ⇒ 'a ⇒ 'a⇒ 'a option"
where "lca_bst Leaf _ _ = None" |
      "lca_bst (Node l v r) p q =
(if isin_bst (Node l v r) p ∧ isin_bst (Node l v r) q then
          (if v > max p q then lca_bst l p q
           else if v < min p q then lca_bst r p q
           else (Some v))
          else None) "
```

规约确保语义精确，建模库复用最大化避免重复工作，以及关键逻辑实现专注算法本质，最终我们构建了 LCA 算法模型，图 12 展示自动合成 LCA 算法框架过程，为后续的自动化分块验证进而函数式可执行代码生成打下坚实的基础。

4.1.2 LCA 算法自动化分块验证

基于第 3.2 节给出的基于验证引理库的自动化分块验证策略，可在 Isabelle 中以较低成本验证 *lca_bst* 函数的功能正确性。在该算法的实现过程中，调用了 $isin_{bst}$ 函数用于判断节点是否存在于树中；在满足后置条件时，还需判断某节点是否为另一个节点的祖先，所需的祖先关系由建模库中已定义的 $is_ancestor_{bst}$ 函数提供。图 13 展示了 $isin_{bst}$ 和 $is_ancestor_{bst}$ 函数相关的验证引理集。这些引理可作为验证前提简化 *lca_bst* 功能正确性定理的证明，并提升自动化验证的成功率。

接下来，我们关注于 LCA 算法本身的功能正确性，即根据规约完成函数式建模后，可构造定理 1 确保算法的语义实现与其完全一致。

定理 1. 函数 *lca_bst* 功能正确性：

theorem lca_bst_correct: "$bst\ t \Rightarrow lca_bst\ t\ p\ q$ = Some $lca \Rightarrow isin_{bst}\ t\ p \wedge isin_{bst}\ t\ q \wedge isin_{bst}\ t\ lca \wedge is_ancestor_{bst}\ t\ lca\ p \wedge is_ancestor_{bst}\ t\ lca\ q \wedge (\forall v.\ (is_ancestor_{bst}\ t\ v\ p \wedge is_ancestor_{bst}\ t\ v\ q) \rightarrow is_ancestor_{bst}\ t\ v\ lca)$"。

定理 1 表明当输入满足有效的二叉搜索树 *t*，*p*、*q* 的 LCA 是 *lca*，则 *p*、*q*、*lca* 节点均存在树 *t* 中，*lca* 是 *p*、*q* 的祖先，并且在所有的祖先中，*lca* 的深度最大。为了让证明更加清晰且易于机械化处理，我们使用定理拆分策略将这一综合命题拆分为 3 个相互独立的子目标：目标 1 关注节点存在性，目标 2 关注祖先关系本身，目标 3 则专注最深性条件。

对于以上 3 个引理的验证，在采用归纳法分解目标后，使用 simp 方法进行化简和分解，可将结论拆分为多个子目标，基于验证引理库使用 try 命令，所有的子目标都可以自动证明。定理 1 的功能正确性通过合并 3 个引理即可得证，如图 14 所示。

由于 *lca_bst* 函数不会改变树的属性，无需验证结构正确性。整个算法的证明过程充分依赖已验证的 $isin_{bst}$、$is_ancestor_{bst}$ 及排序等辅助引理，以最小的手工参与实现了对 LCA 算法功能正确性的自动化证明。图 15 展示了 LCA 自动化分块验证过程。

4.1.3 LCA 算法 Haskell 程序生成

完成 *lca_bst* 算法的形式化建模与验证后，我们得到正确的 LCA 算法实现，通过 export_code 命令可

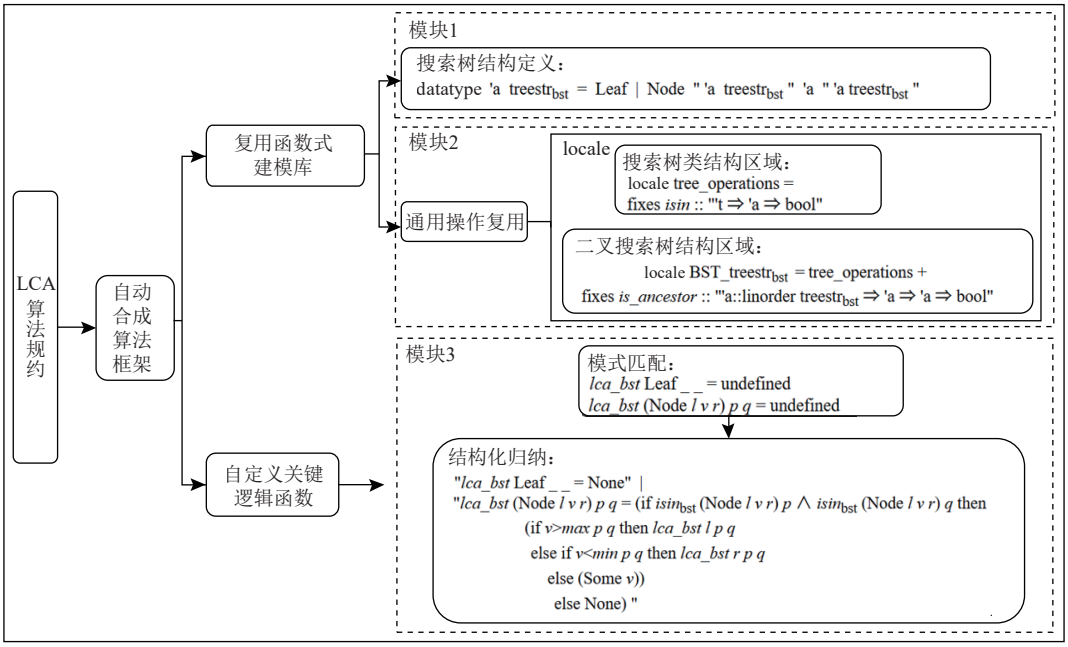


Fig. 12 Framework for automatic synthesis of LCA algorithm

图 12 自动合成 LCA 算法框架

$isin_{bst}$函数验证引理集

$bst\ t \Longrightarrow isin_{bst}\ t\ x \longleftrightarrow x \in set\ (inorder\ t)$

$bst\ t \Longrightarrow isin_{bst}\ t\ a = isin_{list}\ (inorder\ t)\ a$

$is_ancestor_{bst}$函数验证引理集

$bst\ (\text{Node}\ l\ v\ r) \Longrightarrow is_ancestor_{bst}\ (\text{Node}\ l\ v\ r)\ v\ desc \longleftrightarrow isin_{bst}\ (\text{Node}\ l\ v\ r)\ desc$

$is_ancestor_{bst}\ t\ anc\ desc \Longrightarrow isin_{bst}\ t\ desc$

$bst\ (\text{Node}\ l\ v\ r) \Longrightarrow is_ancestor_{bst}\ l\ anc\ desc \Longrightarrow is_ancestor_{bst}\ (\text{Node}\ l\ v\ r)\ anc\ desc$

$bst\ (\text{Node}\ l\ v\ r) \Longrightarrow is_ancestor_{bst}\ r\ anc\ desc \Longrightarrow is_ancestor_{bst}\ (\text{Node}\ l\ v\ r)\ anc\ desc$

$bst\ t \Longrightarrow t = \text{Leaf} \Longrightarrow is_ancestor_{bst}\ t\ anc\ desc = \text{False}$

$bst\ t \Longrightarrow is_ancestor_{bst}\ t\ anc\ desc \Longrightarrow isin_{bst}\ t\ anc \wedge isin_{bst}\ t\ desc$

$bst\ t \Longrightarrow isin_{bst}\ t\ x \Longrightarrow is_ancestor_{bst}\ t\ x\ x$

