软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(8):3531−3553 [doi: 10.13328/j.cnki.jos.007349] [CSTR: 32375.14.jos.007349]

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

动态顺序统计树类结构的函数式建模及其自动化验证*

左正康[1], 刘增鑫[1], 柯雨含[1], 游 珍[1,2], 王昌晶[1]

[1](江西师范大学 计算机信息工程学院, 江西 南昌 330022)

[2](网络化支撑软件国家国际科技合作基地 (江西师范大学), 江西 南昌 330022)

通信作者: 王昌晶, E-mail: wcj@jxnu.edu.cn

摘 要: 动态顺序统计树结构是一类融合了动态集合、顺序统计量以及搜索树结构特性的数据结构, 支持高效的数据检索操作, 广泛应用于数据库系统、内存管理和文件管理等领域. 然而, 当前工作侧重讨论结构不变性, 如平衡性, 而忽略了功能正确性的讨论. 且现有研究方法主要针对具体的算法程序进行手工推导或交互式机械化验证, 缺乏成熟且可靠的通用验证模式, 自动化水平较低. 为此, 设计动态顺序统计搜索树类结构的 Isabelle 函数式建模框架和自动化验证框架, 构建经过验证的通用验证引理库, 可以节省开发人员验证代码的时间和成本. 基于函数式建模框架, 选取不平衡的二叉搜索树、平衡的二叉搜索树 (以红黑树为代表) 和平衡的多叉搜索树 (以 2−3 树为代表) 作为实例化的案例来展示. 借助自动验证框架, 多个实例化案例可自动验证, 仅需要使用归纳法并调用一次 *auto* 方法或使用 *try* 命令即可, 为复杂数据结构算法功能和结构正确性的自动化验证提供了参考.

关键词: 动态顺序统计树; 搜索树; 函数式建模; 自动化验证; Isabelle 定理证明器

中图法分类号: TP311

中文引用格式: 左正康, 刘增鑫, 柯雨含, 游珍, 王昌晶. 动态顺序统计树类结构的函数式建模及其自动化验证. 软件学报, 2025, 36(8): 3531–3553. http://www.jos.org.cn/1000-9825/7349.htm

英文引用格式: Zuo ZK, Liu ZX, Ke YH, You Z, Wang CJ. Functional Modeling and Automatic Verification of Dynamic Order Statistic Tree Structures. Ruan Jian Xue Bao/Journal of Software, 2025, 36(8): 3531–3553 (in Chinese). http://www.jos.org.cn/1000-9825/7349.htm

Functional Modeling and Automatic Verification of Dynamic Order Statistic Tree Structures

ZUO Zheng-Kang[1], LIU Zeng-Xin[1], KE Yu-Han[1], YOU Zhen[1,2], WANG Chang-Jing[1]

[1](School of Computer Information Engineering, Jiangxi Normal University, Nanchang 330022, China)

[2](National-level International S & T Cooperation Base of Networked Supporting Software (Jiangxi Normal University), Nanchang 330022, China)

Abstract: Dynamic order statistic tree structures are a type of data structure that integrates the features of dynamic sets, order statistics, and search tree structures, supporting efficient data retrieval operations. These structures are widely used in fields such as database systems, memory management, and file management. However, current research primarily focuses on structural invariants, such as balance, while neglecting discussions on functional correctness. In addition, existing research methods mainly involve manual derivation or interactive mechanized verification for specific algorithms, lacking mature and reliable general verification frameworks and exhibiting a low level of automation. To address this, a functional modeling and automated verification framework for dynamic order statistic search tree structures, based on Isabelle, has been designed. A verified general lemma library is established to reduce the time and cost of code verification for developers. Using this functional modeling framework, unbalanced binary search trees, balanced binary search trees

* 基金项目: 国家自然科学基金 (62462036, 62462037); 江西省自然科学基金面上项目 (20232BAB202010, 20212BAB202018); 江西省教育厅科技重点项目 (GJJ210307, GJJ2200302, GJJ210333); 江西省主要学科学术与技术带头人培养项目 (20232BCJ22013)
本文由“形式化方法与应用”专题特约编辑陈明帅研究员、田聪教授、熊英飞副教授推荐.
收稿时间: 2024-08-26; 修改时间: 2024-10-14; 采用时间: 2024-11-26; jos 在线出版时间: 2024-12-10
CNKI 网络首发时间: 2025-04-03

(exemplified by red-black trees), and balanced multi-way search trees (exemplified by 2–3 trees) are selected as instantiated cases for demonstration. With the help of the automated verification framework, multiple instantiated cases can be automatically verified by simply using induction and invoking the auto method once, or by using the try command. This provides a reference for automated verification of functional and structural correctness in complex data structure algorithms.

Key words: dynamic order statistic tree; search tree; functional modeling; automated verification; Isabelle theorem prover

动态顺序统计树结构是一类融合了动态集合、顺序统计量以及搜索树结构特性的数据结构. 除了基本的动态集合操作外, 还支持快速的数据检索、高效的插入和删除, 且能在 $O(\log n)$ 时间内实现顺序统计操作[1]. 其中, 动态集合是指由算法操作的数据集在整个过程中能增大、缩小或发生其他变化[2]. 顺序统计量是指在一组统计样本中按照大小顺序排列后的特定位置的值, 其对应的顺序统计操作主要包括两种, 一种是在一组数据中找到第 k 小 (或第 k 大) 元素的选择操作, 另一种是给出一个指定元素在数据集的全序位置的求秩操作[3,4]. 动态顺序统计树类结构能在具体的搜索树结构中通过扩充树的大小信息实现, 广泛应用于数据库系统、内存管理和文件管理等领域. 然而, 当前搜索树类结构的相关工作侧重讨论结构特性, 如平衡性, 而忽略了功能正确性的重要性[5]. 这可能使得该结构及对应算法出现未预料到的错误, 增加了后期调试和维护的难度. 因此, 对动态顺序统计树等搜索树类结构进行功能正确性验证, 是确保算法可靠性、安全性和有效性的重要措施.

在建模方面, 建立函数式建模框架可以提供一个结构化的开发环境, 能将数据结构对应的算法函数功能清晰地呈现出来, 有助于开发人员更快速、更高效地进行开发工作. 而在对动态顺序统计树类结构进行函数式建模的过程中, 需要综合考虑选用实例化的搜索树结构、具体的应用场景以及它们在实际应用中的功能. 这就意味着需要在不同的复杂结构中找到其中的共性, 制定统一的建模策略和操作模式, 需要更多的创造性思维和技巧. 在验证方面, 形式化定理证明是保证程序正确性的有效途径, Isabelle/HOL[6,7]是当前被广泛使用的定理证明器. 然而, 大多数在 Isabelle 上对搜索树类结构的验证, 都是针对具体的算法程序进行交互式机械化验证. 一方面, 其验证的可复用性较低, 验证对象或场景发生变化时需要重新设计验证方案. 另一方面, 其自动化的程度低, 需要更多的人为干预, 验证过程需要耗费大量时间和人力成本. 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, 可以在保证验证准确性的前提下, 尽可能地减少人为参与. 因此, 研究函数式建模框架和自动化验证框架可以简化建模和验证流程, 有效地减少人工干预的需求, 从而节省开发人员验证代码的时间和成本.

文献 [5,8] 分别介绍了在 Isabelle 中实现自动验证的两种方式: 一是通过简单的归纳法或分情况讨论, 然后调用一次 *auto* 方法得到自动化证明[5]; 二是通过 *try* 命令, 调用 Sledgehammer[8]等组件将当前命题翻译为一阶逻辑, 然后搜索引理库中的引理探索证明路径, 自动生成证明. 两种方式均采用一组固定的基本引理以及关于辅助函数的引理作为参数. 这意味着在 Isabelle 中实现自动验证还需要设计通用的验证引理库. 一方面, 引理库整体的设计思想应注重简化问题复杂性, 将复杂问题逐步分解为更小的子问题, 使自动化证明组件能够高效处理. 另一方面, 引理库中辅助引理的设计应考虑可靠性和通用性, 保证辅助引理正确性的同时, 确保其能应对不同场景下的证明需求.

本文旨在 Isabelle 中设计动态顺序统计树类结构的函数式建模和自动化验证框架, 并将该建模框架实例化为不同的搜索树类结构, 借助自动化验证框架自动验证其中函数的功能正确性. 本文的技术路线图如图 1 所示. 由此, 本文工作的主要贡献点总结如下.

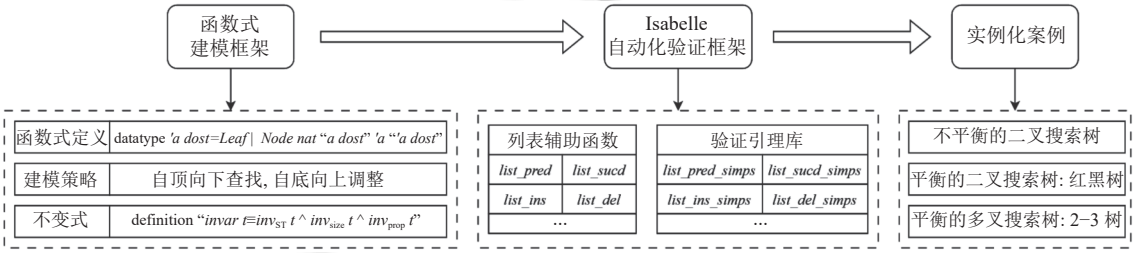


图 1 技术路线图

(1) 综合考虑不同搜索树结构的性质和操作, 将动态集合和顺序统计量的相关操作融入其中, 添加了额外的高效数据检索操作. 基于此, 建立了动态顺序统计树类结构的函数式建模框架, 包含其函数式定义、建模策略和不变式, 简化了建模流程;

(2) 提出了动态顺序统计树类结构相关算法的自动化验证框架, 使用同态的列表模拟动态顺序统计树类结构的 10 种操作, 设计并验证了列表辅助函数, 借助列表辅助函数构建了通用且可复用的验证引理库, 可自动化验证动态顺序统计树相关算法的功能正确性;

(3) 将动态顺序统计树类结构的函数式建模框架分别实例化为不平衡的二叉搜索树、红黑树和 2−3 树, 实例化得到的数据结构具有动态顺序统计树的高效检索特性. 本文所提的自动化验证框架除了适用现有工作已经得到验证的搜索树操作 (查找、插入和删除), 还涵盖了动态集合操作 (如前继和后继等) 和顺序统计操作 (如选择和求秩等).

本文第 1 节是对相关工作进行介绍. 第 2 节详细介绍动态顺序统计树类结构的函数式建模框架. 第 3 节介绍动态顺序统计树类结构的 Isabelle 自动化验证框架. 第 4 节将建模框架实例化为不同的具体搜索树结构, 并借助自动化验证框架对其正确性进行自动验证. 第 5 节进一步讨论自动化验证框架的适用范围和效果对比. 第 6 节是对全文的总结以及未来的工作展望.

1 相关工作

在搜索树类结构的算法领域, 研究者们致力于开发高效、可扩展且具有良好性能的算法, 以应对不同应用场景下的挑战. 在本节中, 将介绍与搜索树类结构相关的研究, 主要从具体搜索树结构的建模及验证, 和搜索树类结构的建模及验证框架两个方面进行介绍.

在具体搜索树结构建模及验证的发展中, 研究者们提出了各种各样的算法, 并通过手工推导或交互式机械化验证的方式确保正确性. 文献 [9] 将快速排序思想成功应用到二叉搜索树中, 形式化定义了随机二叉搜索树的结构, 并对其高度、结构不变式以及对搜索和插入操作的影响进行了讨论和验证. 文献 [10] 验证了 Braun 树基本操作的正确性, 并验证了从列表到 Braun 树的线性时间转换函数, 为 Braun 树的进一步研究和应用提供了新的思路. 文献 [11] 提出了根平衡树的概念, 并在函数式环境中验证了其摊销复杂度, 同时提供了无对数代码的实现以及与传统树结构的竞争性评估. 文献 [12] 提供了 splay 树的可执行结构, 并验证了其基本性质及摊销复杂度. 文献 [13] 提出并验证了 2−3 手指树结构及性质, 支持在均摊的恒定时间内访问列表的两端. 文献 [14] 确定了原始权重平衡树算法中插入和删除旋转参数的确切有效范围, 并使用生成反例树的有效算法证明了其完备性. 文献 [15] 利用集合的抽象数据类型高效实现并证明了 van Emde Boas 树相关函数的正确性, 并从证明的可信基础中消除了人工定义的时序函数. 总之, 尽管研究者们在具体搜索树结构的建模及验证方面取得了显著进展, 但现有研究主要针对特定的单一算法程序进行建模及验证, 缺乏成熟且可靠的通用建模及验证模式, 可复用性较低.

