软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software [doi: 10.13328/j.cnki.jos.007034]

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

LLRB 算法的函数式建模及其机械化验证*

左正康[1], 黄志鹏[1], 黄 箐[1], 孙 欢[2], 曾志城[1], 胡 颖[1], 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](江西师范大学 数字产业学院, 江西 上饶 334006)

通信作者: 王昌晶, E-mail: wcj@jxnu.edu.cn

摘 要: 基于机器定理证明的形式化验证技术不受状态空间限制, 是保证软件正确性、避免因潜在软件缺陷带来严重损失的重要方法. LLRB (left-leaning red-black trees) 是一种二叉搜索树变体, 其结构比传统的红黑树添加了额外的左倾约束条件, 在验证时无法使用常规的证明策略, 需要更多的人工干预和努力, 其正确性验证是一个公认的难题. 为此, 基于二叉搜索树类算法 Isabelle 验证框架, 对其附加性质部分进行细化, 并给出具体化的验证方案. 在 Isabelle 中对 LLRB 插入和删除操作进行函数式建模, 对其不变量进行模块化处理, 并验证函数的正确性. 这是首次在 Isabelle 中对函数式 LLRB 插入和删除算法进行机械化验证, 相较于目前 LLRB 算法的 Dafny 验证, 定理数由 158 减少到 84, 且无需构造中间断言, 减轻了验证的负担; 同时, 为复杂树结构算法的函数式建模及验证提供了一定的参考价值.

关键词: LLRB; 函数式建模; 机械化验证; Isabelle 定理证明器; 二叉搜索树

中图法分类号: TP311

中文引用格式: 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. http://www.jos.org.cn/1000-9825/7034.htm

英文引用格式: 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 (in Chinese). http://www.jos.org.cn/1000-9825/7034.htm

Functional Modeling and Mechanized Verification of LLRB Algorithm

ZUO Zheng-Kang[1], HUANG Zhi-Peng[1], HUANG Qing[1], SUN Huan[2], ZENG Zhi-Cheng[1], HU Ying[1], WANG Chang-Jing[1]

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

[2](School of Digital Industry, Jiangxi Normal University, Shangrao 334006, China)

Abstract: Unlimited by the state and space, the formal verification technology based on mechanized theorem proof is an important method to ensure software correctness and avoid serious loss from potential software bugs. LLRB (left-leaning red-black trees) is a variant of binary search trees, and its structure has an additional left-leaning constraint over the traditional red-black trees. During verification, conventional proof strategies cannot be employed, which requires more manual intervention and effort. Thus, the LLRB correctness verification is widely acknowledged as a challenging problem. To this end, based on the Isabelle verification framework for the binary search tree algorithm, this study refines the additional property part of the framework and provides a concrete verification scheme. The LLRB insertion and deletion operations are functionally modeled in Isabelle, with modular treatment of the LLRB invariants. Subsequently, the function correctness is verified. This is the first mechanized verification of functional LLRB insertion and deletion algorithms in Isabelle. Compared to the current Dafny verification of the LLRB algorithm, the theorem number is reduced from 158 to 84, and it is unnecessary for constructing intermediate assertions, which alleviates the verification burden. Meanwhile, this study provides references for

* 基金项目: 国家自然科学基金 (61862033, 62262031); 江西省自然科学基金 (20212BAB202018); 江西省教育厅科技重点项目 (GJJ 210307)

收稿时间: 2022-12-15; 修改时间: 2023-03-23, 2023-06-12; 采用时间: 2023-07-17; jos 在线出版时间: 2023-12-06

functional modeling and verification of complex tree structure algorithms.

Key words: left-leaning red-black trees (LLRB); functional modeling; mechanized verification; Isabelle theorem prover; binary search tree

二叉搜索树类结构能够提供有效的内存空间分配、有规则的数据存储和支持强大的搜索算法, 广泛应用于信息加密、数据解压和内存管理等领域. 然而在形式化验证领域, 对于涉及元素集合的复杂数据结构, 其形式化规约的生成和功能的正确性验证是具有挑战性的[1]. 树形数据结构在验证系统中被定义为验证基准, RB (red-black trees) 作为二叉搜索树的变体, 更是这组基准中最经典且难度最大的[2]. 因而对二叉搜索树类算法的实现和机械化验证进行研究具有一定意义.

机械化定理证明是保证程序正确性的有效途径, 使用机械化定理证明工具对算法性质进行机器证明已成为一种发展趋势[3]. Isabelle/HOL 是当前被广泛使用的、LCF 方法的机械化定理证明器[4], 对于树这种较复杂数据结构的算法进行正确性验证更为适宜. LLRB 是由 Sedgewick 提出的一种二叉搜索树变体[5], 其结构比传统的 RB 结构多了“左倾”的约束条件. 在验证 LLRB 算法的正确性时, 需验证所有子树是否满足左倾结构, 还需考虑自平衡调整对“左倾”结构的影响. 换言之, 其验证时需要处理更多的特殊情况, 无法直接使用常规的证明策略, 需要更多的人工干预和努力[6]. 尤其是在删除操作中, 删除节点后不仅要考虑“黑高相同”“不能连红”等常规性质, 还需考虑“左倾”特性是否被破坏. 这就意味着需要考虑更复杂的不变量, 其正确性验证需要更多的创造性思维和技巧[7].