$bst\ t \Longrightarrow is_ancestor_{bst}\ t\ a\ b \Longrightarrow isin_{bst}\ t\ a \wedge isin_{bst}\ t\ b \wedge (\exists t'.\ subtree\ t\ a = \text{Some}\ t' \wedge isin_{bst}\ t'\ b)$

Fig. 13 Verification lemma set for $isin_{bst}$ and $is_ancestor_{bst}$ functions

图 13 $isin_{bst}$ 和 $is_ancestor_{bst}$ 函数的验证引理集

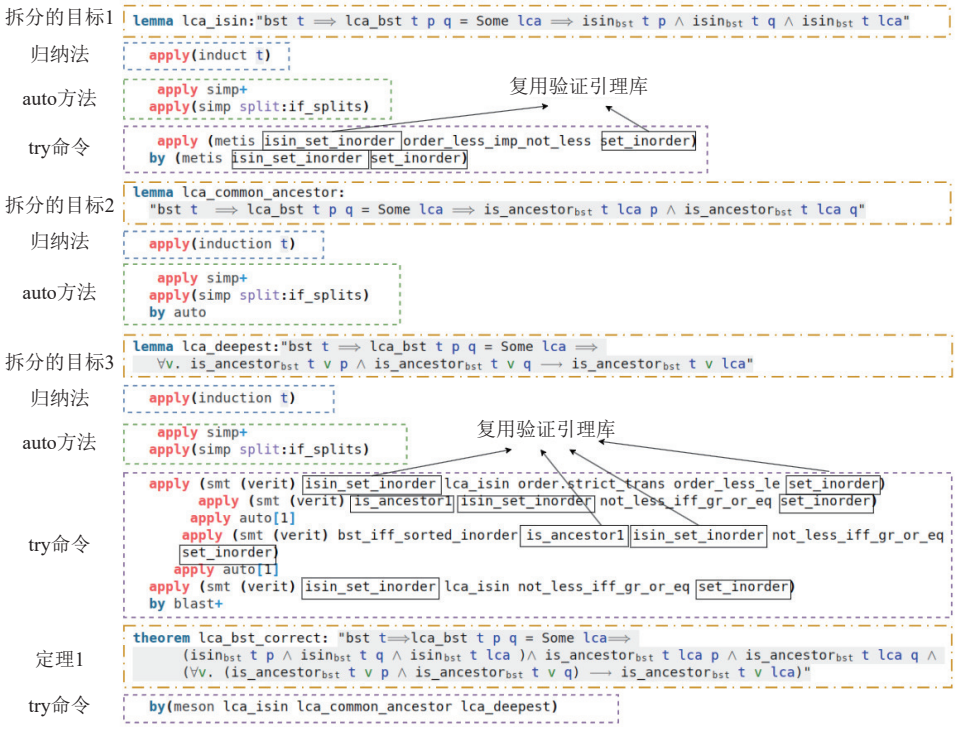


Fig. 14 Automated verification of *lca_bst* functional correctness

图 14 *lca_bst* 功能正确性的自动化验证过程

将其依赖导出为符合 Haskell 语法的源文件。如图 16 所示。该生成过程保留了原始 *lca_bst* 算法的函数式结构和抽象逻辑，确保程序在形式语义上既完全符合高层逻辑规约，又具工业级的可读性和可维护性。

4.2 动态维护 Top *k* 算法案例

动态维护 Top *k* 算法是处理海量流式数据与实时系统的核心技术之一，支撑了互联网内容推荐、实时监控、金融风控以及大规模数据分析等典型应用场景。该问题的关键在于如何在动态数据更新过程中始终高效且正确地维护并返回集合中的前 *k* 大的元素。

4.2.1 动态维护 Top *k* 算法模块化建模

为了形式化刻画算法行为，采用算法规约对其

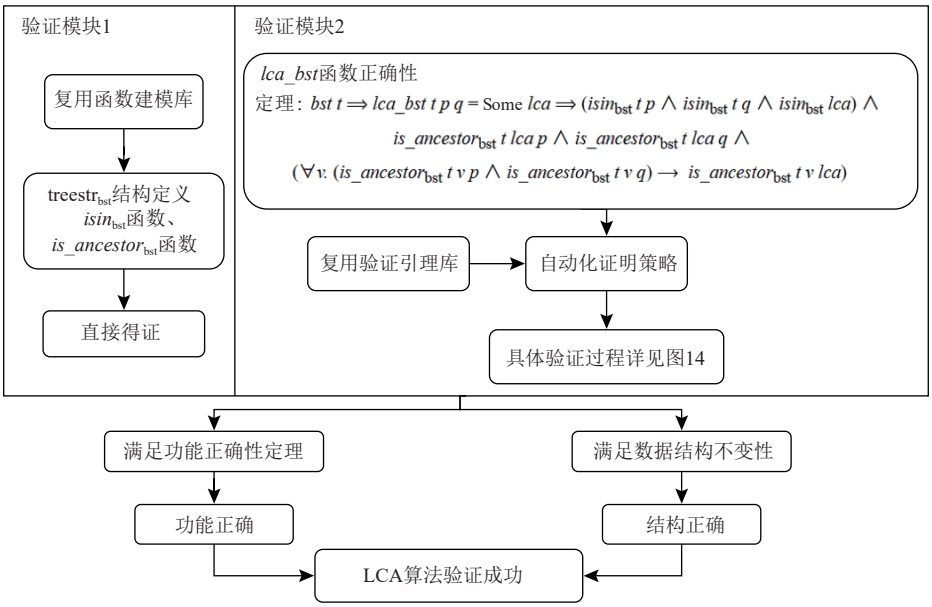


Fig. 15 Automated block-wise verification of LCA algorithm

图 15 LCA 算法自动化分块验证

```
export_code lca_bst in Haskell module_name Lca file_prefix lcaalgorithm
```

导出文件

lcaalgorithm Directory

Lca.hs 1.6 kB File

具体Haskell代码

```
data Tree a = Leaf | Node (Tree a) a (Tree a);
cmp :: forall a. (Eq a, Linorder a) => a -> a -> Cmp_val;
cmp x y = (if less x y then LT else (if x == y then EQ else GT));

isin_b_s_t :: forall a. (Eq a, Linorder a) => Tree a -> a -> Bool;
isin_b_s_t Leaf x = False;
isin_b_s_t (Node l a r) x = (case cmp x a of {
                              LT -> isin_b_s_t l x;
                              EQ -> True;
                              GT -> isin_b_s_t r x;
                            });

min :: forall a. (Ord a) => a -> a -> a;
min a b = (if less_eq a b then a else b);

max :: forall a. (Ord a) => a -> a -> a;
max a b = (if less_eq a b then b else a);

lca_bst :: forall a. (Eq a, Linorder a) => Tree a -> a -> a -> Maybe a;
lca_bst Leaf uu uv = Nothing;
lca_bst (Node l val r) p q =
  (if isin_b_s_t (Node l val r) p && isin_b_s_t (Node l val r) q
    then (if less (max p q) val then lca_bst l p q
           else (if less val (min p q) then lca_bst r p q else Just val))
    else Nothing);

}
```

Fig. 16 Haskell code generation for LCA algorithm

图 16 LCA 算法的 Haskell 代码生成

操作进行描述，主要包括查询、插入与删除操作。

采用霍尔三元组严格描述查询操作的行为规约：

$$\{invar\ t, k > 0\}\ list := top_k(k, t),$$

$$\{list = rev(drop(size\ t - k)(inorder\ t))\}。$$

表明若搜索树满足不变式且 k>0，则执行 *top_k* 返回的结果等价于树的中序遍历逆序后的前 k 个元素。

对于插入操作，其规约可表述为：

$$\{invar\ t, k > 0\}(t', list) := insert_top_k(x, k, t),$$

$$\{invar\ t' \wedge list = rev(drop(size\ t' - k)(inorder\ t'))\},$$

即在搜索树不变式保持的前提下执行 *insert_top_k* (x, k, t)，得到更新后的树 t'与结果 *list*，要求 t' 继续满足不变式且 *list* 与更新后的列表逆序的前 k 个元素保持一致。删除操作与插入操作具有相似的规约，这里不再赘述。

根据上述规约要求，算法需要频繁动态插入与删除，故在搜索树类型上选择基于红黑树的动态顺序统计树实现以保证平衡性与高效性。在建模过程中，通过导入函数式建模库中的 Tree_DOST_RBT 文件，可直接复用基于红黑树的动态顺序统计树定义，并将其引入到当前文件 Dynamic_Maintain_Top_k 中。实现过程中依赖库中已经过验证的通用操作，见图 17，包括中序遍历 (*inorder*)、树大小 (*size*)、结构不变式 (*invar*)、插入 (*insert*) 和删除 (*delete*) 函数，这些复用的操作建模代码共计 84 行，而完整的验证代码需要 212 行。