在搜索树类结构的建模及验证框架方面, 文献 [16] 通过探究二叉搜索树类算法之间共性, 建立了二叉搜索树类算法的函数式建模框架, 用区域 (*locale*) 刻画二叉搜索树类结构插入和删除高阶泛化函数, 在对具体的二叉搜索树变体结构 (如 AVL 树、RBs 等) 进行建模时, 可进行相应的实例化, 并对其附加性质的验证进行细化, 给出了具体化的验证方案. 不过, 该建模框架未对多叉的搜索树类结构进行讨论, 且其验证方案仍然停留在交互式机械化验证的层面, 自动化水平较低. 文献 [5] 提出了一种用于验证搜索树类结构功能正确性的自动化验证框架, 借助一个小型引理库, 对不同搜索树结构的功能正确性进行自动化证明, 展示了该框架的有效性和广泛适用性, 并讨论了从集合到映射的泛化过程. 然而, 该框架仅对搜索树类结构的查找、插入和删除这 3 种基本操作进行了自动化验证, 不适用于动态集合和顺序统计量相关算法.

本文提出了一个针对动态顺序统计树类结构的 Isabelle 函数式建模和自动化验证框架, 涵盖了动态集合、顺序统计和搜索树等 10 种常用操作, 构建了完善的验证引理库, 并给出了具体的验证方法. 进一步选取了不平衡的二叉搜索树、平衡的二叉搜索树 (红黑树) 和平衡的多叉搜索树 (2−3 树) 为实例, 检验了建模和自动化验证框架

的适用性和有效性.

2 动态顺序统计树类结构的函数式建模框架

为了简化建模流程, 提高代码的可重用性, 本文建立了动态顺序统计树类结构的函数式建模框架, 包括其函数式定义、建模策略和不变式. 本节讨论了动态顺序统计树类结构的函数式定义, 设计了统一的建模策略. 对于其对应的选择、求秩等操作, 刻画了通用的高阶复合函数, 并对这些操作算法的不变式进行了规范. 此外, 本节还讨论了如何在确保整体操作的时间复杂度不变的情况下, 添加额外的高效数据检索操作.

2.1 动态顺序统计树类结构的函数式定义

搜索树类结构允许根据键值进行元素搜索, 能够快速检索、插入和删除元素, 可用于实现动态集合的相关操作[17]. 动态顺序统计树类结构可以在搜索树的基础上, 通过添加树的大小信息实现. 对于二叉型动态顺序统计树类结构, 可以将其数据类型递归地定义为:

$$\text{datatype } 'a\ dost = Leaf \mid Node\ nat\ \text{“}'a\ dost\text{”}\ 'a\ \text{“}'a\ dost\text{”},$$

其中, $'a$ 是抽象的, 可以根据需要实例化为不同的具体的类型. *Leaf* 表示叶子节点 (即节点为空), *Node* 表示非叶子节点 (即节点非空), 其 4 个参数分别对应树的大小、左子树、根节点和右子树信息. 通过这种定义方式, 该结构不仅能实现搜索树类结构原有的基本操作, 还能高效实现动态集合和顺序统计量相关操作.

上述定义的数据类型是抽象的, 在不同的应用场景中, 可以根据需要将其实例化为不同的类型, 得到相对复杂的结构. 例如, 将其实例化为动态顺序统计红黑树或动态顺序统计 2−3 树等, 可应对实际应用场景. 在实例化到不同的搜索树结构的过程中, 可能会出现含有其他附加信息的情况, 如红黑树的颜色信息等, 统称为 *extra_prop*. 可使用关键字 type_synonym 来实现继承数据类型 *dost* 的基础上扩充额外的信息[7]. 基于此, 可以定义数据类型 *dost* 的类型同义词 $dost_{\text{syn}}$, 如下所示:

$$\text{type\_synonym } 'a\ dost_{\text{syn}} = \text{“}('a \times extra\_prop)\ dost\text{”}.$$

基于上述定义的数据类型, 可以定义函数 *tsize* 用于获取动态顺序统计树结构中树的大小信息. 如下所示.

```
//获取树的大小信息
fun tsize :: "'a dost ⇒ nat" where
"tsize Leaf = 0"|"tsize (Noden___) = n"
```

这个函数需要升级常规的搜索树操作来保证获取信息的准确性. 函数 *size*2 用于计算任一层根节点对应的树的大小. 基于 *size*2 函数, 可定义函数 *szNode* 更新节点对应的树的大小信息, 并将其代替原始的 *Node*. 如此做, 在对树中的节点信息进行更新时, 树的大小信息也会同步更改. 这些函数的正确执行依赖于建模策略和不变式的构建, 在后文中会对其进行详细的介绍.

```
//计算树的大小信息
definition "size2 l r ≡ tsize l + tsize r + 1"
//更新节点对应的树的大小信息
definition "szNode l a r ≡ Node (size2 l r) l a r"
```

需要说明的是, 函数 *tsize*、*size*2 和 *szNode* 都能在运行时间为 $O(1)$ 的情况下实现. 所以, 将 *Node* 转换为 *szNode* 不会影响整体的时间复杂度. 因此, 在动态顺序统计树类结构中关于搜索树的基本操作, 与底层搜索树操作具有相同的复杂度.

在上述定义的基础上, 我们给出如下缩写描述:

$$\begin{cases} Leaf \equiv <> \\ Node\ n\ l\ a\ r \equiv < n, l, a, r > \end{cases}.$$

对于多叉型动态顺序统计树类结构, 可按照相似的思路对其结构进行递归定义. 以每个非叶子节点都有 2 个或 3 个子节点的 2−3 树[1]为例, 叶子节点和 2 个子节点的情况可以沿用二叉型动态顺序统计树类结构的 *Leaf* 和 *Node* 定义. 而对于 3 个节点的情况, 可按照定义 2 个子节点情况的相似思路, 将其定义为<*n*, *l*, *a*, *m*, *b*, *r*>. 其中, *n* 同样表示树的大小信息, *a* 和 *b* 表示根节点上的两个关键字信息, *l*、*m* 和 *r* 分别表示左、中和右子树信息. 其对应的操作函数只需要按照同样的思路, 额外处理 3 个节点的情况即可. 本文的第 4 节给出了多叉型结构的典型代表 2−3 树的具体建模实现思路. 在本文的剩余部分, 将主要以二叉型的动态顺序统计树类结构为基准进行阐述. 对于多叉型的结构, 在实际建模时可遵循类似的思路补充额外的多节点情况.

2.2 动态顺序统计树类结构的函数式建模策略

动态顺序统计树类结构可以在搜索树结构中扩充树的大小信息实现, 因此其仍保留了搜索树性质. 结合树形结构来看, 搜索树性质可概括为: 左子树的节点键值均小于根节点, 右子树的节点键值均大于根节点. 基于搜索树的这一性质, 本文提出了标准化、统一的函数式建模策略, 即自顶向下查找, 自底向上调整, 如图 2 所示.

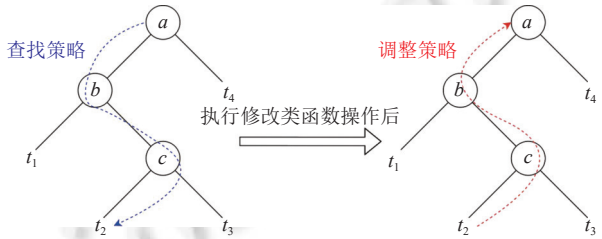


图 2 统一的函数式建模策略

对于不改变树中节点信息的函数操作, 统称为检索类函数, 如 *isin*、*select* 等, 采取查找策略. 检索类函数操作的核心在于找到相应的节点信息. 根据查找策略, 在寻找节点时, 需要从根节点出发, 根据根节点与目标节点的关键字大小关系, 选择向左或向右的路径, 自顶向下递归地在其子树中寻找目标节点. 在寻找的过程中, 可以根据需要执行一些前置处理操作, 如汇总树的大小信息, 以满足不同函数的实现需求. 在寻找到目标节点的位置后, 根据函数所需实现的功能处理最后的信息并返回最终的结果.

对于会改变树中节点信息的函数操作, 统称为修改类函数, 如 *insert*、*delete* 等, 结合使用查找策略和调整策略. 修改类函数操作的核心在于调整整棵树, 以维持整体的平衡性. 利用查找策略, 可以找到目标节点的位置. 在找到目标节点的位置之后, 在对其进行处理时, 可能会破坏树的整体平衡性. 此时, 需要借助调整策略对树的整体进行调整. 由于修改类函数并未对查找路径以外的节点位置进行操作, 所以在调整过程中只需对查找路径上的节点进行调整操作即可. 根据调整策略, 在处理完最后的信息后, 还需沿原路径返回并执行相应的调整操作. 换言之, 调整策略从目标节点的位置出发, 自底向上地执行调整操作, 调整到根节点时结束, 返回调整后得到的树.

为了方便叙述, 在本节中将建模策略分成了两步操作. 实际上, 利用递归的特性, 这两步操作在函数式建模中可以用一步递归完成. 这一点在后文的具体建模中会有更清晰的体现.

根据函数式建模策略, 可以建立动态顺序统计树类结构相关操作的高阶复合函数, 如下所示.

```
fun < *funcName* >:: " < *targetType* > ⇒ 'a dost ⇒ < *resultType* > " where
  " < *funcName* > x Leaf = [*adjustLeaf*] < *resultLeaf* > "|
  " < *funcName* > x (Node n l a r) =
        ([ * preOperation*] case < *compOperation* > of
            LT ⇒ [*adjustLT*] < *resultLT* > |
            GT ⇒ [*adjustGT*] < *resultGT* > |
            EQ ⇒ [*adjustEQ*] < *resultEQ* >)"
```

其中, *'a* 类型对应的元素需满足全序关系 (后文也是如此), 符号<* *> 表示在对具体的函数建模过程中, 需根据函数的实际功能填充的部分. 符号 [* *] 则表示可选项.

依据函数式建模策略可以推知, 通过目标节点信息和树的信息, 可以确定整个函数的操作路径. 所以, 在高阶复合函数的函数类型中, 前两个参数分别表示目标节点 (*targetType*) 和动态顺序统计树的数据类型 (*'a dost*), 以此来获取对应的信息并确定函数的操作路径. 第 3 个参数表示最后的结果类型 (*resultType*), 对于检索类函数, 对应检索得到的信息; 对于修改类函数, 则对应调整后得到的动态顺序统计树.

在高阶复合函数的函数体中, 对于树为空的情况, 只需要根据函数功能返回对应的结果 (*resultLeaf*) 即可. 而对于树不为空的情况的操作则更为复杂些. 首先, 需借助查找策略, 比较目标节点和当前节点的大小关系 (*compOperation*). 然后, 根据比较的结果确定需要执行的函数操作. 若比较结果为小于 (*LT*) 或大于 (*GT*), 则递归地在其左子树或右子树上寻找目标节点的位置 (分别对应 *resultLT* 和 *resultGT*). 若为相等 (*EQ*), 则说明查找成功, 执行相应的函数操作对目标节点进行处理, 返回处理后的结果 (*resultEQ*). 在查找的过程中, 可以根据需要补充一些前置操作 (*preOperation*) 对当前的信息进行处理. 对于修改类函数操作, 则还需要在返回最终结果前, 依据调整策略进行平衡调整 (*adjustLeaf*、*adjustLT*、*adjustGT* 和 *adjustEQ*).

2.3 动态顺序统计树类结构的不变式

不变式是指程序在执行过程中必须遵守的逻辑规则[18]. 在对动态顺序统计树类结构对应的算法进行建模时, 需要保证算法执行前后得到的动态顺序统计树均满足不变式. 因此, 不变式是函数式建模的一个标准, 当对修改类函数操作进行建模时, 其调整操作得到的结果需满足不变式.

动态顺序统计树结构的性质可被定义如下.