本文在 Isabelle 中对 LLRB 结构及其插入和删除操作进行函数式建模, 并对其性质进行划分: 1) 基本性质, 即中序遍历节点的键值是线性升序的. 2) 附加性质, 相较于标准二叉搜索树所需满足的额外性质, 即 LLRB 的颜色、高度等特性. 最后验证了 LLRB 函数式算法的功能正确性. 本文工作的主要贡献总结如下.

(1) 通过探究二叉搜索树类算法之间共性, 基于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、RB、LLRB 等) 进行建模时, 可进行相应的实例化.

(2) Nipkow[8]提出的 Isabelle 验证框架具有很强的通用性. 为了涵盖二叉搜索树类所有算法, 对附加性质只进行了概述. 本文通过挖掘验证规约与辅助引理之间的关系, 对附加性质的验证进行细化, 给出具体化的验证方案, 该方案对解决 LLRB 算法的正确性验证有效.

(3) 基于二叉搜索树类结构的插入和删除高阶泛化函数, 实例化生成 LLRB 的函数式算法, 并基于贡献 (2), 首次在 Isabelle 中给出了函数式 LLRB 插入和删除算法的机械化验证, 相较于目前 LLRB 算法的 Dafny 验证, 定理数由 158 减少到 84, 且无需构造中间断言, 减轻了验证的负担.

本文第 1 节是对相关工作进行介绍和比较. 在第 2 节二叉搜索树类算法的函数式建模框架中, 用区域刻画了其插入和删除的高阶泛化函数. 第 3 节对二叉搜索树类算法的 Isabelle 验证框架的附加性质进行细化, 并给出具体化的验证方案. 第 4 节以 LLRB 算法为实例, 对其进行 Isabelle 函数式建模. 第 5 节对 LLRB 的算法性质进行正确性验证. 第 6 节与现有研究的实验进行对比. 第 7 节是对全文的总结以及未来工作展望.

1 相关工作

近年来, 针对 RBs 类 (包括 RB, LLRB, RLRB 等) 算法的研究, 主要包括算法实现和机械化验证两个方面.

RBs 类算法实现: 二叉搜索树类结构的插入和删除操作往往需要考虑大量不同的情况[9], 同时 RBs 类算法也继承了其复杂性. 文献 [10] 提出的 RB 算法实现在插入过程中有 6 种修复不变量的情况, 且对于删除操作的自平衡引入了 8 种情况, 仅删除操作的代码量就有 80 行. 文献 [11] 提出了一个更为简单的 RB 版本, 称为 AA 树, 在插入或删除过程中引入 *skew* 和 *split* 两个基本的变换以修复不变量, 然而连续进行这些变换时最多可能需要 5 次, 插入和删除的自平衡代码大约有 100 行的 Java 代码. 上述有关 RBs 算法的命令式程序比较复杂, 需考虑的自平衡情况较多. 而函数式程序可以提高代码的复用性, 代码实现更为简洁, 且不会产生任何改变程序状态的副作用[12]. 文献 [13] 提出了 RB 算法的函数式版本. 该版本虽将自平衡的情况减少到 4 种, 但其内部节点的左右子节点可以是红色或黑色[6], 限制较少, 且没有给出删除操作的算法.

RBs 类算法的机械化验证: 基于机器定理证明的形式化验证技术不受状态空间限制, 是保证软件正确性、避免因潜在软件缺陷带来严重损失的重要方法. 文献 [14] 通过引入模块 (*module*) 和函子 (*functor*), 在 Coq 辅助证明工具中验证了基于有限集 (*finite sets*) 实现的 RB 算法, 文献 [7] 在其基础上, 提出了一种更为高效且经过 Coq 验证的 RB 算法. 上述文献是基于有限集的实现方式提供的规约表示, 由于有限集内的元素的无序性, 对于二叉搜索树类结构的有序性质需要进行额外的刻画, 这给算法的正确性及终止性的机械化验证带来极大的困难. Why3 资源库中[15]实现了 RB 算法的函数式版本, 并带有大量的辅助引理证明其基本操作的正确性, 但不包括删除操作及其正确性验证. Nipkow 改进了传统的基于有限集 (*finite sets*) 的实现方案, 针对多类二叉搜索树线性升序的性质提出了验证框架[8], 并在其基础上对 RB 算法进行了完整的机械化验证[16], 但为了追求框架的通用性, 对于附加结构性质具体化的验证方案尚未给出. 文献 [6] 利用 Dafny 平台验证了 LLRB 插入和删除操作, 证明了相应函数的终止性和正确性. 然而相较于 Isabelle 定理证明器, Dafny 的定理 (requires+ensures) 主要通过断言 (assert) 来描述, 难以继承[17], 而 Isabelle 的定理 (theorem+lemma) 支持模块化的设计, 可通过定义新的类型、常量和规则对父理论进行扩充, 有效地避免了重复的理论定义; Dafny 只能为算法进行证明推理, 不可生成函数式可执行代码[17], 而 Isabelle 中大多数定义和函数都达到了可执行级别, 也可以自动转换为其他函数式编程语言 (如 Haskell、ML 等)[18]; Dafny 是一种命令式与函数式混合的描述语言, 在验证时探索前后置条件以及大量的中间断言是一段相当痛苦的过程[6].