在算法的自定义关键逻辑函数部分，可利用中序遍历的逆序收集前 k 个节点，从而形成完整的 Top k 列表。插入与删除操作则基于查询逻辑扩展实现，具体算法实现如下：

```
//查询
definition Top_k :: "nat ⇒ 'a dost_rbt ⇒ 'a list" where
  "Top_k k t = (if k > dost_size t
     then rev(dost_rbt_inorder t)
     else take k (rev(dost_rbt_inorder t)))"
//插入
```

definition *insert_top_k* :: "'a::linorder ⇒ nat ⇒ 'a $dost_{rbt}$ ⇒ ('a $dost_{rbt}$ * 'a list)" where

"*insert_top_k x k t* =

(let *t'* = $dost_{rbt}$_*insert x t* in (*t', Top_k k t'*))"

4.2.2 动态维护 Top *k* 算法自动化分块验证

在算法验证过程中，红黑树及其变体作为搜索树类结构中较为复杂的一类，其结构性约束的正确性验证往往具有较高难度。基于本文所构建的验证引理库，上述正确性问题已通过预先证明得到系统性保障。在关键逻辑函数验证中，我们主要以插入操作的结构正确性验证为例说明，在验证引理库的基础上结合自动化分块验证策略，能够直接复用中间函数在库中已确立的正确性结果，插入操作的验证调用一次 try 命令即可得证，仅需 2 行代码，验证过程如图 18 所示。

```
locale Tree_DOST_RBT = tree_operations +
  fixes dostrbt_baliL :: "'a::linorder dostrbt ⇒ 'a ⇒ 'a dostrbt ⇒ 'a dostrbt"
  fixes dostrbt_baliR :: "'a dostrbt ⇒ 'a ⇒ 'a dostrbt ⇒ 'a dostrbt"
  fixes dostrbt_baldL :: "'a dostrbt ⇒ 'a ⇒ 'a dostrbt ⇒ 'a dostrbt"
  fixes dostrbt_baldR :: "'a dostrbt ⇒ 'a ⇒ 'a dostrbt ⇒ 'a dostrbt"
  assumes baliL_cor: "inorder (dostrbt_baliL l a r) = inorder l @ a # inorder r"
  assumes baliR_cor: "inorder (dostrbt_baliR l a r) = inorder l @ a # inorder r"
  assumes baldL_cor: "inorder (dostrbt_baldL l a r) = inorder l @ a # inorder r"
  assumes baldR_cor: "inorder (dostrbt_baldL l a r) = inorder l @ a # inorder r"

                    ⇩ 局部参数的实例化

interpretation Tree_DOST_RBT
  where empty = dostrbt_empty and invar = invarrbt and preorder = dostrbt_preorder
    and inorder = dostrbt_inorder and postorder = dostrbt_postorder and max = dostrbt_max
    and min = dostrbt_min and pred = dostrbt_pred and sucd = dostrbt_sucd and tsize = dost_size
    and select = dostrbt_select and selectd = dostrbt_selectd and  rank = dostrbt_rank
    and isin = dostrbt_isin and insert = dostrbt_insert and delete = dostrbt_delete
    and range_query = dostrbt_range_query and set_tree = set_dostrbt and dostrbt_baliL = dostrbt_baliL
    and dostrbt_baliR = dostrbt_baliR and dostrbt_baldL = dostrbt_baldL and dostrbt_baldR=dostrbt_baldR
```

Fig. 17 Locale and parameter instantiation of RBT-based dynamic order statistic tree

图 17 基于红黑树的动态顺序统计树区域及参数实例化

```
插入操作结构正确性  lemma insert_Top_k_invar: "invarrbt t ⟹ (insert_top_k x k t) = (t',list) ⟹invarrbt t'"
try命令            by (metis dostrbt_invarrbt_insert eq_fst_iff insert_top_k_def)
                                  ↖ 复用验证引理库
```

Fig. 18 Automated verification of structural invariant for insertion operations in the dynamically maintained Top *k* algorithm

图 18 动态维护 Top *k* 算法插入操作结构不变性自动化验证

4.2.3 动态维护 Top *k* 算法 Haskell 程序生成

借助 export_code 命令，可以将经过形式化建模与验证的动态维护 Top *k* 算法自动转换为可执行的 Haskell 程序，通过该方式生成的 Haskell 程序，既规避了人工编写可能引入的逻辑错误，又具备工业级的可读性与可维护性，实现从逻辑规约到可执行程序的转换。

4.3 适用范围

本文提出了搜索树类结构算法的建模及自动化验证方法，通过规约精确定义算法意图，结合函数式建模库与验证引理库支持，实现算法逻辑的严谨建模与自动化分块验证，并在功能与结构正确性得到保证后，自动生成结构化且可读的函数式可执行程序。本文所提方法主要适用于 2 种情况：