(1) 搜索树性质不变式 (简称 inv_{ST}), 即中序遍历节点的键值按照线性升序的方式排序[17]. 函数 *inorder* 按照中序遍历的顺序将动态顺序统计树中的节点转换为列表. 得到转换后的列表之后, 借助 Isabelle 中的函数 *sorted* 可判断该列表是否按升序排列. inv_{ST} 可定义为这两个函数的组合, 如下所示.

```
//将动态顺序统计树转换为列表 (针对dost类型)
fun inorder :: "'a dost ⇒ 'a list" where
"inorder <>= []"|
"inorder < _,l,a,r >= inorder l @ a # inorder r"
//搜索树性质不变式
definition "inv_ST t ≡ sorted (inorder t)"
```

inv_{ST} 的实现需要获取树中节点的键值信息, 而不依赖其他额外的信息. 对于扩充了额外信息的 $dost_{syn}$ 类型, 其节点中不仅包含键值信息, 还有其他额外信息. 因此, 在 $dost_{syn}$ 类型中实现 inv_{ST} 时, 需要从节点中提取出键值信息, 即将上述的 *inorder* 函数替换为 *inorder*2 函数, 如下所示. 可以发现, 相较于 *inorder* 函数, *inorder*2 函数仅在处理过程中额外添加了一步忽略额外信息的操作. 类似地, 对于其他同样不依赖额外信息的 $dost_{syn}$ 类型函数, 实现思路与 *dost* 类型一致, 只需要忽略额外信息即可.

```
//将动态顺序统计树转换为列表 (针对dost_syn类型)
fun inorder2 :: "'a dost_syn ⇒ 'a list" where
"inorder2 <>= []"|
"inorder2 < _,l,(a, _),r >= inorder l @ a # inorder r"
```

(2) 树的大小不变式 (简称 inv_{size}), 即树中的每个 *Node* 节点中的 *nat* 类型对应的信息, 都应正确存储对应的树的大小值. inv_{size} 可通过递归实现, 即需保证根节点中存储的树的大小值是正确的, 且其左、右子树对应的节点也同样正确. 基于 *size*2 函数, 可对 inv_{size} 进行函数式建模, 其定义如下.

```
//树的大小不变式
fun inv_size :: "'a dost ⇒ bool" where
"inv_size <>= True"|
"inv_size < n,l,a,r >= (inv_size l ∧ inv_size r ∧ (size2 l r = n))"
```

(3) 附加性质不变式 (统称 inv_{prop}), 这是一个可选项. 当实例化所用到的具体搜索树结构, 需要维持附加性质不变时 (如颜色等), 加上相应的附加性质.

总而言之, 上述不变式 (1)−(3) 统称为动态顺序统计树类结构的不变式 (简称 *invar*), 其定义如下.

```
//动态顺序统计树类结构的不变式
definition "invar t ≡ inv_ST t ∧ inv_size t ∧ inv_prop t"
```

3 动态顺序统计树类结构的 Isabelle 自动化验证框架

为了更高效地验证动态顺序统计树类结构相关算法函数的正确性, 减少开发人员验证代码的时间和成本, 本节构建了动态顺序统计树类结构在 Isabelle 中的自动化验证框架, 如图 3 所示. 其设计理念为用列表模拟动态顺序统计树类结构的 10 种操作, 设计对应的列表辅助函数, 并验证其正确性. 借助列表辅助函数可构建通用的验证引理集, 合并 10 种操作对应的验证引理集, 组成验证引理库. 基于验证引理库, 动态顺序统计树类结构的 10 种操作的功能正确性均能实现自动化验证. 第 3.1 节阐述了设计自动化验证框架的理论基础, 第 3.2 节设计并验证了列表辅助函数, 第 3.3 节构建了通用的验证引理库.

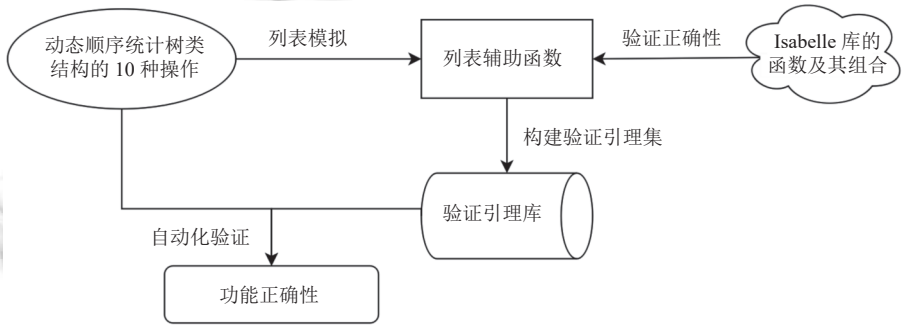


图 3 动态顺序统计树类结构的自动化验证框架结构

3.1 自动化验证框架的理论基础

实现自动化验证需要简化问题的复杂性, 将复杂的问题逐步拆解为 Isabelle 自动化证明组件能高效处理的形式. 具体函数的实现可以用同态的抽象函数来代替证明函数的正确性, 这一过程已被证明是正确的[19]. 使用同态的抽象函数代替具体函数进行函数式验证, 可以将复杂的问题证明分解为更为简单的形式, 从而简化证明过程[5]. 对于动态顺序统计树类结构, 在验证其操作算法时, 涉及的情况较多, 自动化证明组件难以应付. 而对于列表这类线性结构, 其结构的定义契合递归的思想, 对应函数的正确性验证较为简单. 如果能用列表模拟动态顺序统计树的函数操作, 并将列表作为中介去验证动态顺序统计树的函数正确性, 则可以简化验证过程, 进一步实现自动化验证.

这一方法的关键在于实现将动态顺序统计树映射到列表的映射函数 *Abs* :: *'t* ⇒ *'a list*. 并验证 *Abs* 是同态的, 即对于动态顺序统计树类结构中所有需要实现的函数操作 (统称为 *tree_fun*), 及其列表辅助函数操作 (统称为 *list_fun*), 满足以下关系[20]:

$$invar\ t \Rightarrow Abs(tree\_fun(t)) = list\_fun(Abs(t)).$$

函数 *inorder* 能够将动态顺序统计树中的节点按照中序遍历的顺序映射到列表中. 因此, 映射函数 *Abs* 可以通

过函数 *inorder* 来实现.

对于 *Abs* 同态性的验证, 一方面, 需要额外定义列表辅助函数模拟动态顺序统计树函数操作, 并保证列表辅助函数的正确性 (第 3.2 节); 另一方面, 需要针对列表辅助函数设计通用的验证引理集, 形成通用的验证引理库, 以应对不同场景下的需求 (第 3.3 节).

3.2 自动化验证框架的列表辅助函数

动态顺序统计树类结构的种类繁多, 但每个具体的动态顺序统计树都至少提供 10 种操作. 我们在 Isabelle 中用区域 (*locale*) 刻画了这些基本操作, 如图 4 所示. 其中, *'t* 类型可以用 *'a dost* 或 *'a* $dost_{syn}$ 代替. 对于树为空的情况 (*empty*), 其正确性证明是简单的, 本文将忽略对其的讨论.

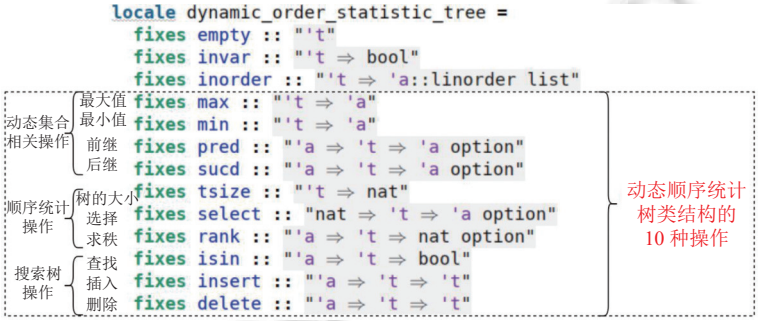


图 4 动态顺序统计树类结构的 10 种操作

对于函数 *tsize*, 在第 2 节中讨论了其特殊实现方式. 基于这种实现方式及建模策略, 函数 *tsize* 的正确性验证也变得更为简单, 只需要通过归纳法并调用一次 *auto* 方法即可得证, 无需构造额外的列表辅助函数及引理集.

而对于其他操作, 我们设计相应的列表辅助函数进行模拟, 以达到简化问题的目的. 借助 Isabelle 源码库中已经得到验证的函数及其组合, 可以验证列表辅助函数的正确性. 具体如下.

(1) *list_max* 和 *list_min*: 查找有序列表中的最大值和最小值.

如图 5 所示, 对于动态顺序统计树类结构中求最大值和最小值的函数 *max* 和 *min*, 设计了 *list_max* 和 *list_min* 模拟. 其中, *list_max* 通过遍历有序列表至表尾, 获取最大值, 而 *list_min* 则通过提取有序列表的表头元素, 获取最小值. 这两个函数的正确性证明可以分别借助 Isabelle 集合 (*set*) 中的 *Max* 和 *Min* 函数辅助验证, 使用归纳法并调用一次 *auto* 方法就能得到证明.

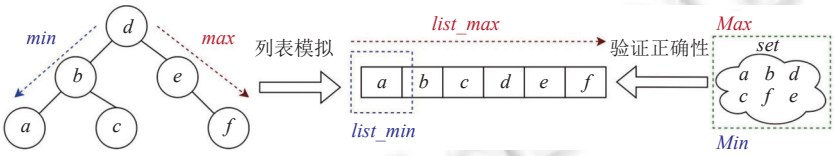


图 5 *list_max* 和 *list_min* 的模拟操作及验证

(2) *list_pred* 和 *list_sucd*: 查找在有序列表中, 指定元素的前继和后继元素.

一个元素的前继元素是指树中小于该元素的最大元素, 后继元素则是指大于该元素的最小元素. 在动态顺序统计树类结构中, 求前继元素的 *pred* 函数和求后继元素的 *sucd* 函数的实现均依赖于搜索树性质.

如图 6 所示, 以 *pred* 函数为例, 一般来说, 元素 *x* 左子树中的最大元素即为元素 *x* 的前继元素. 若元素 *x* 的左子树为空, 则需要向上查找其祖先节点以确定前继元素. 结合树形结构来看, 即沿着元素 *x* 的父节点向上查找, 直到找到一个元素 *y*, 元素 *y* 对应结点的右子树包含元素 *x*. 而 *sucd* 函数的实现思路则相反, 找元素 *x* 右子树中的最小元素或找祖先节点中左子树含有元素 *x* 的节点元素, 即为元素 *x* 的后继元素. 在实际的建模场景中, 两个函数的实现均需考虑较多的情况, 其正确性验证较为复杂, 常规的自动化证明组件难以应付.

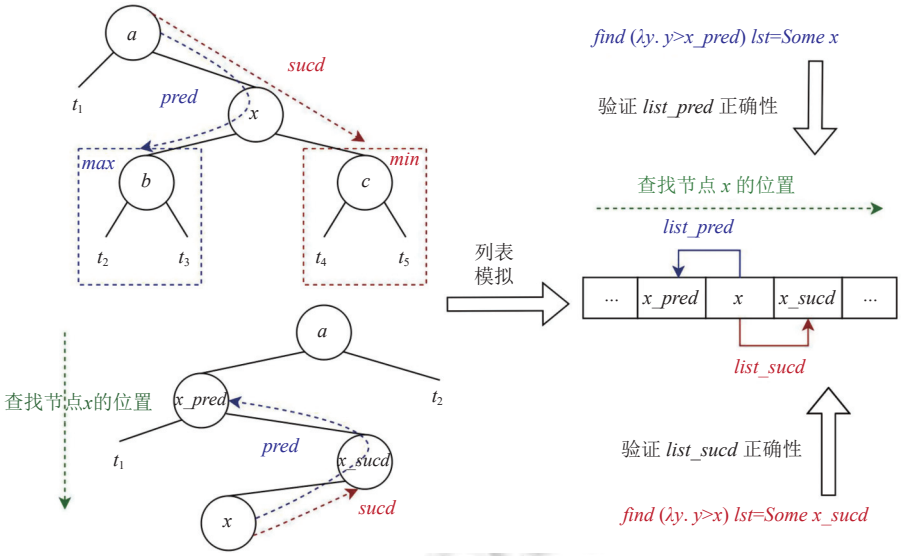


图 6 *list_pred* 和 *list_sucd* 的模拟操作及验证

如果有序列表 *lst* 中第 1 个大于元素 *x* 的元素是 *y*, 那么元素 *x* 是元素 *y* 的前继元素, 元素 *y* 则是元素 *x* 的后继元素. 在有序列表中实现 *list_pred* 和 *list_sucd* 只需要先查找节点 *x* 的位置, 然后找到节点 *x* 前一个位置的元素和后一个位置的元素即可, 涉及的情况较少. 因此, 用 *list_pred* 和 *list_sucd* 来模拟 *pred* 和 *sucd* 函数操作, 可以降低问题的复杂度, 便于自动化证明组件处理.

Isabelle 中的 *find* 函数可以按条件查询列表中满足条件的第 1 个元素. 借助 *find* 函数和 λ 函数组合可验证 *list_pred* 和 *list_sucd* 的正确性.