本文基于 Nipkow[8]提出的二叉搜索树类算法的 Isabelle 验证框架, 改进了传统的基于有限集 (*finite sets*) 的实现方案, 使用 *inorder* 方法将树映射到有序列表 (*sorted list*), 可直接刻画二叉搜索树类结构的有序性质. 并对其附加性质的验证进行细化, 给出具体化的验证方案. 进一步以 LLRB 为实例, 在 Isabelle 中给出了 LLRB 的完整函数式设计算法, 包括 LLRB 的结构定义、不变量的划分以及插入、删除操作的实现. 首次在 Isabelle 中给出了该数据结构包括删除在内的函数式证明, 并从验证过程、验证脚本的模块化和代码的可执行性 3 个方面与相关工作进行对比, 展示了本文提出的插入和删除的高阶泛化函数以及针对附加性质提出的具体化验证方案的有效性.

2 二叉搜索树类算法的函数式建模框架

二叉搜索树类结构包括标准体和多种变体, 在对该类算法进行函数式建模时, 其结构、不变量和基本操作的定义在整体上都符合一定的模式. 对于插入和删除操作, 基于文献 [8] 的二叉搜索树类算法的函数式建模框架, 我们用区域 (*locale*) 刻画了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进行建模时, 可进行相应的实例化, 更有利于函数式设计.

2.1 二叉搜索树类结构的函数式定义

二叉搜索树类结构能够有效地存储和访问元素的集合, 允许元素的快速搜索、删除和插入, 可用于实现动态集合 (dynamic sets)、查找表 (lookup tables) 等数据结构[12]. 对于标准二叉搜索树 (简称 *search_tree*), 其数据类型可以递归地定义为[8]:

$$datatype\ 'a\ search\_tree\ =\ Leaf\ |\ Node\ ('a\ search\_tree)\ 'a\ ('a\ search\_tree).$$

在上述定义的基础上，我们给出了如下缩写描述:

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

对于二叉搜索树变体 (简称为 *BsTreeVar*, 在后文可实例化为某种具体的二叉搜索树), 其节点可能含有附加属性 (如颜色、高度等, 统称为 *add_ prop*). 在 Isabelle 内部, *type_synonym* 可以被完全展开而不被输出. 在提高理论可读性的前提下, *type_synonym* 可以像其他类型一样被使用[18], 也可在继承某一数据类型的基础上加以扩充. 基于此, 定义了 *search_tree* 的类型同义词 *BsTreeVar*:

$$type\_synonym\ 'a\ BsTreeVar\ =\ ('a\ \times\ add\_prop)\ search\_tree.$$

2.2 二叉搜索树类结构的不变量

不变量是指程序在执行过程中必须遵守的逻辑规则, 对程序进行验证时, 可以通过证明不变量在程序执行前

后为真来完成[19]. 二叉搜索树类结构的性质可以被定义为[8]: 基本性质, 即中序遍历节点的键值按照线性升序的方式排序, 以及附加性质 (如颜色、高度等). 维持基本性质不变的逻辑规则称为基本不变量 (简称 *bst*), 维持附加性质不变的逻辑规则称为结构不变量 (简称 *inv*), 两者统称为二叉搜索树类的不变量 (简称 *invar*), 这些性质在进行基本操作 (如查找、插入和删除) 前后需维持不变, 从而证明相应函数的正确性.

二叉搜索树类结构支持集合和映射, 在 Isabelle 中, 抽象函数 *set_search_tree* 实现了将树映射到其元素集合[8]:

$$\begin{cases} set\_search\_tree\ ::\ 'a\ search\_tree\ \Rightarrow\ 'a\ set \\ set\_search\_tree\ <>\ =\ \{\} \\ set\_search\_tree\ <l,a,r>\ =\ set\_search\_tree\ l\ \cup\ \{a\}\ \cup\ set\_search\_tree\ r \end{cases}.$$

基于抽象函数 *set_search_tree*, 可对二叉搜索树类结构的基本不变量 *bst* 进行函数式建模, 其定义如下:

$$\begin{cases} bst\ ::\ ('a::linorder)\ search\_tree\ \Rightarrow\ bool \\ bst\ <>\ =\ True \\ bst\ <l,a,r>\ =\ ((\forall x \in set\_search\_tree\ l.\ x<a)\ \wedge\ (\forall x \in set\_search\_tree\ r.\ a<x)\ \wedge\ bst\ l \wedge\ bst\ r) \end{cases},$$