1）二叉搜索树类结构：包括二叉搜索树（BST）、红黑树（RBT）和 AVL 树等典型的二叉平衡搜索树。在这些树形结构中，不仅确保了其基础操作的功能正确性，且结构不变式维护均通过了形式化规约加以刻画，可在模块化建模阶段进行复用与扩展。本文方法能为不同变种的二叉搜索树提供统一的函数式建模与验证支撑，并尤其适合于对二叉搜索树类算法进行端到端的自动化验证，最终生成可信赖的函数式可执行程序。

2）多叉搜索树类结构：以 2-3 树和 2-3-4 树为代表。此类多叉搜索树结构在节点中存储多个关键字，并允许多个子树分支能够在更高层次上实现平衡性与搜索效率的统一。本文方法通过扩展对多关键字节点及其不变式的刻画，保证了在插入、合并或分裂等操作中结构约束得以保持，从而支持多叉树结构下算法功能与结构双重正确性的自动化验证与代码生成。

需要说明的是，Trie 树类结构其路径分支依赖字符序列的层级匹配而非全局有序性约束，导致树在节点访问逻辑、键比较方式及不变式定义上，与本文所提模型和通用操作建模之间存在一定差异，因此 Trie 结构暂未纳入本文方法的直接适用范围。鉴于本文方法在搜索树类结构底层逻辑上的通用性以及验证框架所具备的可扩展性，未来工作将针对 Trie 树的建

模特点，扩展现有验证引理库和函数式建模策略，使其能够支持 Trie 树及相关变体的功能与结构验证。

5 效果比对

在形式化验证领域，交互式定理验证是主流手段，其优势在于灵活性高、逻辑推理过程严谨，能够支持复杂系统的逐步构造性证明。然而该方法对开发者的专业知识和验证经验要求较高，构造辅助引理与组织证明结构通常需要大量人工干预，导致验证效率受限。随着自动化定理证明技术的持续发展，尤其是大语言模型在该领域的深入应用，基于生成式 AI 的验证方式逐渐展现出强大的扩展性与生成能力，在提高验证便捷性与自动化水平方面取得积极进展，但其在稳定性、逻辑一致性以及对复杂证明细节的把控方面仍存在一定不足。

为验证本文所提出方法的有效性，限于篇幅，本节以 LCA 算法为例，分别从交互式定理验证与生成式 AI 验证 2 个方面进行对比分析。为确保对比的一致性，本节统一使用完整的函数式建模输入，包括算法主体与中间辅助函数，作为方法验证的共同输入，并将验证目标统一设定为算法的正确性。

5.1 交互式定理验证

交互式定理验证方法要求用户手动完成建模、引理设计与证明组织等流程。在实际操作中，为完成验证，需用户手动展开函数定义、构造中间引理、设定归纳结构，并结合多种证明策略逐步推进。该过程虽保证了较高的形式严谨性，但对用户的逻辑表达能力与系统使用熟练度要求较高，整体验证效率不高，尤其在面对更为复杂的数据结构和多条件合取的定理时，验证成本呈指数增长。

以 LCA 算法为例，交互式定理验证方法主要依赖于人工驱动和策略指导，我们从算法规约出发，以函数 *lca_bst* 的功能正确性为目标。在交互式验证中，其主定理通常难以直接由自动化工具证明，因此为简化验证流程，开发者可对其进行定理拆分，将其划分为较易处理的子引理，逐步构造证明链。这里我们将其拆分为 2 个子引理。图 19 展示了交互式定理验证的过程，定理拆分成 2 个子目标，需构造 6 条自定义前置引理，验证代码总数为 178 行。

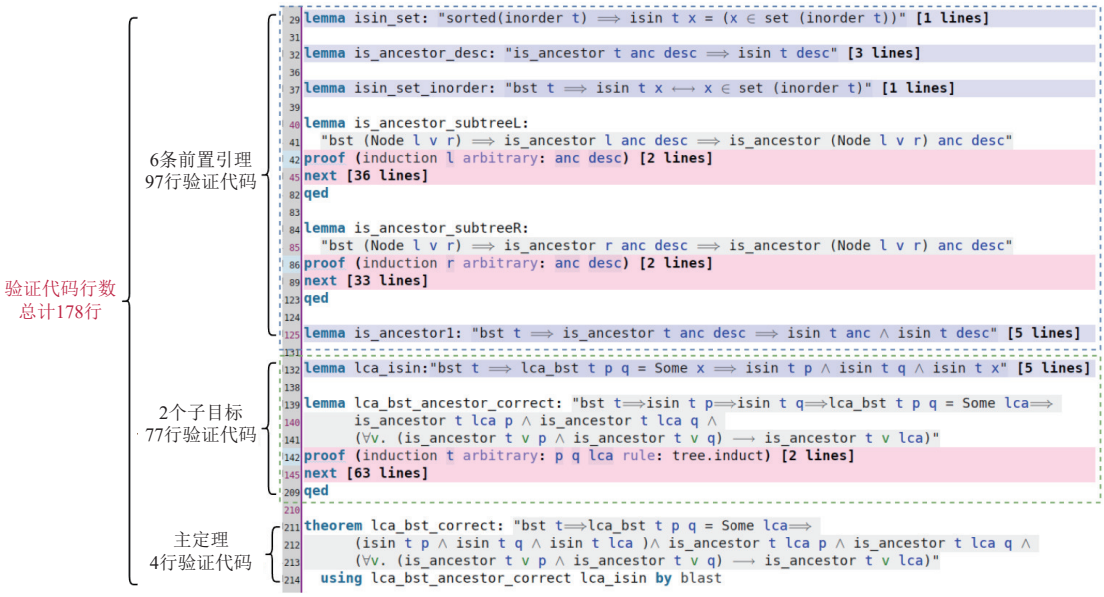


Fig. 19 Interactive theorem verification of LCA algorithm

图 19 LCA 算法交互式定理验证

函数定义通常具有递归结构，涉及多分支逻辑，验证者往往需要采用启发式策略构造中间引理。整个过程中，用户不仅要设计合适的引理，还需手动调用证明策略（如 induct, auto, simp, blast 等），反复调试证明路径，使之在各个子目标下都能被定理证明器接受。同时，开发者还需不断尝试不同的引理组合和证明顺序，以避免陷入无法自动化证明的瓶颈。这一验证过程具有高度的不确定性，严重依赖开发者的经验积累和对定理结构的理解，导致验证过程较为复杂、时间成本较高。

5.2 生成式 AI 验证

面向定理证明的大语言模型在不断发展，Chat-

GPT-4o 作为多模态大语言模型在处理复杂数学推导、逻辑问题等方面表现更加稳定，为辅助形式化验证提供了更为通用和交互友好的解决方案。DeepSeek-Prover-V2-671B 采用和 DeepSeek V3-0324 相同的架构，采用冷启动推理数据合成与强化学习相结合的训练策略，成功将非形式化数学推理与形式化证明能力融合为一体[24]。基于上述优势，本节选取这 2 种模型对基于二叉搜索树的 LCA 算法进行功能正确性验证，旨在评估生成式 AI 模型在自动化定理证明中的实际应用效果。基于文献 [24] 中的 CoT(chain-of-thought)模式，我们设计了 2 阶段的模型提示词。首先，在初始生成阶段，我们引导模型进行初步证明，其提示词如图 20 所示。其次，在交互修正阶段，我们将 Isabelle 控制台返回的错误信息反馈给模型，指导其迭代优化证明思路，直至生成高质量版本，此阶段的错误修复提示词如图 21 所示。

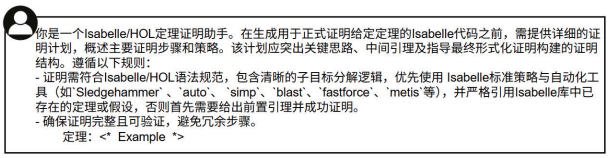


Fig. 20 Prompt in initial generation phase

图 20 初始生成阶段提示词

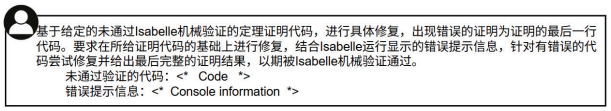


Fig. 21 Error fixing prompt in the interactive phase

图 21 交互阶段错误修复提示词

5.2.1 ChatGPT-4o 模型验证

在对 ChatGPT-4o 模型进行验证实验时，我们将 LCA 算法的完整建模代码作为输入，向 ChatGPT-4o 模型发出执行补全验证功能正确性定理的命令。该命令包含对数据结构、相关函数的定义以及核心定理 lca_bst_correct 的陈述，促使模型自动推导并生成完整的证明过程。在接收到该输入后，模型能够快速生成结构化的完整证明脚本，其验证过程如图 22 左图所示，存在 3 处未成功验证的部分。在最初生成的版本中，模型尝试通过 rule: *lca_bst*.induct 启动归纳



ChatGPT-4o生成的验证过程　　基于ChatGPT-4o的验证框架交互式验证成功的过程

Fig. 22 ChatGPT-4o model verification of LCA algorithm

图 22 LCA 算法 ChatGPT-4o 模型验证

证明过程，但由于其自动生成的 case (Leaf *p q*) 情形中存在类型匹配错误，验证器在此处抛出具体的反例并报错。值得注意的是，ChatGPT-4o 在多轮交互过程中能够识别出该错误的具体原因，并尝试修正其推理路径与归纳策略，最终生成一个结构基本完整的验证脚本。虽然在个别复杂分支处仍使用 sorry 占位符代替未完成的子目标，但整体验证框架的生成质量已达较高水平。