(3) *list_sel* 和 *list_rank*: 前者表示给定一个具体数值 *k*, 查找有序列表中第 *k* 小的元素. 后者则表示给定一个元素 *x*, 计算该元素为有序列表中的第几小的元素.

如图 7 所示, 选择函数 *select* 实现在动态顺序统计树类结构中查找具有给定次序的元素的功能, 即查找第 *k* 小的元素. *select* 函数功能的实现依赖于树的节点大小信息. 通过比较给定值 *k* 与左子树的大小信息, 来判断往树的左子树或者右子树中查找元素. 如果往右子树中查找元素, 则需将 *k* 的值依次减去左子树的大小值和根节点单个节点的大小值 (值为 1), 再将得出的值作为新一轮递归的输入. 求秩函数 *rank* 实现确定一个元素的次序的功能, 即给定一个元素 *x*, 求该元素是树中第几小的元素. 根据目标节点与当前根节点的大小关系, 判断查找路径的方向, 同时累计树的大小信息, 表示比目标节点元素键值更小的节点个数. 当找到目标元素 *x* 后, 累计的树的大小信息为最终的结果. 两个函数的实现都需要在常规的递归查找过程中添加额外的处理树的大小信息操作, 增加了额外的证明义务. 需要将问题简化, 才能实现自动化验证.

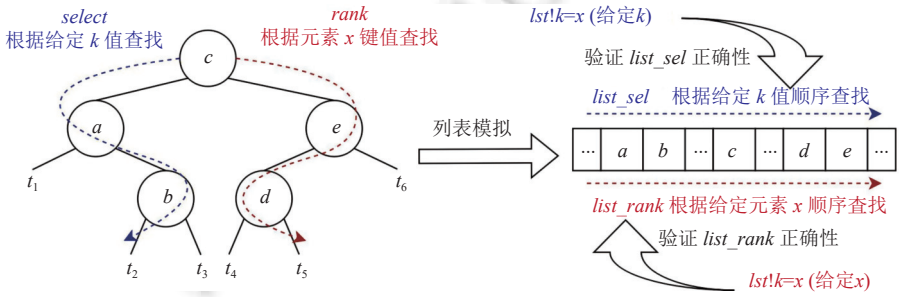


图 7 *list_sel* 和 *list_rank* 的模拟操作及验证

在有序列表中 *list_sel* 和 *list_rank* 函数的实现均可通过顺序查找完成. 在查找过程中, *list_sel* 函数通过递减给定的 *k* 值查找, 当 *k* 值为 0 时, 得到第 *k* 小的元素; *list_rank* 函数根据元素 *x* 的键值查找, 找到元素 *x* 后返回累计遍历的元素个数. 两个函数操作都是线性的, 简化了复杂操作, 自动化证明组件可以应付.

Isabelle 中的 *nth* 函数 (简写为符号!) 可以获取列表中的指定索引位置的元素. 利用这个函数, 可以验证 *list_sel* 和 *list_rank* 函数的正确性.

(4) *list_isin*、*list_ins* 和 *list_del*: 分别表示在有序列表中查找、插入和删除元素.

如图 8 所示, 动态顺序统计树类结构中的 *isin*、*insert* 和 *delete* 函数的实现都是需要先定位到元素 *x* 的位置. *isin* 函数在查找后直接返回结果即可. *insert* 和 *delete* 函数则需要在查找之后执行相应的插入和删除操作. 由于插入和删除操作可能会改变树的节点大小信息, 且会破坏树的平衡, 因此需要额外的调整操作来维持平衡, 即维持不变式成立. 而不同的动态顺序统计树结构所对应的调整操作是不同的, 且调整操作需要考虑到不同的不平衡情况, 要处理的情况繁多且复杂, 直接用自动化证明组件得到自动验证是不实际的.

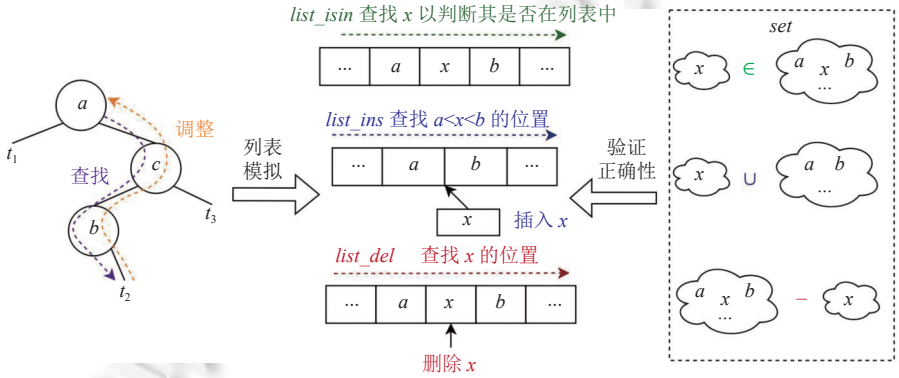


图 8 *list_isin*、*list_ins* 和 *list_del* 的模拟操作及验证

对于有序列表, *list_isin*、*list_ins* 和 *list_del* 函数需要递归查找元素 *x* 的位置, 然后执行相应的查找、插入或删除操作即可. 而由于列表本身是有序的, 故在插入和删除操作之后, 通过简单地合并列表, 能够保持有序性. 用这 3 个函数模拟动态顺序统计树类结构的查找、插入和删除操作, 可以将其在复杂的调整操作中抽象出来, 专注于查找、插入和删除的功能实现, 将问题简化.

list_isin、*list_ins* 和 *list_del* 函数的正确性证明可以借助集合中的属于 (∈)、并集 (∪) 和差集 (–) 来辅助验证, 同样可以仅使用归纳法并调用一次 *auto* 方法证明.

3.3 自动化验证框架的通用验证引理库

通过列表辅助函数模拟动态顺序统计树类结构的 10 种操作, 可以简化问题. 要进一步实现自动化验证, 需借助列表辅助函数构建通用的验证引理集, 合并 10 种操作对应的验证引理集, 组成验证引理库. 图 9 展示了基于引理库的自动化验证策略. 对于待证明的定理 *T*, 一般采用如下方式进行自动化验证.

待证明的定理 *T*	***theorem*** *T*: *invar t* ⟹ *tree_fun x t* = *list_fun x* (*inorder t*)		
归纳法	*apply* (***induct*** *t arbitrary*: *x*)	*apply* (***induction*** *x t rule*: *tree_fun.induct*)	
auto 方法	*by* / *apply* (***auto*** *simp*: *list_fun_simps fun_defs proven_lemmas split*: *type.splits*)		
try 命令	*by* / *apply* (***metis*** *other_lemmas*)	*by* / *apply* (***meson*** *other_lemmas*)	…

图 9 基于引理库的自动化验证策略

(1) 归纳法: *induct* 和 *induction* 均表示使用归纳法, 前者对单一对象进行归纳, 后者可处理多个对象. *arbitrary* 表示对除归纳变元 t 以外的自由变元使用全称量化[7], *rule* 明确指定归纳法则. 通过归纳法, 可以将目标定理拆解为平凡情况和非平凡情况. 平凡情况对应树为空的情况. 非平凡情况则以树为 t_1、t_2 时命题成立作为前提, 验证树为 $<n,t_1,a,t_2>$ 时命题成立, 即对应归纳法"从 n 到 n+1"的思想.

(2) *auto* 方法: Isabelle 中提供的一种自动证明方法. *simp* 表示应用简化规则, 在这里需添加验证引理库中的辅助验证引理集, 以简化子目标甚至直接得证. 除此之外, 在子目标中将使用到的函数定义以及辅助函数引理也需添加其中. 例如, 求前继元素的函数 *pred* 中调用了 *max* 函数, 在证明 *pred* 函数的正确性时, 需要将 *max* 函数的相关引理添加其中. 在子目标含有 if、case 等表达式时, *split* 可以有效地处理这些分支结构, 使 *auto* 方法可以分别处理这些分支情况, 从而证明目标.

(3) *try* 命令: *try* 命令可调用 Isabelle 中一些用于自动化证明的组件, 如 Sledgehammer 等, 这些组件会给出自动化证明的结果. 经过上述两步简化操作之后, 一般的定理可以直接得证. 而对于较为复杂的定理, 我们的验证引理库也可将其简化成足够简单的形式, 借助 *try* 命令调用自动证明机器能够给出证明的结果.

因此, 验证引理库的构造需覆盖子目标拆解后对应的所有情况, 包括边界情况 (即查找失败) 和找到目标节点的情况. 结合函数 *inorder*, 借助第 3.2 节中构建的列表辅助函数可以初步简化目标命题. 为了进一步简化目标, 在寻找目标节点的过程中, 可以将列表辅助函数中使用到的列表拆成 3 份, 对应树形结构的左子树、根节点和右子树. 在使用归纳法处理后, 非平凡情况下对应的子目标可能会导致节点数量的增加. 对于情况更为复杂的命题, 为了进一步提升自动化验证效率, 可以额外添加对这些情况处理的引理.

为了实现动态顺序统计类结构 10 种操作对应定理的自动化验证, 需要根据各操作的逻辑特点设计相应的验证引理集, 并将这些验证引理集整合成一个统一的验证引理库. 图 10 给出了 10 种操作的逻辑规约, 这些逻辑规约实际对应需证明的定理, 包括功能正确性定理和结构正确性定理. 功能正确性是指各个函数操作是否能正确执行预想的功能, 结构正确性是指执行修改类函数操作前后不变式 *invar* 成立. 对于动态顺序统计树类结构 10 种操作的功能正确性, 借助第 3.3.1 节给出的 10 种操作对应定理的验证引理集, 可以实现自动化验证. 而对于 *insert* 和 *delete* 等修改类函数, 在操作过程中可能会改变树中的节点信息, 破坏树的平衡性. 对于此类函数操作, 除了验证其功能正确性以外, 还需保证操作前后不变式 *invar* 成立, 即验证结构正确性. 不同的动态顺序统计树结构有不同的调整策略维持整体平衡性, 这些调整策略没有统一的构建思路, 差异较大. 因此, 对于结构正确性的验证, 仍需要构建针对性的引理辅助验证, 基于本文的自动化验证框架不能直接得到自动证明. 不过, 借助第 3.3.2 节中提出的拆分方案, 拆分不变式 *invar* 得到细化引理, 以降低目标命题的验证复杂度. 对于拆分后得到的细化引理, 自动化验证框架可以适用, 提升整体的自动化验证水平.

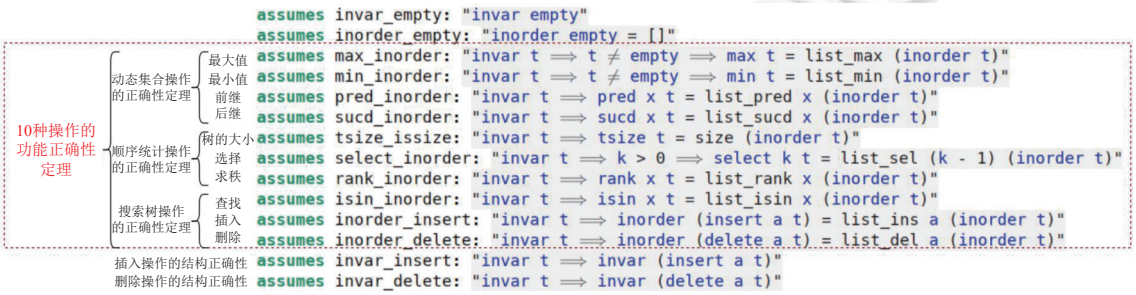


图 10 动态顺序统计类结构 10 种操作的逻辑规约

3.3.1 功能正确性验证引理集

定理 1. *max* 函数. *invar t* ⇒ *t* ≠ *empty* ⇒ *max t* = *list_max* (*inorder t*).

定理 2. *min* 函数. *invar t* ⇒ *t* ≠ *empty* ⇒ *min t* = *list_min* (*inorder t*).

在动态顺序统计树类结构中, 根据 inv_{ST} 可以推知, 树中最右边、最左边的节点键值分别为树的最大键值、最

小键值. 因此, *max* 和 *min* 函数可以分别通过递归遍历右、左子树节点实现. *list_max* 和 *list_min* 可以很好地模拟这一情况, 只需遍历获得列表最末端、首端的键值即可. 由此, 可分别构建 *list_max* 和 *list_min* 的验证引理集, 如图 11 所示. 引理 1 表示 *list_max* 可以正确获取列表最末端的键值元素. 引理 2 和引理 3 分别表示 *list_max*、*list_min* 能够模拟递归向右、左子树查找的过程.