其中, *'a* 类型元素满足全序关系 (后文中所有*'a* 的出现都是如此), 且对于二叉搜索树类结构的不变量, 其输出都为 *bool* 类型 (*True* 或者 *False*). 另外, 对于二叉搜索树类变体的附加性质的不变量定义, 可通过抽象函数 *inv* :: *'a BsTreeVar* ⇒ *bool* 来表示. 因此标准二叉搜索树和二叉搜索树变体的不变量可以分别被定义为:

$$\begin{cases} invar\ search\_tree\ =\ bst\ search\_tree \\ invar\ BsTreeVar\ =\ bst\ BsTreeVar\ \wedge\ inv\ BsTreeVar \end{cases}.$$

2.3 二叉搜索树类结构查找、插入和删除的高阶泛化函数

二叉搜索树类结构的基本操作包括对节点键值的查找、插入和删除, 作为具体的数据类型, 本文用抽象类型 *set* 来实现其一系列操作. 通过定义固定的接口 (即一组操作 *lookup*, *ins*, *del*), 从而可以操作抽象数据类型的值[8], 其中 *'a* 是 *set* 的元素类型:

$$\begin{cases} lookup\ ::\ 'a\ BsTreeVar\ \Rightarrow\ 'a\ \Rightarrow\ bool \\ ins\ ::\ 'a\ \Rightarrow\ 'a\ BsTreeVar\ \Rightarrow\ 'a\ BsTreeVar. \\ del\ ::\ 'a\ \Rightarrow\ 'a\ BsTreeVar\ \Rightarrow\ 'a\ BsTreeVar \end{cases}$$

二叉搜索树类结构的一系列操作的函数式实现基于比较运算函数 *cmp*, 该函数返回 *datatype cmp_val = LT | EQ | GT* (小于|等于|大于) 的一种情况. 这使得函数式代码更加对称, 且相比于通过=、< 和> 来比较元素拥有更高的效率, 具体定义如下:

$$\begin{cases} datatype\ cmp\_val =\ LT|EQ|GT \\ cmp\ ::\ 'a \Rightarrow\ 'a\ \Rightarrow\ cmp\_val \\ cmp\ x\ y =\ (\text{if}\ x<y\ \text{then}\ LT\ \text{else if}\ x=y\ \text{then}\ EQ\ \text{else}\ GT) \end{cases}.$$

(1) 查找

根据二叉搜索树的特征, 查找算法属于二分查找, 为了简化运算, 查找函数返回 *True* 或 *False*, 表示能否找到某个键值 *x*. *lookup* 函数的实现框架如下:

$$\begin{cases} lookup\ <\ >\ \_ =\ False \\ lookup\ <l,(a,\_),r>\ x =\ (case\ cmp\ x\ a\ \text{of}\ LT \Rightarrow\ lookup\ l\ x \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad GT \Rightarrow\ lookup\ r\ x \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad EQ \Rightarrow\ True) \end{cases}.$$

(2) 插入和删除

在这一部分中, 首先使用 Isabelle 的区域 (*locale*) 来定义高阶泛化函数, 用于插入和删除操作. 区域是一种程序模块化和参数化的复用机制, 能充分表达函数式程序结构之间复杂的依赖关系[20]. 通过区域声明, 可定义通用的泛化数据类型和函数, 并通过解释对其进行实例化, 以实现复用. 其结构如下:

locale loc_name=//区域名
  fixes x_1:: τ_1 //声明参数
  …
  fixes x_n:: τ_n
  begin//区域体
    definition/*fun*/*function*/*primrec* a_1:: ϕ_1 //泛化函数
      …
    definition/*fun*/*function*/*primrec* a_n:: ϕ_n
    end

本文的区域声明包括两部分: 区域名 *loc_name* 和局部参数 *fixes* (接口). 接口 $x_1 :: \tau_1$ 表示接口名 x_1 及其类型 τ_1, 其中 τ_1 可以是数据类型、函数类型或任意多态类型. 区域声明之后, 可以定义一个由 *begin-end* 块构成的区域体, 区域体内部可定义一系列的泛化函数 *definition*/*fun*/*function*/*primrec*, 这些泛化函数可调用区域声明中的接口. 泛化函数 $a_1 :: \phi_1$ 定义了函数 a_1 及其函数类型 ϕ_1.

区域声明并定义区域体后, 可通过解释 (*interpretation*) 将该区域中声明的接口和已有的泛化函数实例化, 使它们在当前上下文中可被复用. *interpretation* 一般语法如下:

$$inorder\ \ t_1\ @\ a_1\ \#\ inorder\ t_2\ @\ a_2\ \#\ \ldots\ \#\ inorder\ \ t_n.$$

在本文中, *loc_name* 是需要实例化的区域名, $loc\_instance_{1\ldots}loc\_instance_n$ 是用于实例化区域 *loc_name* 中接口 *fixes* 的具体函数.