为全面评估验证框架的有效性，我们以 ChatGPT-4o 所生成的代码为基础，在 Isabelle 的交互式环境中补全剩余的证明内容，并最终成功完成了主定理 lca_bst_correct 的验证，完整验证过程如图 22 右图所示。所需构造的前置引理与交互式验证方法的大致相同，只额外增加了 1 条关于节点存在性前置引理的证明。ChatGPT-4o 模型将整个定理作为目标不进行拆分，需要构造 7 条自定义前置引理，验证代码共计 169 行。在“生成式 AI+交互式验证”工作模式中，模型输出后的人工作业环节主要包括：对模型生成的前置引理进行筛选与语义修正，对生成的证明脚本进行语法调整与证明命令重构，以及在 Isabelle 中进行交互验证与错误修复。实际统计结果表明，该模型生成的证明仍需大量人工参与，其中人工修改行数占最终验证成功总行数的 73.37%。

通过对比图 22 左图与图 22 右图可以看出，验证是否成功的关键在于辅助函数 *isin* 与 *is_ancestor* 所涉及的一系列前置引理的充分构造与调用。该类引理往往用于描述节点在树中的存在性、祖先关系的传递性等性质，而这些性质无法通过主定理本身直接得出，但却是主定理成功归纳推理、结构拆分以及分支展开的关键逻辑支撑。由于 *isin* 和 *is_ancestor* 函数的语义本身具有递归性和结构依赖性，相关引理的构造必须对其函数定义的递归展开模式、边界行为等有深入理解，大模型未能推导出其中的依赖关系，因此必须由开发者结合函数语义精心设计和导出。此外，在证明生成失败后，定理证明器返回的反馈多以类型冲突、子目标无法匹配等非自然语言呈现。模型对这类低层反馈的解析能力有限，难以据此定位问题根源并进行针对性修复，因而在多轮迭代修正后仍无法通过验证。

5.2.2 DeepSeek-Prover-V2-671B 模型验证

同样地，我们将 LCA 算法的完整建模代码作为输入，向 DeepSeek-Prover-V2-671B 模型发出执行补全验证功能正确性定理的命令，特别选择其 CoT 模式以生成可解释的证明过程[24]。

根据 CoT 模式的推理机制，模型在处理二叉搜索树结构下非空树情形时，自动生成了 6 条辅助引理，生成的辅助引理如图 23 所示。这些引理在形式上看似紧密关联于主定理的证明目标，涵盖了节点存在性、祖先关系及其在树结构中的传递性等多个方面。但分析发现，其中 lca_in_l 和 lca_in_r 引理在模型生成的具体证明过程中完全未被调用；而 is_ancestor_left、is_ancestor_right、is_ancestor_main 以及 is_ancestor_main2 引理虽在某些分支下被模型尝试调用，但将其移除后对整体证明过程并无实质性影响。更重要的是，所有这些自动生成的引理均未能覆盖主定理所依赖的关键前提条件，亦无法替代任何经过人工构造的有效前置引理。这表明模型虽具备生成辅助定理结构的能力，却未能正确识别出有助于归纳展开和逻辑化简的核心性质。因此，这些引理在证明中表现为冗余，无法在实际验证中发挥支撑作用，在缺乏目标驱动和语义约束的情境下，其生成的引理往往缺乏逻辑深度与语义针对性。

```
have lca_in_l: "x > max p q ⟹ isin (Node l x r) lca ⟹ isin l lca"
  sorry
have lca_in_r: "x < min p q ⟹ isin (Node l x r) lca ⟹ isin r lca"
  sorry
have is_ancestor_left: "x > max p q ⟹ is_ancestor (Node l x r) lca p = is_ancestor l lca p"
  sorry
have is_ancestor_right: "x < min p q ⟹ is_ancestor (Node l x r) lca p = is_ancestor r lca p"
  sorry
have is_ancestor_main: "x > max p q ⟹ is_ancestor (Node l x r) lca p"
  sorry
have is_ancestor_main2: "x < min p q ⟹ is_ancestor (Node l x r) lca p"
  sorry
```

Fig. 23 Model-generated auxiliary lemmas for non-empty trees

图 23 在非空树情况下模型生成的辅助引理

将冗余辅助引理删去后，模型生成的验证过程如图 24 左图所示，验证报错的地方尝试用 try 命令证明，无效则用 sorry 占位符取代模型无法验证成功的部分，共存在 3 处未成功验证的部分，可以看出整体上模型已能够构建出具有清晰逻辑结构的验证框架，并较为准确地划分了定理证明所需考虑的不同情况。这一结果在一定程度上展现出大语言模型在形式化验证任务中初步具备推导能力，尤其是在结构归纳与



DeepSeek-Prover-V2-671B生成的验证过程　　基于DeepSeek-Prover-V2-671B的验证框架交互式验证成功的过程

Fig. 24　DeepSeek-Prover-V2-671B model verification of LCA algorithm

图 24　LCA 算法 DeepSeek-Prover-V2-671B 模型验证

函数分析方面能够给出合理的分支判断和策略初始化。

对模型验证过程的进一步分析发现，其在处理递归结构中的具体分支推理时，尚未形成稳定有效的策略，导致证明任务在关键位置中断，无法完成整体推导。为深入评估其验证框架的可行性，我们以该模型生成的初始验证框架为基础，在交互式定理验证环境中对核心定理 lca_bst_correct 进行逐步补全和修改。证明过程中构造的前置引理与交互式验证方法构造的前置引理所需一致，最终获得完整、可通过验证器检查的形式化证明过程，如图 24 右图所示。验证过程中会将定理拆分成 3 个子目标进行推理验证，需构造 6 条自定义前置引理，验证代码共计 170 行，其中人工修改行数占最终验证成功总行数的 65.88%。通过对比分析，发现模型验证的主要失败原因是其概率生成机制限制了形式化推理以及在逻辑推导过程中未能主动引入一系列相关的前置引理，从而无法对复杂命题进行有效化简。lca_bst_correct 定理本身是一个结构复杂及涉及多个逻辑约束交织的复合性命题，其证明不仅需要树节点存在性的前提，还需要对祖先关系的传递性等性质进行精确刻画。模型基于概率生成机制自动生成证明步骤的过程中，并未意识到这些前置引理的必要性，导致在具体展开树结构或推理祖先关系时无法完成自动推导。