引理 1 $list\_max\ (lst\ @\ [a]) = a$

引理 2 $lst_2 \neq [] \Longrightarrow list\_max\ (lst_1\ @\ a\ \#\ lst_2) = list\_max\ lst_2$

引理 3 $lst_1 \neq [] \Longrightarrow list\_min\ (lst_1\ @\ a\ \#\ lst_2) = list\_min\ lst_1$

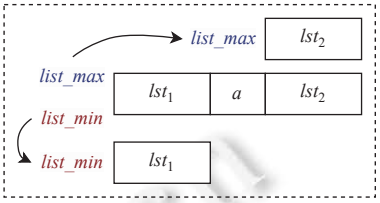


图 11 *list_max* 和 *list_min* 的验证引理集

定理 3. *pred* 函数. *invar t* ⇒ *pred x t* = *list_pred x* (*inorder t*).

pred 函数获取前继元素对应两种情况: 一是左子树不为空时, 获取目标节点左子树中的最大元素; 二是左子树为空时, 自底向上找到第 1 个右子树包含目标节点的祖先节点, 祖先节点对应的元素即为前继元素. 前者依赖于 *max* 函数, 可以用函数 *list_max* 模拟. 后者的实现需要正确记录祖先节点信息. 在自顶向下的查找过程中, 如果依据查找策略判断得出沿着右边的路径进行查找, 则需记录当前的根节点信息 (对应记录目标节点的祖先节点信息). 当找到目标节点时, 若其左子树为空, 则最近记录的根节点即为目标节点的前继节点.

基于上述分析, 可以构建函数 *list_pred* 的验证引理集, 如图 12 所示. 引理 4–6 处理边界情况, 即没有前继元素的情况, 引理 4 和引理 5 表示若元素 *x* 不在列表中, 则无法求得其前继元素; 引理 5 表示列表中最小的元素 (即有序列表的第 1 个元素) 没有前继元素. 引理 7–9 分别模拟目标节点等于、小于和大于当前根节点的情况. 若等于, 需要调用函数 *link_max* 来获取对应左子树中的最大元素, 该元素即为前继元素. 若小于或大于, 则需模拟树中向左或向右递归查找的过程. 对应引理 7–9, 引理 10–12 则分别处理使用归纳法后节点增加的情况.

引理 4 $sorted\ (a\ \#\ lst) \Longrightarrow x < a \Longrightarrow list\_pred\ x\ (a\ \#\ lst) = None$

引理 5 $sorted\ (a\ \#\ lst) \Longrightarrow list\_pred\ a\ (a\ \#\ lst) = None$

引理 6 $sorted\ (lst\ @\ [a]) \Longrightarrow x > a \Longrightarrow list\_pred\ x\ (lst\ @\ [a]) = None$

引理 7 $sorted\ (lst_1\ @\ a\ \#\ lst_2) \Longrightarrow lst_1 \neq [] \Longrightarrow list\_pred\ a\ (lst_1\ @\ a\ \#\ lst_2) = Some\ (list\_max\ lst_1)$

引理 8 $sorted\ (lst_1\ @\ a\ \#\ lst_2) \Longrightarrow x < a \Longrightarrow list\_pred\ x\ (lst_1\ @\ a\ \#\ lst_2) = list\_pred\ x\ lst_1$

引理 9 $sorted\ (lst_1\ @\ a\ \#\ lst_2) \Longrightarrow x > a \Longrightarrow list\_pred\ x\ (lst_1\ @\ a\ \#\ lst_2) = list\_pred\ x\ (a\ \#\ lst_2)$

引理 10 $sorted\ (a\ \#\ lst_1\ @\ b\ \#\ lst_2) \Longrightarrow lst_1 \neq [] \Longrightarrow list\_pred\ b\ (a\ \#\ lst_1\ @\ b\ \#\ lst_2) = Some\ (list\_max\ lst_1)$

引理 11 $sorted\ (a\ \#\ lst_1\ @\ b\ \#\ lst_2) \Longrightarrow x < b \Longrightarrow list\_pred\ x\ (a\ \#\ lst_1\ @\ b\ \#\ lst_2) = list\_pred\ x\ (a\ \#\ lst_1)$

引理 12 $sorted\ (a\ \#\ lst_1\ @\ b\ \#\ lst_2) \Longrightarrow x > b \Longrightarrow list\_pred\ x\ (a\ \#\ lst_1\ @\ b\ \#\ lst_2) = list\_pred\ x\ (b\ \#\ lst_2)$

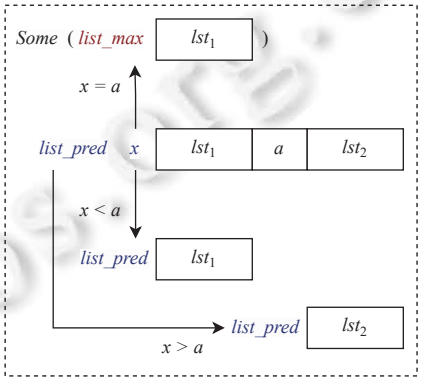


图 12 *list_pred* 的验证引理集

定理 4. *sucd* 函数. *invar t* ⇒ *sucd x t* = *list_sucd x* (*inorder t*).

求后继元素的函数 *sucd* 的实现思路与 *pred* 类似, 两者互为镜像操作. 函数 *sucd* 需借助 *min* 函数并记录左边路径的根节点信息. 由此, 可以构建函数 *list_sucd* 的验证引理集, 如图 13 所示. 引理 13–15 处理没有后继元素的情

况, 引理 13 和 14 表示元素不在列表中, 引理 15 则表示列表中最大的元素 (即有序列表的最后一个元素) 没有后继元素. 引理 16–18 分别对应根节点 (等于)、左子树 (小于)、和右子树 (大于) 的情况. 对于二叉型的结构, 这种方式已经涵盖了所有的路径执行情况. 而对于多叉型结构, 同样具有搜索树性质. 依据建模策略, 同样可以将其分为小于、等于和大于的情况, 再根据具体的结构, 在上述 3 种情况的基础上再细分小于、等于和大于的情况. 将树结构映射到列表结构中后, 二叉型和多叉型结构对应的情况是一样的, 所以该方式同样有效. 引理 19 对应处理归纳法中非平凡情况节点增加的情况.

引理 13 $sorted\ (lst\ @\ [a]) \Longrightarrow x > a \Longrightarrow list\_sucd\ x\ (lst\ @\ [a]) = None$

引理 14 $sorted\ (a\ \#\ lst) \Longrightarrow x < a \Longrightarrow list\_sucd\ x\ (a\ \#\ lst) = None$

引理 15 $sorted\ (lst\ @\ [a]) \Longrightarrow list\_sucd\ a\ (lst\ @\ [a]) = None$

引理 16 $sorted\ (lst_1\ @\ a\ \#\ lst_2) \Longrightarrow lst_2 \neq [] \Longrightarrow list\_sucd\ a\ (lst_1\ @\ a\ \#\ lst_2) = Some\ (list\_min\ lst_2)$

引理 17 $sorted\ (lst_1\ @\ a\ \#\ lst_2) \Longrightarrow x < a \Longrightarrow list\_sucd\ x\ (lst_1\ @\ a\ \#\ lst_2) = list\_sucd\ x\ (lst_1\ @\ [a])$

引理 18 $sorted\ (lst_1\ @\ a\ \#\ lst_2) \Longrightarrow x > a \Longrightarrow list\_sucd\ x\ (lst_1\ @\ a\ \#\ lst_2) = list\_sucd\ x\ lst_2$

引理 19 $sorted\ (lst_1\ @\ a\ \#\ b\ \#\ lst_2) \Longrightarrow list\_sucd\ a\ (lst_1\ @\ a\ \#\ b\ \#\ lst_2) = Some\ b$

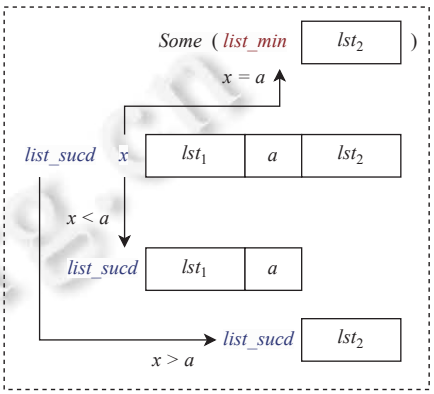


图 13 *list_sucd* 的验证引理集

定理 5. *tsize* 函数. $invar\ t \Rightarrow tsize\ t = size\ (inorder\ t)$.

定理 5 表示 *tsize* 函数获取的树的节点大小值是正确的. 对于该定理的证明, 自动化证明组件足以应付, 无需构造额外的验证引理集. 图 14 展示了定理 5 的自动化验证 (*tsize* 实例化为 *dost_size*), 只需要通过归纳法并调用一次 *auto* 方法即得证.

```
lemma dost_size_issize: "inv_size t ⟹ dost_size t = size t"
  by (induct t) (auto simp add: size2_def)
```

图 14 *dost_size* 的正确性验证

定理 6. *select* 函数. $invar\ t \Rightarrow k > 0 \Rightarrow select\ k\ t = list\_sel\ (k-1)\ (inorder\ t)$.

定理 7. *rank* 函数. $invar\ t \Rightarrow rank\ x\ t = list\_rank\ x\ (inorder\ t)$.

与函数 *list_pred* 函数 *list_sucd* 验证引理集的构建思路类似, 函数 *list_sel* 和 *list_rank* 均对不在列表中、等于、小于和大于的情况进行了模拟操作并验证, 如图 15 所示. 函数 *select* 的查找过程依赖于树的大小信息, 因此, 函数 *list_sel* 的模拟过程需根据列表的大小信息来判断查找方向, 如引理 20–24 所示. 函数 *rank* 的查找过程为常规的递归查找, 因此 *list_rank* 的模拟过程则根据键值大小判断方向即可, 如引理 25–29 所示.

引理 20 $k > size\ lst \Longrightarrow list\_sel\ k\ lst = None$

引理 21 $k = size\ lst_1 \Longrightarrow list\_sel\ k\ (lst_1\ @\ a\ \#\ lst_2) = Some\ a$

引理 22 $k < size\ lst_1 \Longrightarrow list\_sel\ k\ (lst_1\ @\ a\ \#\ lst_2) = list\_sel\ k\ lst_1$

引理 23 $k > size\ lst_1 \Longrightarrow list\_sel\ k\ (lst_1\ @\ a\ \#\ lst_2) = list\_sel\ (k-size\ lst_1-1)\ lst_2$

引理 24 $k > size\ lst_1+1 \Longrightarrow list\_sel\ k\ (lst_1\ @\ a\ \#\ lst_2) = list\_sel\ (k-size\ lst_1-1-1)\ lst_2$

引理 25 $sorted\ (a\ \#\ lst) \Longrightarrow x < a \Longrightarrow list\_rank\ x\ lst = None$

引理 26 $sorted\ (lst\ @\ [a]) \Longrightarrow x > a \Longrightarrow list\_rank\ x\ (lst\ @\ [a]) = None$

引理 27 $sorted\ (lst_1\ @\ a\ \#\ lst_2) \Longrightarrow k = size\ lst_1 + 1 \Longrightarrow list\_rank\ a\ (lst_1\ @\ a\ \#\ lst_2) = Some\ k$

引理 28 $sorted\ (lst_1\ @\ a\ \#\ lst_2) \Longrightarrow x < a \Longrightarrow list\_rank\ x\ (lst_1\ @\ a\ \#\ lst_2) = list\_rank\ x\ lst_1$

引理 29 $sorted\ (lst_1\ @\ a\ \#\ lst_2) \Longrightarrow x > a \Longrightarrow k = size\ lst_1 + 1 \Longrightarrow list\_rank\ x\ (lst_1\ @\ a\ \#\ lst_2) = plus\_option\ k\ (list\_rank\ x\ lst_2)$

图 15 *list_sel* 和 *list_rank* 的验证引理集

定理 8. *isin* 函数. $invar\ t \Rightarrow isin\ x\ t = list\_isin\ x\ (inorder\ t)$.

定理 9. *insert* 函数. $invar\ t \Rightarrow inorder\ (insert\ a\ t) = list\_ins\ a\ (inorder\ t)$.

定理 10. *delete* 函数. $invar\ t \Rightarrow inorder\ (delete\ a\ t) = list\_del\ a\ (inorder\ t)$.

定理 8−10 给出了搜索树结构常规的查找、插入和删除操作对应的正确性验证操作. 具体的验证引理集的构建思路与上文中的其他定理类似. 这些函数的引理集如图 16 所示.