考虑到二叉搜索树变体的结构不变量可能会随元素 *x* 的插入或删除而被破坏, 本节声明了区域 *BsTreeVar_op* 中定义了接口 pre_inv_l、pre_inv_r 和 pre_inv_{split}, 分别针对左子树、右子树和删除中的 *split* 操作来维持结构不变量 *inv*. 使用这 3 个接口可定义二叉搜索树插入和删除操作的泛化函数 *ins'*、*del'*. *BsTreeVar_op* 构造了二叉搜索树类结构的基本操作, 可进行具体的实例化, 这一点在第 4.2 节详细阐述. 接口的定义如下:

$$\begin{aligned}
&locale\ \ BsTreeVar_op = \\
&\quad fixes\ \ pre_inv_l :: \text{“}'a\ \ BsTreeVar \Rightarrow 'a\ \ BsTreeVar \Rightarrow 'a \Rightarrow 'a\ \ BsTreeVar \Rightarrow 'a\ \ BsTreeVar\text{”} \\
&\quad and\ \ pre_inv_r :: \text{“}'a\ \ BsTreeVar \Rightarrow 'a \Rightarrow 'a\ \ BsTreeVar \Rightarrow 'a\ \ BsTreeVar \Rightarrow 'a\ \ BsTreeVar\text{”} \\
&\quad and\ \ pre_inv_{split} :: \text{“}'a\ \ BsTreeVar \Rightarrow 'a \Rightarrow 'a\ \ BsTreeVar \Rightarrow 'a\ \ BsTreeVar\text{”}
\end{aligned}$$

插入算法基于二分查找从根节点逐步向下搜索合适的位置, 并将新的元素插入, 可分为两种情况.

1) 若键值 *x* 插入到空树< >, 则将该叶子直接替换键值为 *x* 的内部节点.

2) 若键值 *x* 插入到非空树<*l*, (*a*, _), *r*>, 则需调用 *cmp* 函数比较 *x* 与当前内部节点键值 *a* 的大小, 若相等 (*EQ*) 则表明已有该键值, 不做插入; 否则, 按 *x* 与 *a* 的大于 (*GT*) 和小于 (*LT*) 情况, 递归地将 *x* 插入到右子树或左子树中.

ins' x t 表示插入元素 *x* 到二叉搜索树 *t* 中, 并返回插入 *x* 后的新树, *ins'*的高阶泛化函数如下:

$$\begin{aligned}
&fun\ \ ins' :: \text{“}'a \Rightarrow 'a\ \ BsTreeVar \Rightarrow 'a\ \ BsTreeVar\text{”} \\
&\quad ins'\ x <>=<<>,(x,_),<>> \\
&\quad ins'\ x < l,(a,_),r >= (case\ \ cmp\ \ x\ \ a\ \ \text{of}\ \ LT \Rightarrow pre_inv_l\ (ins'\ x\ l)\ l\ a\ r \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad GT \Rightarrow pre_inv_r\ l\ a\ r\ (ins'\ x\ r) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad EQ \Rightarrow l\ a\ r)
\end{aligned}$$

二叉搜索树类结构的删除操作相较于插入操作更为复杂, 被删除元素可能存在一定数量的左右子树, 这种情况下要考虑其左右子树如何与上方的树节点进行连接或合并. 这里需要考虑以下两种情况.

1) 若待删除节点的右子树为叶子节点, 则将其左子树合并到其父节点下.

2) 若待删除节点的右子树是内部节点, 则从右子树中删除最小的元素, 并将其放到待删除节点的位置, 从而维持新的树仍然满足二叉搜索树的性质.

del' x t 表示从二叉搜索树 *t* 中删除元素 *x*, 并返回删除 *x* 后的新树, *del'*的高阶泛化函数如下:

$$\begin{aligned}&fun\ del' :: "'a \Rightarrow 'a\ BsTreeVar \Rightarrow 'a\ BsTreeVar"\\&\quad del'\ x <>=<>\\&\quad del'\ x < l,(a,\_),r >= (case\ \ cmp\ \ x\ a\ of\ \ LT \Rightarrow pre\_inv_l\ (del'\ x\ l)\ l\ a\ r\\&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad GT \Rightarrow pre\_inv_r\ l\ a\ r\ (del'\ x\ r)\\&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad EQ \Rightarrow pre\_inv_{split}\ l\ a\ r)\end{aligned}$$

3 二叉搜索树类算法的 Isabelle 验证框架

为了在 Isabelle 中验证二叉搜索树类算法的正确性, 第 3.1 节和第 3.2 节概述了 Nipkow 在文献 [8] 中提出的 *set* 实现原理、*inorder* 方法, 并基于上述原理和方法给出了维持二叉搜索树类结构基本不变量 *bst* 的规约.

对于基本不变量 *bst*, 第 3.3 节基于文献 [8] 的 Isabelle 验证框架, 添加了平衡函数引理集和删除操作中分离函数的引理集; 并通过挖掘验证规约与辅助引理之间的关系, 给出了相应的辅助引理选用依据.

对于结构不变量 *inv*, 鉴于文献 [8] 提出的 Isabelle 验证框架是通用的, 对附加性质只进行了概述, 本文对附加性质进行细化, 给出了具体化的验证方案. 在第 3.4 节中, 我们首先给出了证明附加性质的规约, 并通过分析附加性质与平衡函数之间的关系从而探索辅助引理之间的共性, 依据插入和删除操作是否会破坏结构不变量 *inv* 划分为两类问题, 进一步构造证明辅助引理集以提高验证效率, 最大化地将创造性劳动转化为非创造性劳动. 使得对于此类较为复杂的数据结构, 机械化验证能够按照一定的模式进行.