5.3　对比分析

交互式验证方法在形式化建模与定理证明过程中具有较高的严谨性与可信度。但验证过程高度依赖人工参与，证明脚本的编写及中间引理的构造均需研究者手动完成，整体工作量较大。在面对复杂数据结构或多层嵌套逻辑约束时，证明步骤往往呈指数级增长，人工推理负担显著加重，导致验证效率较低[25]。且不同开发者在证明策略选择和工具使用熟练程度方面存在差异，验证过程的时间成本和稳

定性难以保证。

大语言模型在形式化验证任务中表现出良好的交互能力和自我修正能力，能快速构建具备清晰逻辑结构的验证框架，体现其在定理自动化推理方面的潜力。但其在处理复杂数据结构和多条件逻辑推理任务时，失败原因可归纳为 3 点：1)概率生成机制限制了严格形式化推理。大模型具有概率性，缺乏完备证明保证[25]，主要基于语法相关性，通过概率预测生成证明，而非语义因果推理。因此在定理证明涉及不变式保持或复杂约束时，容易出现推理链条断裂或关键步骤缺失。2)缺乏辅助引理的构造能力。模型难以识别并构造必要的中间引理，无法有效组织完整证明结构，导致最终定理的证明过程不完整。3)对验证器低层反馈解析能力不足。返回的错误信息通常以类型冲突、反例信息、约束不满足等结构化形式呈现，而非自然语言描述。当前大模型对形式化证明代码这类低层反馈的语义理解与修复能力有限，难以将其有效转化为针对性的修改策略，从而在一定程度上制约了闭环优化流程的泛化能力[25]。

本文构建的函数式建模库以参数化结构模型为核心，覆盖多类搜索树结构，可降低建模负担与错误风险；同时，验证引理库系统化刻画验证通用操作语义，支撑算法验证自动化。二者契合搜索树算法的结构归纳特性，兼具广泛适用性与扩展能力，实现了算法的统一建模与低成本、高自动化验证，为搜索树类算法形式化验证提供了通用、可扩展的技术基础。

以 LCA 算法的功能正确性和动态维护 Top k 算法(插入操作)正确性为例，表 2 给出了各个验证方法 sorry 数、拆分的目标数、自定义引理条数、验证代码行数以及人工交互占比的对比。

Table 2 Comparison of Verification Methods

表 2 验证方法的对比

验证目标	对比项		ChatGPT-4o	DeepSeek-Prover-V2-671B(CoT)	交互式定理验证	ChatGPT-4o+交互式定理验证	DeepSeek-Prover-V2-671B(CoT)+交互式定理验证	本文验证方法
LCA 算法功能正确性验证	sorry 数		3	3	0	0	0	0
	拆分的目标数		0	3	2	0	3	3
	自定义引理条数		0	6	6	7	6	0
	验证代码行数	自定义引理行数	0	Failure	97	103	97	0
		主定理行数	Failure	Failure	81	66	73	29
	人工交互占比/%				88.76	73.37	65.88	17.24
动态维护 Top k 算法(插入操作)正确性验证	sorry 数		5	5	0	0	0	0
	拆分的目标数		1	1	0	1	1	2
	自定义引理条数		6	10	34	29	29	1
	验证代码行数	自定义引理行数	Failure	Failure	143	129	138	2
		主定理行数	Failure	Faliure	26	26	36	9
	人工交互占比/%				87.85	67.74	67.24	36.36

注：“Failure”表示验证未成功完成，验证代码行数不可统计。

表 2 中，在“人工交互”环节，本次实验共招募 9 名参与者，年龄均为 22~26 岁，5 名主修计算机科学与技术、3 名主修软件工程、1 名主修人工智能，其中 6 名男性与 3 名女性；在经验方面，3 名为初学者(约 1 年经验)，6 名为具有 1~3 年经验的程序员。所有参与者均使用 Isabelle/HOL 定理证明器，均在相同的建模函数、相同的定理目标以及相同的大模型(ChatGPT-4o、DeepSeek-Prover-V2-671B)生成框架下开展形式化证明任务。最终实验结果以 9 名参与者的数据平均值作为统计结果。需要指出的是，尽管所有参与者均基于相同的建模函数与验证目标开展定理证明工作，但由于证明方式(Isar 或 apply 风格)、证明选择策略以及对引理库熟悉程度的不同，不同开发者在相同验证任务下仍可能存在效率差异。因此，“人工交互占比”指标主要用于衡量验证自动化程度提升的相对趋势，而非对个体开发效率进行绝对评价。该指标能够反映方法在减少人工干预方面的整体效果，但其结果仍受到人员背景与经验差异的影响。

具体而言，LCA 算法正确性若仅依赖大模型生成，ChatGPT-4o 和 DeepSeek-Prover-V2-671B 模型二者均无法成功完成完整验证，存在 3 处 sorry 占位符。

交互式定理验证方法将定理拆分成 2 条子目标，需构造 6 条自定义引理，自定义引理验证代码 97 行，主定理需 81 行。大语言模型结合交互式验证定理时，ChatGPT-4o 未对定理进行拆分，而 DeepSeek-Prover-V2-671B 将定理拆分成 3 个子目标。二者验证成功分别需要自定义构造 7 条和 6 条引理，需要 103 行和 97 行代码，主定理验证需要 66 行和 73 行代码。这些前置引理均需手工额外构造并通过交互式定理证明补全，验证过程仍存在较高的人工依赖，人工交互占比分别为 73.37%、65.88%。而本文提出基于验证引理库的验证方法，将定理拆分成 3 条子目标，无需构造前置引理的前提下完成自动化证明，验证代码行数缩减至 29，仅在目标拆分时需要人工参与，人工交互占比为 17.24%。在动态维护 Top k 算法（插入操作）的验证中，仅依赖大模型进行验证无法成功。采用交互式定理验证方法时，需构造 34 条自定义引理，其中自定义引理验证代码 143 行，主定理 26 行。在“ChatGPT-4o+交互式定理验证”和“DeepSeek-Prover-V2-671B(COT)+交互式定理验证”中，模型将结构正确性拆分处理。二者验证成功都需自定义构造 29 条引理，验证成功分别需要 129 行和 138 行代码，主定理验证需要 26 行和 36 行代码，人工交互占比分别为 67.74%、67.24%。本文所提方法将定理拆分成 2 条子目标，构造查询操作正确性前置引理后可完成自动化证明，主定理验证代码仅需 9 行，人工交互占比为 36.36%。本文所提方法自动验证能力大幅提高，主要得益于验证引理库系统化地预先封装了算法依赖的关键逻辑性质与前置引理，从而避免了重复构造和逐条验证的高人工成本。

6 总结与展望