引理 30 $list\_isin\ x\ lst = (x \in set\ lst)$

引理 31 $sorted\ (x\ \#\ xs) = ((\forall y \in set\ xs.\ x < y) \wedge sorted\ xs)$

引理 32 $sorted\ (xs\ @\ [x]) = (sorted\ xs\ \wedge (\forall y \in set\ xs.\ y < x))$

引理 33 $[\![sorted\ (lst_1\ @\ [a]); x < a\ ]\!] \Rightarrow list\_ins\ x\ (lst_1\ @\ a\ \#\ lst_2) = (list\_ins\ x\ lst_1)\ @\ a\ \#\ lst_2$

引理 34 $[\![sorted\ (lst_1\ @\ [a]); a \leqslant x]\!] \Rightarrow list\_ins\ x\ (lst_1\ @\ a\ \#\ lst_2) = lst_1\ @\ list\_ins\ x\ (a\ \#\ lst_2)$

引理 35 $[\![sorted\ (lst_1\ @\ a\ \#\ lst_2); a \leqslant x]\!] \Rightarrow list\_del\ x\ (lst_1\ @\ a\ \#\ lst_2) = lst_1\ @\ list\_del\ x\ (a\ \#\ lst_2)$

引理 36 $[\![sorted\ (lst_1\ @\ a\ \#\ lst_2); x < a\ ]\!] \Rightarrow list\_del\ x\ (lst_1\ @\ a\ \#\ lst_2) = list\_del\ x\ lst_1\ @\ a\ \#\ lst_2$

引理 37 $[\![sorted\ (lst_1\ @\ a\ \#\ lst_2\ @\ b\ \#\ lst_3); x < b\ ]\!] \Rightarrow list\_del\ x\ (lst_1\ @\ a\ \#\ lst_2\ @\ b\ \#\ lst_3) = list\_del\ x\ (lst_1\ @\ a\ \#\ lst_2)\ @\ b\ \#\ lst_3$

引理 38 $[\![sorted\ (lst_1\ @\ a\ \#\ lst_2\ @\ b\ \#\ lst_3\ @\ c\ \#\ lst_4); x < c\ ]\!] \Rightarrow list\_del\ x\ (lst_1\ @\ a\ \#\ lst_2\ @\ b\ \#\ lst_3\ @\ c\ \#\ lst_4) = list\_del\ x\ (lst_1\ @\ a\ \#\ lst_2\ @\ b\ \#\ lst_3)\ @\ c\ \#\ lst_4$

引理 39 $[\![sorted\ (lst_1\ @\ a\ \#\ lst_2\ @\ b\ \#\ lst_3\ @\ c\ \#\ lst_4\ @\ d\ \#\ lst_5); x < d\ ]\!] \Rightarrow list\_del\ x\ (lst_1\ @\ a\ \#\ lst_2\ @\ b\ \#\ lst_3\ @\ c\ \#\ lst_4\ @\ d\ \#\ lst_5) = list\_del\ x\ (lst_1\ @\ a\ \#\ lst_2\ @\ b\ \#\ lst_3\ @\ c\ \#\ lst_4\ )\ @\ d\ \#\ lst_5$

图 16 *list_isin*、*list_ins* 和 *list_del* 的验证引理集

函数 *isin* 实现判断元素是否在树中的操作, 这一实现与集合中的属于 (∈) 的功能一致. 因此, 可以进一步将列表辅助函数 *list_isin* 转化为集合操作, 用集合来辅助验证, 以简化命题, 如引理 30–32 所示. 函数 *insert* 和 *delete* 的实现按常规思路即可, 函数 *list_ins* 和 *list_del* 只需考虑左、右插入和删除节点的情况, 如引理 33–39 所示.

定理 1−10 描述的是动态顺序统计树类结构 10 种操作的功能正确性定理. 基于自动化验证框架, 10 种操作的功能正确性均能得到自动验证.

3.3.2 结构正确性验证拆分方案

除了验证动态顺序统计树类结构 10 种操作的功能正确性以外, 还需保证修改类函数 (*insert* 和 *delete*) 操作前后不变式 *invar* 成立, 即验证结构正确性.

对于修改类函数, 在操作过程中可能会改变树中的节点信息, 还需要引入额外的调整策略来维持整体平衡性. 因此, 对于此类函数的结构正确性验证, 可通过定理 11 和定理 12 来保证.

定理 11. *insert* 函数. $invar\ t \Rightarrow invar\ (insert\ a\ t)$.

定理 12. *delete* 函数. $invar\ t \Rightarrow invar\ (delete\ a\ t)$.

不同的动态顺序统计树结构所具有的不变式 *invar* 都不相同, 维持这些 *invar* 的调整策略差异较大, 没有统一的设计思路. 所以, 对于结构正确性的验证, 仍需要构建针对性的引理辅助验证. 不过, 这些针对性引理的设计也可遵循固定的模式. 以函数 *insert* 的结构正确性验证为例, 可采取拆分方案, 如下所示.

$$
invar\ t \Rightarrow invar\ (insert\ a\ t)\begin{cases} invar\ t \Rightarrow invar\ (bal\_fun_1\ t)\begin{cases} inv_{ST}\ t \Rightarrow inv_{ST}\ (bal\_fun_1\ t) \\ inv_{size}\ t \Rightarrow inv_{size}\ (bal\_fun_1\ t) \\ inv_{prop1}\ t \Rightarrow inv_{prop1}\ (bal\_fun_1\ t) \\ \cdots \\ inv_{propn}\ t \Rightarrow inv_{propn}\ (bal\_fun_1\ t) \end{cases} \\ \cdots \\ invar\ t \Rightarrow invar\ (bal\_fun_n\ t)\begin{cases} inv_{ST} t \Rightarrow inv_{ST}\ (bal\_fun_n\ t) \\ inv_{size}\ t \Rightarrow inv_{size}\ (bal\_fun_n\ t) \\ inv_{prop1}\ t \Rightarrow inv_{prop1}\ (bal\_fun_n\ t) \\ \cdots \\ inv_{propn}\ t \Rightarrow inv_{propn}\ (bal\_fun_n\ t) \end{cases} \end{cases}.
$$

在验证其结构正确性 (定理 11) 之前, 可先验证其维持平衡的调整函数 (bal_fun_1 等) 的结构正确性. 进一步, 可将不变式 *invar* 拆分为 inv_{ST}、inv_{size} 和 inv_{prop} (定义见第 2.3 节) 分别进行验证, 以降低整体的验证复杂度. 对于拆分后得到的细化引理, 本文的自动化验证框架可以适用, 提升整体的自动化验证水平.

对于函数 *delete* 的结构正确性验证 (定理 12), 可同样按照上述拆分方案对定理进行拆分, 以达到简化目标定理的目的, 进一步借助自动化验证框架自动验证拆分后的细化引理, 提高自动化水平.

4 建模和自动化验证实例

为了评估动态顺序统计树类结构函数式建模和 Isabelle 自动化验证框架的有效性, 本文选取了不平衡的二叉搜索树、平衡的二叉搜索树 (红黑树) 和平衡的多叉搜索树 (2−3 树) 为实例, 从不同维度检验框架的适用性.

4.1 实例化为不平衡的二叉搜索树

不平衡的二叉搜索树即为最基础的二叉搜索树, 只有搜索树性质而没有平衡操作, 不带有附加性质. 基于动态顺序统计树类结构的函数式定义 (第 2.1 节) 和不变式 (第 2.3 节), 将 $dost_{syn}$ 实例化为 $dost_{bst}$, 不变式 *invar* 实例化为 $invar_{bst}$, 不平衡的动态顺序统计二叉搜索树的结构和不变式定义可表示如下.

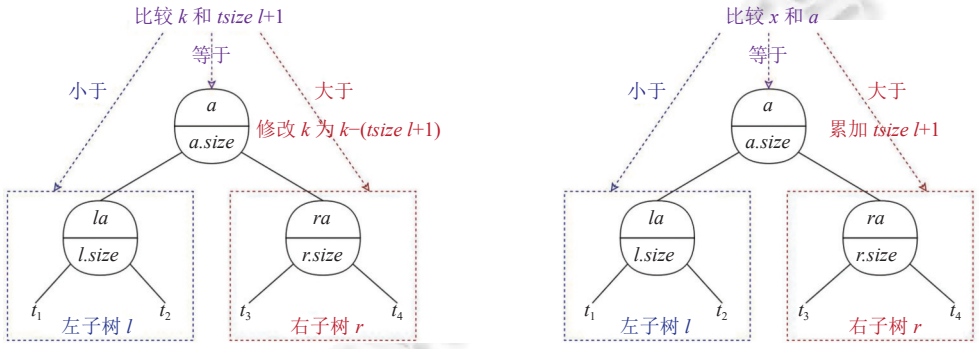
type_synonym $'a\ dost_{bst}$ = "$'a\ dost$"//不平衡的二叉搜索树的结构定义
definition "$invar_{bst}\ t \equiv inv_{ST}\ t \wedge inv_{size}\ t$"//不平衡的二叉搜索树的不变式定义

不平衡的二叉搜索树的正确性定理包括功能正确性定理和结构正确性定理, 功能正确性定理表示各个函数操作能够正确实现预想的功能, 结构正确性定理表示执行修改类函数操作前后不变式 $invar_{bst}$ 成立.

4.1.1 $dost_{bst}$ 的功能正确性验证

(1) 函数式建模

基于第 2.2 节中的函数式建模策略, 可以设计不平衡的二叉搜索树 $dost_{bst}$ 的具体函数. 以选择和求秩函数为例, 可分别实例化为 $dost_{bst}\_select$ 和 $dost_{bst}\_rank$. 选择函数 $dost_{bst}\_select$ 用于查找树 $dost_{bst}$ 中第 k 小的元素, 求秩函数 $dost_{bst}\_rank$ 用于计算给定元素 x 的排名, 即比它小的元素的数量. 动态顺序统计树类结构在每个节点处维护了树的大小值, 即以当前节点为根的子树中的节点数量, 该值可以通过函数 *tsize* (定义见第 2.1 节) 获取. 则选择函数 $dost_{bst}\_select$ 可以通过比较给定值 k 与 $tsize\ l+1$ 来确定查找路径, 并递归执行, 而求秩函数 $dost_{bst}\_rank$ 在查找过程中递归累加比 x 小的节点数量 (值为 $tsize\ l+1$), 两个函数的实现过程如图 17 所示.



(a) 选择函数 $dost_{bst}\_select$ (b) 求秩函数 $dost_{bst}\_rank$

图 17 选择函数 $dost_{bst}\_select$ 和求秩函数 $dost_{bst}\_rank$ 的实现过程

基于第 2.2 节中的高阶复合函数, 可对函数 $dost_{bst}\_select$ 和 $dost_{bst}\_rank$ 进行函数式建模, 如下所示. 其中, $let\ n = tsize\ l+1$ 为查找过程前的前置操作 (*preOperation*), $cmp\ k\ n$ 和 $cmp\ x\ a$ 分别对应两个函数的比较大小操作 (*compOperation*), 通过比较之后的结果 (*LT*、*GT* 和 *EQ*) 确定查找路径.

```
//实例化为不平衡的二叉搜索树dost_bst的选择函数
fun dost_bst_select :: "nat ⇒ 'a dost_bst ⇒ 'a option" where
  "dost_bst_select k <>= None"|
  "dost_bst_select k < _,l,a,r >= (let n = tsize l + 1 in case cmp k n of
       LT ⇒ dost_bst_select k l|
       GT ⇒ dost_bst_select (k − n) r|
       EQ ⇒ Some a)"
//实例化为不平衡的二叉搜索树dost_bst的求秩函数
fun dost_bst_rank :: "'a ⇒ 'a dost_bst ⇒ nat option" where
  "dost_bst_rank_ <>= None"|
  "dost_bst_rank x < _,l,a,r >= (let n = tsize l + 1 in case cmp x a of
       LT ⇒ dost_bst_rank x l|
       GT ⇒ plus_option n (dost_bst_rank x r)|
       EQ ⇒ Some n)"
```

(2) 自动化验证

基于第 3.3 节给出的基于引理库的自动化验证策略, 可以在 Isabelle 中自动验证树 $dost_{bst}$ 函数的功能正确性. 对于函数 $dost_{bst}\_select$ 和 $dost_{bst}\_rank$ 的功能正确性验证, 在采用归纳法拆解目标后, 使用 *auto* 方法调用验证引理库中对应的验证引理集, 并添加相应的函数定义和辅助函数的正确性引理, 可以化简目标命题进而实现自动证明, 如图 18 所示. 对于其他函数功能正确性的验证, 同样仅需使用归纳法并调用一次 *auto* 方法或使用 *try* 命令即可.