3.1 *set* 实现原理

使用同态的抽象函数表示规约可以追溯到 Hoare[21], 并成为面向模型的规约语言 VDM[22]的一个组成部分. 在通用代数的一阶环境下, 存在完全抽象的模型来表示规约[23]. 二叉搜索树这类数据结构的实现已证明与 *set* 同态[17], 即二叉搜索树上的操作 *lookup*, *insert* 和 *delete* 可以用 *set* 中的 $\in$ 、 $\cup$ 和–来模拟. 例如规约 *lookup s x* 可用 *x* $\in$ *set s* 来表示.

在讨论函数 *lookup*, *insert* 和 *delete* 在基本不变量 *bst* 上的功能正确性时, 为了指定这些操作, 需要假设抽象函数 *set_search_tree* :: *'a search_tree* $\Rightarrow$ *'a set* 和二叉搜索树的不变量 *invar* :: *'a search_tree* $\Rightarrow$ *bool* . 图 1 为 *set* 实现的相关规约.

$$\begin{aligned}&invar\ \ BsTreeVar \Rightarrow set\_search\_tree(insert\ \ x\ \ BsTreeVar) = \{x\} \cup\ \ set\_search\_tree\ \ BsTreeVar\\&invar\ \ BsTreeVar \Rightarrow\ \ set\_search\_tree(delete\ \ x\ \ BsTreeVar) =\ \ set\_search\_tree\ \ BsTreeVar - \{x\}\\&invar\ \ BsTreeVar \Rightarrow lookup\ \ BsTreeVar\ \ x = (x \in\ \ set\_search\_tree\ \ BsTreeVar)\\&invar\ \ BsTreeVar \Rightarrow invar(insert\ \ x\ \ BsTreeVar)\\&invar\ \ BsTreeVar \Rightarrow invar(delete\ \ x\ \ BsTreeVar)\end{aligned}$$

图 1 *set* 实现的规约

3.2 *inorder* 方法

在图 1 中用 *set* 来表示二叉搜索树基本操作的验证规约, 并且基于抽象函数 *set_search_tree*, 二叉搜索树的基本不变量 *bst* 可以在 Isabelle 中进行表达. 但实际的验证是基于 *list* 来完成, 因为中序遍历 (*inorder*) 可以很好地表示线性升序关系, 且中序遍历产生的 *list* 也可以很容易转换为集合, 使用具体数据类型 *list* 作为 *set* 和 *search_tree* 之间的中间数据类型, 简称 *inorder* 方法[8]. 当对 *t* 进行归纳时, 会产生形如< <t_1, a_1, t_2>, a_2, <t_3, a_3, t_4> > 的嵌套树结构, 对树结构进行 *inorder* 遍历会产生以下形式的 *list*:

$$inorder\ \ t_1\ @\ a_1\ \#\ inorder\ t_2\ @\ a_2\ \#\ \ldots\ \#\ inorder\ \ t_n.$$

把函数 *sorted* 应用到上述公式中, 得到 *sorted* (xs_1 @ a_1 # xs_2 @ a_2 # … # xs_n), 可以被分解为以下基本公式:

$$\begin{cases} sorted(xs@[a]) \overset{模拟}{\rightarrow} (\forall x \in set\ \ xs.\ \ x < a)\\ sorted(a\#xs) \overset{模拟}{\rightarrow} (\forall x \in set\ \ xs.\ \ a < x)\end{cases}.$$

通过基本公式, 有序的 *list* 在 Isabelle 中模拟了 *set* 的有序性, 从而解释了使用 *list* 作为中间类型证明基本不变量 *bst* 的原因. 以树的 *inorder* 遍历为基础, 从而表示 *inorder* :: *'a t* ⇒ *'a list* 分别与 *bst* 和 *set_search_tree* 之间的关系:

$$\begin{cases} bst\ BsTreeVar\ =\ sorted(inorder\ BsTreeVar) \\ set\_search\_tree\ BsTreeVar\ =\ set(inorder\ BsTreeVar) \end{cases}.$$

为了方便函数 *lookup*, *insert* 和 *delete* 在列表上的表达, 下面定义了 4 个关于列表的辅助函数:

$$\begin{cases} sorted\ ::\ 'a\ list\ \Rightarrow\ bool \\ sorted\ [\,]\ =\ True \\ sorted\ [x]\ =\ True \\ sorted(x\#y\#zs)\ =\ (x<y\ \wedge\ sorted(y\#zs)) \end{cases},$$

其中, *sorted* 表示列表按升序排序.

$$\begin{cases} ins_{list}\ ::\ 'a\ \Rightarrow\ 'a\ list \Rightarrow\ 'a\ list \\ ins_{list}\ x\ [\,]\ =\ [x] \\ ins_{list}\ x\ [a\#xs]\ = \\ (\text{if}\ x<a\ \text{then}\ x\#a\#xs\ \text{else if}\ x=a\ \text{then}\ a\#xs\ \text{else}\ a\#\ ins_{list}\ x\ xs) \end{cases},$$