本文提出了 Isabelle 驱动的搜索树类算法建模及自动化验证方法。构建了涵盖多种搜索树结构的函数式建模库，统一数据结构抽象与通用操作模板。引入模块化建模策略，将数据结构定义、通用操作复用与关键逻辑实现解耦，提升建模的灵活性与可拓展性。同时，通过构建验证引理库与自动化分块验证策略，显著降低自动化验证的复杂度，增强了对复杂结构性质的支撑能力。最后通过代码生成机制，可自动生成语义一致、结构清晰的函数式可执行程序。以 LCA 算法和动态维护 Top k 算法为例，本方法较交互式验证方法代码行数和人工交互占比平均缩减 88.60%、61.51%，与“生成式 AI（ChatGPT-4o、DeepSeek-Prover-V2-671B）+交互式定理验证”方法相比，代码行数和人工交互占比分别平均缩减 88.09%、41.76%。实现对复杂数据结构算法的系统化、高度自动化验证，并提供生成正确函数式程序的可行路径。

在未来的工作中，我们将探索所提方法在并发数据结构中的扩展，以提升其在高性能与多线程场景下的适用性。此外，我们还将深入探索生成式 AI 在引理自动生成和验证引理库动态扩展方面的潜力，以降低人工构造成本并提升整体验证流程的自动化水平。

作者贡献声明：左正康和朱小影共同提出方法思路和实验方案；朱小影完成实验并撰写论文；游珍和王昌晶提出指导意见并修改论文。

参 考 文 献

[1] Shi Haihe, Lan Sunwen, Liu Riming, et al. Unified formal construction and Isabelle verification of the dynamic programming algorithms for biological sequence alignment[J]. Journal of Computer Research and Development, 2025, 62(1): 119−131(in Chinese)
(石海鹤, 蓝孙文, 刘日明, 等. 生物序列比对动态规划算法的统一形式化构造与 Isabelle 验证[J]. 计算机研究与发展, 2025, 62(1): 119−131)

[2] Vazou N, Seidel E L, Jhala R. LiquidHaskell: Experience with refinement types in the real world [C]//Proc of the 2014 ACM SIGPLAN Symp on Haskell. New York: ACM, 2014: 39−51

[3] Xu D N. Extended static checking for Haskell[C]//Proc of the 2006 ACM SIGPLAN Workshop on Haskell. New York: ACM, 2006: 48−59

[4] Hutton G. Programming in Haskell[M]. Cambridge, UK: Cambridge University Press, 2016

[5] Dageförde J C, Teegen F. Structured traversal of search trees in constraint-logic object-oriented programming[C]//Proc of Int Conf on Applications of Declarative Programming and Knowledge Management. Cham: Springer, 2019: 199−214

[6] Huet G, Kahn G, Paulin-Mohring C. The Coq proof assistant a tutorial[R/OL]. Rapport Technique, 1997, 178. [2025-11-02]. https://www.cs.cmu.edu/~emc/15414-s14/lecture/Tutorial.pdf

[7] Nipkow T, Klein G. Concrete Semantics: with Isabelle/HOL[M]. Berlin: Springer, 2021

[8] Nipkow T, Paulson L C, Wenzel M. Isabelle/HOL: A Proof Assistant for Higher-Order Logic[M/OL]. Springer Science & Business Media, 2021[2025-12-10]. https://isabelle.in.tum.de/website-Isabelle2021/dist/Isabelle2021/doc/tutorial.pdf

[9] Zuo Zhengkang, Huang Zhipeng, Huang Qing, et al. Functional modeling and mechanized verification of LLRB algorithm[J]. Journal of Software, 2024, 35(11): 5016−5039 (in Chinese)
(左正康, 黄志鹏, 黄菁, 等. LLRB 算法的函数式建模及其机械化验证[J]. 软件学报, 2024, 35(11): 5016−5039)

[10] Zuo Zhengkang, Liu Zengxin, Ke Yuhan, et al. Functional modeling and automatic verification of dynamic order statistic tree structures[J]. Journal of Software, 2025, 36(8): 3531−3553 (in Chinese)
(左正康, 刘增鑫, 柯雨含, 等. 动态顺序统计树类结构的函数式建模及其自动化验证[J]. 软件学报, 2025, 36(8): 3531−3553)
[11] Nipkow T. Automatic functional correctness proofs for functional search trees[C]//Proc of the 7th Int Conf on Interactive Theorem Proving. Cham: Springer, 2016: 307−322
[12] Guo Daya, Yang Dejian, Zhang Haowei, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning[J]. arXiv preprint arXiv: 2501.12948, 2025
[13] Dubey A, Jauhri A, Pandey A, et al. The Llama 3 herd of models[J]. arXiv preprints arXiv: 2407.21783, 2024