图 18 函数 $dost_{bst}\_select$ 和 $dost_{bst}\_rank$ 的功能正确性自动化验证过程

4.1.2 $dost_{bst}$ 的结构正确性验证

选择和求秩函数操作前后不会改变树的结构, 仅需对其功能正确性进行验证. 但是, 对于插入函数 *insert* 等修改类函数, 在操作过程中可能会改变树中的节点信息. 因此, 对于此类函数的验证, 除了验证其功能正确性以外, 还需保证其操作前后不变式 $invar_{bst}$ 成立, 即验证结构正确性.

(1) 函数式建模

以函数 *insert* 为例, 基于第 2.2 节中的高阶复合函数, 将 *insert* 实例化 $dost_{bst}\_insert$, 对其进行函数式建模, 如下所示. 其中, *szNode* 对应平衡调整操作 (*adjustLeaf*、*adjustLT*、*adjustGT* 和 *adjustEQ*).

```
//实例化为不平衡的二叉搜索树dost_bst的插入函数
fun dost_bst_insert :: "'a ⇒ 'a dost_bst ⇒ 'a dost_bst" where
  "dost_bst_insert x <>= szNode Leaf x Leaf"|
  "dost_bst_insert x < _,l,a,r >= (case cmp x a of
        LT ⇒ szNode (dost_bst_insert x l) a r|
        GT ⇒ szNode l a (dost_bst_insert x r)|
        EQ ⇒ szNode l a r)"
```

(2) 自动化验证

对于其结构正确性的验证, 则需借助第 3.3.2 节的拆分方案. 树 $dost_{bst}$ 的不变式 $invar_{bst}$ 满足 inv_{ST} 和 inv_{size}, 不具有附加性质 inv_{prop}. 因此, 先分别基于 inv_{ST} 和 inv_{size} 进行验证, 再验证函数 $dost_{bst}\_insert$ 的结构正确性. 对于拆分后得到的细化引理, 本文的自动化验证框架可以适用, 提升整体的自动化验证水平, 如图 19 所示.

图 19 函数 $dost_{bst}\_insert$ 的结构正确性验证过程

4.2 实例化为平衡的二叉搜索树

红黑树是最为常用的平衡二叉搜索树, 广泛应用于操作系统内核、数据库索引和任务调度等领域. 例如, 在 Linux 内核中, 可使用动态顺序统计红黑树来管理进程调度、内存区域和虚拟内存页表等数据结构, 优化数据的存储和检索效率. 红黑树最早由 Bayer[21]提出, Guibas 等人[22]引入了红/黑的颜色约定. 红黑树具有附加的颜色性质, 在调整操作中, 需要对颜色性质进行调整. 基于第 2.1 节中动态顺序统计树类结构的函数式定义, 将 $dost_{syn}$ 实例化为 $dost_{rbt}$, 并将 *extra_prop* 实例化为附加的红/黑颜色信息 *tcolor*, 其结构可定义为:

$$\text{type\_synonym}\ 'a\ dost_{rbt} = "('a \times tcolor)\ dost".$$

红黑树具有以下性质:

(1) 每个节点是红色或黑色, 故引入附加信息 *tcolor* 来表示;

(2) 根节点是黑色, 即 $dost_{rbt}\_color\ t = Black$;

(3) 所有的叶子节点 (<>) 都是黑色, 即 $dost_{rbt}\_color\ Leaf = Black$, 这是默认的;

(4) 每个红色节点的两个子节点都是黑色, 即从每个叶子节点到根节点的所有路径中, 不能出现两个连续的红色节点, 用 inv_{color} 来表示这一性质;

(5) 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点, 用 inv_{height} 来表示这一性质.

颜色性质不变式 inv_{rb} 表示红黑树需满足上述性质. 基于第 2.3 节的不变式, 将 *invar* 实例化为 $invar_{rbt}$, 将 inv_{prop} 实例化为 inv_{rb}, 可定义树 $dost_{rbt}$ 的不变式, 如下所示.

```
//红黑树dost_rbt的颜色性质不变式
definition "inv_rb t ≡ inv_color t ∧ inv_height t ∧ dost_rbt_color t = Black"
//红黑树dost_rbt的不变式
definition "invar_rbt t ≡ inv_ST t ∧ inv_size t ∧ inv_rb t"
```

以选择和求秩函数为例, 基于第 2.2 节中的高阶复合函数可对其进行函数式建模, 实现思路与第 4.1.1 节类似, 限于篇幅略.

对于红黑树中函数功能正确性的验证, 基于第 3.3 节给出的基于引理库的自动化验证策略, 同样能够得到自动化验证. 第 3.3.1 节中给出的验证引理库均是通用的, 因此同样使用验证引理库中的 *list_sel* 和 *list_rank* 验证引理集, 且只需要使用归纳法并调用一次 *auto* 方法即可完成选择和求秩函数的自动化验证, 如图 20 所示.

图 20 函数 $dost_{rbt}\_select$ 和 $dost_{rbt}\_rank$ 的功能正确性自动化验证过程

对于红黑树中函数结构正确性的验证, 实现思路与第 4.1.2 节类似. 借助第 3.3.2 节的拆分方案, 将不变式 $invar_{rbt}$ 拆分为 inv_{ST}、inv_{size} 和 inv_{rb} 等, 得到细化引理. 对于细化引理, 本文的自动化验证框架同样可以自动化验证.

4.3 实例化为平衡的多叉搜索树

2–3 树是一种特殊的 B-树[23]结构, 常用于文件系统和数据库系统中, 动态顺序统计 2–3 树可用于实现目录结构管理, 快速检索相应数据. 2–3 树的每个非叶节点都有 2 个或 3 个子树, 而且所有叶子节点都在同一层上. 本节以 2–3 树作为平衡的多叉搜索树的典型案例进行讨论. 基于第 2.1 节中动态顺序统计树类结构的函数式定义, 可将其结构定义如下.

```
datatype 'a dost_23 = Leaf |
Node2 nat "'a dost_23" 'a "'a dost_23" |
Node3 nat "'a dost_23" 'a "'a dost_23" 'a "'a dost_23"
```

对于 2–3 树这类包含多个子树的多叉搜索树, 其操作函数的实现思路与二叉搜索树是一致的, 只需要额外补充对多子树节点 (*Node*3) 的处理. 将函数 *tsize* 更新为函数 $dost_{23}\_size$, 只需要额外添加对 *Node*3 节点的处理操作即可. 如下展示了不变式 inv_{size} 额外添加的处理操作, 其中函数 *size*3 用于计算 *Node*3 节点对应树的大小值.

//函数$dost23_size$中额外补充的 $Node3$ 节点情况

$dost_{23}\_size < n,\_,\_,\_,\_,\_ >= n$

//计算$Node3$节点对应树的大小值

definition "$size3\ l\ m\ r \equiv dost_{23}\_size\ l + dost_{23}\_size\ m + dost_{23}\_size\ r + 2$"

//函数inv_{size}中额外补充的$Node3$节点情况

$inv_{size} < n,l,\_,m,\_,r >= (inv_{size}\ l \wedge inv_{size}\ m \wedge inv_{size}\ r \wedge (size3\ l\ m\ r = n))$

2−3 树的所有叶子节点都在同一层, 即其同一层非叶子节点的高度是一致的. 函数 $dost_{23}\_height$ 可以计算 2−3 树的高度. 由此, 可定义 2−3 树的高度不变式 inv_{height} 确保 2−3 树高度一致性. 基于第 2.3 节的不变式, 可定义 2−3 树的不变式 $invar_{23}$, 其不仅要满足 inv_{ST} 和 inv_{size}, 还需满足高度不变式 inv_{height}. 具体定义如下.

//2−3树$dost_{23}$的高度 不变式对应的 $Node3$ 节点情况

fun inv_{height} :: "$'a\ dost_{23} \Rightarrow bool$" where

"$inv_{height}\ Node3\_l\_m\_r = (inv_{height}\ l\ \ \&\ \ inv_{height}\ m\ \ \&\ \ inv_{height}\ r\ \ \&$

$dost_{23}\_height\ l = dost_{23}\_height\ m\ \ \&\ \ dost_{23}\_height\ m = dost_{23}\_height\ r)$"

//2−3树$dost_{23}$的不变式

definition "$invar_{23}\ t \equiv inv_{ST}\ t\ \wedge\ inv_{size}\ t \wedge\ inv_{height}\ t$"

基于第 2 节的函数式建模框架和第 3 节的自动化验证框架, 可对 2−3 树的函数进行函数式建模并自动化验证, 具体实现思路与第 4.1 节和第 4.2 节类似, 此处略.

本节旨在通过区域 (*locale*) 全面展示 2−3 树动态集合操作 (如前继和后继等), 顺序统计操作 (如选择和求秩等) 和搜索树操作 (查找、插入和删除) 的正确性验证结果. 区域是一种程序模块化和参数化的复用机制, 能有效表达函数式程序结构之间复杂的依赖关系[17]. 通过区域声明, 前文定义了动态顺序统计树类结构通用的局部参数 (第 3.2 节图 4) 和逻辑规约 (第 3.3 节图 10). 其中, 局部参数对应要实现的 10 种函数操作, 逻辑规约对应待证明的正确性定理 (包括功能正确性和结构正确性定理). 通过区域解释将局部参数实例化为 2−3 树的相关函数操作, 然后对其逻辑规约进行验证. 在验证了 10 种函数操作的正确性定理之后, 逻辑规约的验证可以通过 *try* 命令直接得出. 图 21 展示了 2−3 树 10 种函数操作的正确性验证结果, 进一步检验了本文第 2、3 节所提函数式建模和自动化验证框架的有效性.

局部参数的实例化

```
interpretation dynamic_order_statistic_tree
where empty = dost23_empty and invar = invar23 and inorder = dost23_inorder
    and max = dost23_max and min = dost23_min and pred = dost23_pred and sucd = dost23_sucd
    and tsize = dost23_size and select = dost23_select and rank = dost23_rank
    and isin = dost23_isin and insert = dost23_insert and delete = dost23_delete
```

逻辑规约的验证

```
proof (standard, goal_cases)
     case 1 thus ?case by (simp add: invar23_def dost23_empty_def)
next case 2 thus ?case by (simp add: dost23_empty_def)
next case 3 thus ?case by (simp add: invar23_def dost23_max_node)
next case 4 thus ?case by (simp add: invar23_def dost23_min_node)
next case 5 thus ?case by (simp add: invar23_def dost23_pred_inorder)
next case 6 thus ?case by (simp add: invar23_def dost_sucd_inorder)
next case 7 thus ?case by (simp add: invar23_def dost23_size_issize)
next case 8 thus ?case by (simp add: invar23_def dost23_select_inorder)
next case 9 thus ?case by (simp add: invar23_def dost23_rank_inorder)
next case 10 thus ?case by (simp add: invar23_def dost23_isin_inorder)
next case 11 thus ?case by (simp add: invar23_def dost23_inorder_insert)
next case 12 thus ?case by (simp add: invar23_def dost23_inorder_delete)
next case 13 thus ?case by (simp add: dost23_insert)
next case 14 thus ?case by (simp add: dost23_delete)
qed
```

try 命令直接得出

图 21 2−3 树 10 种函数操作的正确性验证结果

5 自动化验证框架的适用范围及效果

5.1 适用范围

本文提出了动态顺序统计树类结构相关算法的自动化验证框架, 该框架的设计思路为: 使用同态的列表模拟动态顺序统计树类结构的操作, 设计并验证列表辅助函数, 进一步借助列表辅助函数构建通用且可复用的验证引理库. 本文的自动化验证框架主要适用于以下情况.

(1) 适用于至少满足两个不变式 inv_{ST} 和 inv_{size} 的树型结构. 在第 4 节中, 以不平衡的二叉搜索树、红黑树和 2−3 树为典型案例进行了详细的讨论, 验证了本文自动化验证框架的适用性和有效性. 实际上, 动态顺序统计树类结构操作的函数实现以及功能正确性验证依赖于搜索树不变式 inv_{ST} 和树的大小不变式 inv_{size}. 对于至少满足两个不变式 inv_{ST} 和 inv_{size} 的树型结构, 如高度平衡树、AA 树、2−3−4 树等, 其函数操作的功能正确性均可借助本文的自动化验证框架得到自动验证.