其中, ins_{list} 表示如果某一元素尚未出现在列表中, 则将该元素插入到顺序列表中的正确位置.

$$\begin{cases} del_{list}\ ::\ 'a\ \Rightarrow\ 'a\ list \Rightarrow\ 'a\ list \\ del_{list}\ x\ [\,]\ =\ [\,] \\ del_{list}\ x\ [a\#xs]\ =\ (\text{if}\ x=a\ \text{then}\ xs\ \text{else}\ a\#\ del_{list}\ x\ xs) \end{cases},$$

其中, del_{list} 表示根据给定的元素值, 删除列表中第 1 个等于该值的元素.

$$\begin{cases} elems\ ::\ 'a\ list \Rightarrow\ 'a\ set \\ elems\ [\,]\ =\ \varnothing \\ elems(a\#xs)\ =\ \{a\}\ \cup\ elems\ xs \end{cases},$$

其中, *elems* 将一个列表转换为它的元素集合.

在上述规约函数 *inorder* 中, *'a t* 为二叉搜索树的类型. 对于任意的一棵二叉搜索树 *t*, 基本不变量 *bst* 可以被表示为 *sorted*(*inorder t*). 在 *inorder* 方法中, 定义了关于列表的 4 个辅助函数: *sorted*, ins_{list}, del_{list}, *elems*, 从而构造在有序类型上基于 *inorder* 方法实现的相关规约.

图 2 中的规约 (1)–(3) 表达了 *insert*, *delete* 和 *lookup* 在基本不变量 *bst* 上的功能正确性. 如果把抽象函数 *set* :: *'a t* ⇒ *'a set* 解释为函数 *elems* 和函数 *inorder* 的复合: $elems \circ inorder$, 使用图 3 中列表操作与集合关系的相关辅助引理, 可以证明图 2 中在有序类型上 *inorder* 方法实现的规约与图 1 中的规约等价[8].

$$invar\ BsTreeVar \Rightarrow inorder(insert\ x\ BsTreeVar) = ins_{list}\ x\ inorder(BsTreeVar) \quad (1)$$
$$invar\ BsTreeVar \Rightarrow inorder(delete\ x\ BsTreeVar) = del_{list}\ x\ inorder(BsTreeVar) \quad (2)$$
$$invar\ BsTreeVar \Rightarrow lookup\ BsTreeVar\ x = (x \in elems\ inorder(BsTreeVar)) \quad (3)$$
$$invar\ BsTreeVar \Rightarrow inv(insert\ x\ BsTreeVar)$$
$$invar\ BsTreeVar \Rightarrow inv(delete\ x\ BsTreeVar)$$

图 2　在有序类型上基于 *inorder* 方法实现的规约

$$elems(ins_{list}\ x\ xs) = \{x\}\ \cup\ elems\ xs$$
$$sorted\ xs \Rightarrow distinct\ xs$$
$$distinct\ xs \Rightarrow elems(del_{list}\ x\ xs) = elems\ xs - \{x\}$$
$$sorted\ xs \Rightarrow sorted(ins_{list}\ x\ xs)$$
$$sorted\ xs \Rightarrow sorted(del_{list}\ x\ xs)$$

图 3　列表操作与集合关系的相关辅助引理

3.3 基本不变量 *bst* 的验证框架

对于 *bst* 的验证框架, 基于 *set* 实现和 *inorder* 方法, 特别是针对二叉搜索树变体, 给出了相应的验证引理集, 并给出了辅助引理的选用策略, 克服引理选用的盲目性. 下面是基本不变量 *bst* 在 *lookup*, *insert* 和 *delete* 操作前后不变量维持所需的规约:

$$\begin{cases} sorted(inorder\ t) \Rightarrow inorder(insert\ x\ t) = ins_{list}\ x\ (inorder\ t) & T_1 \\ sorted(inorder\ t) \Rightarrow inorder(delete\ x\ t) = del_{list}\ x\ (inorder\ t) & T_2 \\ sorted(inorder\ t) \Rightarrow lookup\ t\ x\ =\ (x\ \in\ set\_BsTreeVar\ t) & T_3 \end{cases}$$

通过在 Isabelle 中证明规约 T_1, T_2 和 T_3, 即表示了查找、插入和删除函数的功能正确性. 对于规约 T_3 的证明, 与标准二叉搜索树的证明是相同的[12], 这里我们不予讨论. 下面以规约 T_2 为例, 讨论标准二叉搜索树 *bst* 的正确性证明过程. 通过将 *t* 归纳为<*l*, (*a*, _), *r*>, 假设 *x*<*a*, 其证明过程如下:

$$\begin{aligned} &inorder(delete\ x\ t) \\ &= inorder(delete\ x\ l) @\ a\ \#\ inorder\ r \quad //delete的定义展开 \\ &= del_{list}\ x\ (inorder\ l) @\ a\ \#\ inorder\ r \quad //归纳假设 \\ &= del_{list}\ x\ (inorderl @\ a\ \#\ inorder\ r) \quad //通过构造辅助引理 \\ &= del_{list}\ x\ (inorder\ t) \quad //t的定义 \end{aligned}$$