[14] Yang An, Li Anfeng, Yang Baosong, et al. Qwen3 technical report[J]. arXiv preprint arXiv: 2505.09388, 2025
[15] Dong Kefen, Ma Tengyu. Stp: Self-play LLM theorem provers with iterative conjecturing and proving[J]. arXiv preprint arXiv: 2502.00212, 2025
[16] Zhang Ziyin, Xu Jiahao, He Zhiwei, et al. DeepTheorem: Advancing LLM reasoning for theorem proving through natural language and reinforcement learning[J]. arXiv preprint arXiv: 2505.23754, 2025
[17] Xin Huajian, Guo Daya, Shao Zhihong, et al. DeepSeek-Prover: Advancing theorem proving in LLMs through large-scale synthetic data[J]. arXiv preprint arXiv: 2405.14333, 2024
[18] Mukherjee P, Delaware B. Towards automated verification of LLM-synthesized C programs[J]. arXiv preprint arXiv: 2410.14835, 2024
[19] Zhao Yongwang. Functional programming and proof[EB/OL]. 2021[2025-12-10]. https://www.yuque.com/zhaoyongwang/fpp/
[20] Back R. Invariant based programming: Basic approach and teaching experiences[J]. Formal Aspects of Computing, 2009, 21(3): 227−244
[21] Böhme S, Nipkow T. Sledgehammer: Judgement day[C]//Proc of the 5th Int Joint Conf of Automated Reasoning. Berlin: Springer, 2010: 107−121
[22] Sherriff M, Williams L. Tool support for estimating software reliability in Haskell programs[C]// Proc of IEEE Int Symp on Software Reliability Engineering. Piscataway, NJ: IEEE, 2004: 61−62
[23] Haftmann F, Bulwahn L, Nipkow T. Code generation from Isabelle/HOL theories[EB/OL]. 2025[2025-12-10]. https://isabelle.in.tum.de/doc/codegen.pdf
[24] Ren Z Z, Shao Zhihong, Song Junxiao, et al. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition[J]. arXiv preprint arXiv: 2504.21801, 2025
[25] Wen Cheng, Ma Zhi, Hu Junjie, et al. A review on software formal verification empowered by large language models[J/OL]. Journal of Software (in Chinese), 2026[2026-02-20]. http://www.jos.org.cn/1000-9825/7603.htm
(文成, 马智, 胡俊杰, 等. 大语言模型赋能软件形式化验证研究综述[J/OL]. 软件学报, 2026[2026-02-20]. http://www.jos.org.cn/1000-9825/7603.htm)



Zuo Zhengkang, born in 1980. PhD, professor, PhD supervisor. Distinguished member of CCF. His main research interest includes high-confidence and intelligent software.
左正康, 1980 年生。博士, 教授, 博士生导师。CCF 杰出会员。主要研究方向为高可信与智能化软件。



Zhu Xiaoying, born in 2001. Master candidate. Student member of CCF. Her main research interest includes formal method.
朱小影, 2001 年生。硕士研究生。CCF 学生会员。主要研究方向为形式化方法。



You Zhen, born in 1982. PhD, associate professor, master supervisor. Senior member of CCF. Her main research interests include formal methods, distributed virtual reality, and concurrent distributed computing.
游　珍, 1982 年生。博士, 副教授, 硕士生导师。CCF 高级会员。主要研究方向为形式化方法、分布式虚拟现实、并发分布式计算。



Wang Changjing, born in 1977. PhD, professor, PhD supervisor. Distinguished member of CCF. His main research interests include high-confidence software and intelligent software.
王昌晶, 1977 年生。博士, 教授, 博士生导师。CCF 杰出会员。主要研究方向为高可信软件、智能化软件。