(2) 适用于动态顺序统计树类结构的 10 种操作, 包括动态集合相关操作 (即求最大/小值、前/后继)、顺序统计操作 (即求树的大小、选择和求秩) 和搜索树操作 (即查找、插入和删除). 动态顺序统计树类结构能在 O(log$n$) 时间内实现顺序统计操作[1]. 对于动态顺序统计树类结构常用的 10 种操作功能正确性的自动验证, Nipkow 在文献 [5] 中提出的自动化验证框架仅适用于其中关于搜索树类结构的算法操作, 不适用于动态集合相关操作和顺序统计操作. 而本文提出的自动化验证框架对这 10 种操作的功能正确性均能实现自动验证. 对于这 10 种操作以外的特殊操作功能正确性的自动验证, 如实例化为区间树结构时如何维护区间信息等, 本文的自动化验证框架暂不适用. 不过, 本文中用列表模拟树操作来简化目标命题的方法是通用的, 且验证引理库具有可扩展性. 在验证这类特殊操作的功能正确性时, 可以按照相同的思路针对性地构建辅助验证引理集, 扩展验证引理库, 以此来简化目标命题. 因此, 对于这类特殊操作, 本文的自动化验证框架可提供参考.

(3) 适用于功能正确性的自动化验证. 本文的自动化验证框架可以用于动态顺序统计树类结构的 10 种操作的功能正确性验证, 实现自动证明. 而对于结构正确性的验证, 仍需构建专门的引理进行辅助验证, 自动化验证框架无法直接实现自动证明. 然而, 通过第 3.3.2 节中提出的拆分方案, 可以将不变式 *invar* 进行拆分, 得到细化引理, 从而降低验证目标命题的复杂度. 对于这些拆分后的细化引理, 自动化验证框架是适用的, 从而提高整体的自动化验证效果.

5.2 效果对比

对于搜索树类算法的验证, 现有工作主要通过交互式验证方法[9−16]来确保正确性. 其思路为先设计待证明的引理, 在初步化简命题之后, 分析化简后得到的子目标, 根据子目标的特性构造前置引理, 进一步化简子目标. 通过不断地构造前置引理, 逐步化简子目标, 最终得到证明. 这一过程需要较多的人工参与, 前置引理的构造也需要创造性思维, 对开发人员提出了较高的要求. 且这种方法不具有通用性, 在应用场景发生变化时, 需要重新构造前置引理, 费时费力.

此外, Nipkow 在文献 [5] 中提出了验证搜索树类结构功能正确性的自动化验证框架, 对不同搜索树结构的功能正确性进行了自动化证明. 然而, 该框架仅对搜索树类结构的查找、插入和删除操作进行了自动化验证, 不适用于动态集合和顺序统计量相关算法操作.

本文提出的自动化验证框架, 一方面, 不需要额外构造前置引理, 减少人工参与. 且本文的验证引理库是通用的, 在应用场景发生变化时, 依然可以证明目标命题. 另一方面, 本文的自动化验证框架除了适用现有工作已经得到验证的搜索树相关操作 (查找、插入和删除), 还涵盖了动态集合 (如前继和后继等) 和顺序统计相关操作 (如选择和求秩等). 交互式验证方法、文献 [5] 和本文的自动化验证框架这 3 种方法的对比如表 1 所示.

以函数 *select* 和函数 *rank* 的正确性验证为例, 图 22 给出了交互式验证方法和本文自动化验证框架的验证过程对比. 从图 22 中可以看出, 采用本文提出的自动化验证框架, 不需要额外构造前置引理, 且验证代码行数显著减少.

表 1 验证方法的对比

验证方法	是否为自动化验证	适用动态集合操作				适用顺序统计操作			适用搜索树操作		
		最大值	最小值	前继	后继	树的大小	选择	求秩	查找	插入	删除
交互式[9−16]	○	●	●	●	●	●	●	●	●	●	●
文献[5]	●	○	○	○	○	○	○	○	●	●	●
本文	●	●	●	●	●	●	●	●	●	●	●

(a) 交互式验证方法 (b) 本文自动化验证框架

图 22 交互式验证方法与本文自动化验证框架的验证过程对比

表 2 给出了两种方法的前置引理数和验证代码行数的对比. 具体而言, 对于函数 *select* 和函数 *rank* 的正确性验证, 交互式验证方法分别需要构造 10 条、15 条前置引理, 对应 58 行、81 行验证代码. 而使用本文的自动化验证框架, 无需构造前置引理, 验证代码行数均精简至 4 行.

表 2 前置引理数和验证代码行数的对比

验证方法		交互式验证方法	本文自动化验证框架
*select*函数的正确性验证	前置引理条数	10	0
	验证代码行数	58	4
*rank*函数的正确性验证	前置引理条数	15	0
	验证代码行数	81	4

当应用场景发生改变时, 如改为在红黑树 $dost_{\text{rbt}}$ 中实现, 交互式验证方法需重新构造前置引理, 且由于红黑树 $dost_{\text{rbt}}$ 结构的复杂程度远大于不平衡的二叉搜索树 $dost_{\text{bst}}$, 因此前置引理的构造将更为复杂且繁琐. 而本文所提的自动化验证框架具备良好的通用性, 依旧可以处理. 如图 23 所示, 对于红黑树 $dost_{\text{rbt}}$ 中 *select* 和 *rank* 函数的验证, 本文的自动化验证框架同样不需要构造前置引理, 得到自动验证, 代码行数也仅为 4 行.

(a) 不平衡的二叉搜索树 $dost_{\text{bst}}$ (b) 红黑树 $dost_{\text{rbt}}$

图 23 自动化验证框架的通用性

6 总结与展望

本文基于 Isabelle 定理证明器, 建立了动态顺序统计树类结构的函数式建模框架和自动验证框架. 在建模框架中, 设计了动态顺序统计树类结构的通用结构, 建立了统一建模策略, 并刻画了其 10 种操作的高阶复合函数. 在自动验证框架中, 构建了自动化验证的辅助引理库, 并给出了具体的验证方案, 以克服构造辅助引理时的盲目性. 基于函数式建模框架, 可以在不同的具体搜索树结构中进行相应的实例化, 以适用不同的应用场景. 文中考虑了平衡与不平衡、二叉与多叉两个维度, 选取了不平衡的二叉搜索树、红黑树和 2−3 树做为实例化的案例来展示, 并借助自动化验证框架, 可自动验证多个实例化案例. 这为复杂数据结构算法功能和结构正确性的自动化验证提供了参考.

在未来的工作中, 可以考虑将本文中的函数转化为更具可执行性和效率的编程语言, 如 Haskell 或 OCaml. 将形式化方法的理论应用于实际问题, 将函数式程序建模及其验证投入实际的工业应用中, 探索其在实际应用中的效果和优势, 使其能够达到工业级的要求.

References:

[1] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms. 4th ed., Cambridge: MIT Press, 2022.

[2] Hopcroft JE, Ullman JD. Set merging algorithms. SIAM Journal on Computing, 1973, 2(4): 294–303. [doi: 10.1137/0202024]

[3] Fisher RA. Statistical methods for research workers. In: Kotz S, Johnson NL, eds. Breakthroughs in Statistics: Methodology and Distribution. New York: Springer, 1992. 66–70. [doi: 10.1007/978-1-4612-4380-9_6]

[4] Ahsanullah M, Nevzorov VB, Shakil M. An Introduction to Order Statistics. Paris: Atlantis Press, 2013. [doi: 10.2991/978-94-91216-83-1]

[5] Nipkow T. Automatic functional correctness proofs for functional search trees. In: Proc. of the 7th Int'l Conf. on Interactive Theorem Proving. Nancy: Springer, 2016. 307–322. [doi: 10.1007/978-3-319-43144-4_19]

[6] Nipkow T, Klein G. Concrete Semantics: With Isabelle/HOL. Cham: Springer, 2014. [doi: 10.1007/978-3-319-10542-0]

[7] Nipkow T, Paulson LC, Wenzel M. A Proof Assistant for Higher-order Logic. New York: Springer, 2021.

[8] Böhme S, Nipkow T. Sledgehammer: Judgement day. In: Proc. of the 5th Int'l Joint Conf. of Automated Reasoning. Edinburgh: Springer, 2010. 16–19 [doi: 10.1007/978-3-642-14203-1_9]

[9] Eberl M, Haslbeck MW, Nipkow T. Verified analysis of random binary tree structures. Journal of Automated Reasoning, 2020, 64(5): 879–910. [doi: 10.1007/s10817-020-09545-0]

[10] Nipkow T, Sewell T. Proof pearl: Braun trees. In: Proc. of the 9th ACM SIGPLAN Int'l Conf. on Certified Programs and Proofs. New Orleans: ACM, 2020. 18–31. [doi: 10.1145/3372885.3373834]

[11] Nipkow T. Verified root-balanced trees. In: Proc. of the 15th Asian Symp. on Programming Languages and Systems. Suzhou: Springer, 2017. 255–272. [doi: 10.1007/978-3-319-71237-6_13]

[12] Nipkow T, Brinkop H. Amortized complexity verified. Journal of Automated Reasoning, 2019, 62(3): 367–391. [doi: 10.1007/s10817-018-9459-3]

[13] Hinze R, Paterson R. Finger trees: A simple general-purpose data structure. Journal of Functional Programming, 2006, 16(2): 197–217. [doi: 10.1017/S0956796805005769]

[14] Hirai Y, Yamamoto K. Balancing weight-balanced trees. Journal of Functional Programming, 2011, 21(3): 287–307. [doi: 10.1017/S0956796811000104]

[15] Ammer T, Lammich P. van Emde boas trees. 2024. https://www.isa-afp.org/browser_info/current/AFP/Van_Emde_Boas_Trees/document.pdf

[16] Zuo ZK, Huang ZP, Huang Q, Sun H, Zeng ZC, Hu Y, Wang CJ. Functional modeling and mechanized verification of LLRB algorithm. Ruan Jian Xue Bao/Journal of Software, 2024, 35(11): 5016–5039 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7034.htm [doi: 10.13328/j.cnki.jos.007034]

[17] Zhao YW. Functional programming and proof. 2021 (in Chinese). https://www.yuque.com/zhaoyongwang/fpp/

[18] Back RJ. Invariant based programming: Basic approach and teaching experiences. Formal Aspects of Computing, 2009, 21(3): 227–244. [doi: 10.1007/s00165-008-0070-y]

[19] Nipkow T. Are homomorphisms sufficient for behavioural implementations of deterministic and nondeterministic data types? In: Proc. of the 4th Annual Symp. on Theoretical Aspects of Computer Science. Passau: Springer, 1987. 260–271. [doi: 10.1007/BFb0039611]

[20] Haftmann F, Krauss A, Kunčar O, Nipkow T. Data refinement in Isabelle/HOL. In: Proc. of the 4th Int'l Conf. of Interactive Theorem

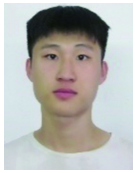
Proving. Rennes: Springer, 2013. 100–115. [doi: 10.1007/978-3-642-39634-2_10]
[21] Bayer R. Symmetric binary B-trees: Data structure and maintenance algorithms. Acta Informatica, 1972, 1(4): 290–306. [doi: 10.1007/BF00289509]
[22] Guibas LJ, Sedgewick R. A dichromatic framework for balanced trees. In: Proc. of the 19th Annual Symp. on Foundations of Computer Science. Ann Arbor: IEEE, 1978. 8–21. [doi: 10.1109/SFCS.1978.3]
[23] Mündler N, Nipkow T. A verified implementation of B+-trees in Isabelle/HOL. In: Proc. of the 19th Int'l Colloquium on Theoretical Aspects of Computing. Tbilisi: Springer, 2022. 324–341. [doi: 10.1007/978-3-031-17715-6_21]

附中文参考文献:

[16] 左正康, 黄志鹏, 黄箐, 孙欢, 曾志城, 胡颖, 王昌晶. LLRB 算法的函数式建模及其机械化验证. 软件学报, 2024, 35(11): 5016–5039. http://www.jos.org.cn/1000-9825/7034.htm [doi: 10.13328/j.cnki.jos.007034]
[17] 赵永望. 函数式程序设计与证明. 2021. https://www.yuque.com/zhaoyongwang/fpp/

左正康(1980－), 男, 博士, 教授, CCF 高级会员, 主要研究领域为形式化方法, 智能化软件.

游珍(1982－), 女, 博士, 副教授, CCF 高级会员, 主要研究领域为形式化方法, 分布式虚拟现实, 并发分布式计算.



刘增鑫(1999－), 男, 硕士生, CCF 学生会员, 主要研究领域为定理证明, 形式化方法.

王昌晶(1977－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为高可信软件, 智能化软件.

柯雨含(1998－), 男, 硕士生, 主要研究领域为定理证明, 形式化方法.