上面的第 3 步是通过构造辅助引理得到的, 构造的引理如下:

$$\begin{cases} sorted(xs\ @\ y\ \#\ ys) = (sorted(xs\ @\ y) \wedge sorted(y\ \#\ ys)) \\ sorted(xs\ @\ [a]) \wedge x < a \Rightarrow del_{list}\ x\ (xs\ @\ a\ \#\ ys) = (del_{list}\ x\ xs)\ @\ a\ \#\ ys \end{cases}.$$

第 1 个引理重写假设 *sorted*(*inorder t*) 到 *sorted*(*inorder l* @ [*a*])∧*sorted*(*a* # *inorder r*), 从而允许第 2 个引理重写项 $del_{list}\ x$ (*inorder l* @ *a* # *inorder r*) 到 $del_{list}\ x$ (*inorder l*) @ *a* # *inorder r*).

这是标准二叉搜索树中函数 *delete* 能维持 *bst* 的一个证明过程, *insert* 亦是如此. 图 4 中给出了二叉搜索树及其变体基本不变量 *bst* 的验证引理集.

$$\text{list\_simps}\begin{cases} l_1 : sorted(xs@y\#ys) = (sorted(xs@[y]) \wedge sorted(y\#ys)) \\ l_2 : sorted(x\#xs@y\#ys) = (sorted(x\#xs) \wedge x < y \wedge sorted(xs@[y]) \wedge sorted(y\#ys)) \\ l_3 : sorted(x\#xs) \Rightarrow sorted\ \ xs \\ l_4 : sorted(xs@[y]) \Rightarrow sorted\ \ xs \end{cases}$$

$$\text{insert / delete\_def}\begin{cases} l_5 : sorted(xs@[a]) \Rightarrow ins_{list}\ x\ \ (xs@a\#ys)\ = \\ \quad (if\ \ x < a\ \ then\ \ ins_{list}\ \ x\ \ xs@a\#ys\ \ else\ \ xs@ins_{list}\ \ x(a\#ys)) \\ l_6 : sorted(xs@a\#ys) \Rightarrow del_{list}\ \ x\ \ (xs@a\#ys)\ = \\ \quad (if\ \ x < a\ \ then\ \ del_{list}\ \ x\ \ xs@a\#ys\ \ else\ \ xs@del_{list}\ \ x(a\#ys)) \end{cases}$$

$$\text{bst\_balance\_functions}\{\ l_7 : inorder(balance\_fun\ \ l\ \ a\ \ r)\ =\ inorder\ \ l@a\#inorder\ \ r$$

$$\text{bst\_split}\begin{cases} l_8 : split\_max\ \ t = (a,t') \wedge t \neq leaf \Rightarrow inorder\ \ t'@[a] = inorder\ \ t \\ l_9 : split\_min\ \ t = (a,t') \wedge t \neq leaf \Rightarrow a\#inorder\ \ t' = inorder\ t \end{cases}$$

图 4　二叉搜索树及其变体基本不变量 *bst* 的验证引理集

图 4 中的引理集可用来证明规约 T_1 和 T_2 的正确性, 其中 l_1 和 l_2 是对有序列表@和#操作进行展开和化简的辅助引理集, l_3 和 l_4 是有序列表中与其子列表的蕴含关系. l_5 和 l_6 基于 *insert*/*delete* 的定义, 表示的是 *sorted* 分别与列表中插入、删除操作之间的关系. 对于标准二叉搜索树, 引理 l_1–l_6 足以完成对 *bst* 的证明, 称为基本引理集 (*list_simps* 和 *insert/delete_def*).

对于二叉搜索树变体基本不变量 *bst* 的证明, 引理 l_1–l_6 的使用是默认的. 由于其插入和删除操作需要构造平衡函数来维持结构不变量, 并不能保证不会破坏 *bst*, 因此在对 *bst* 进行证明时, 还需给出满足 *inorder* 遍历的平衡函数引理集 l_7 (*bst_balance_functions*). 在 l_7 中, *balance_fun* 表示为了维持二叉搜索树附加性质在进行插入、删除操作时所定义的泛化平衡函数, 其参数均为 *l*, *a*, *r*.

另外, 对于删除操作, 需要用待删除节点子树的最大元素或最小元素去替换待删除节点, 因此给出了分离函数

引理集 l_8 和 l_9 (*bst_split*), 其中引理 l_8 适用于 AVL 树、AA 树等, 引理 l_9 适用于非平衡树、RBs、2-3-4 树、1-2 兄弟树等[8]. 在图 5 中给出了规约 T_2 在 Isabelle 中的证明示例, 其中①为基本引理集的应用, 对于标准二叉搜索树基本不变量 *bst* 的验证可使用①自动证明; 而对于二叉搜索树变体, 除①之外, 还需使用②和③的引理, ②表示平衡函数和分离函数的引理集, ③是其他引理集, 不属于图 4 中的验证引理集范围, 但在证明过程中会用到.

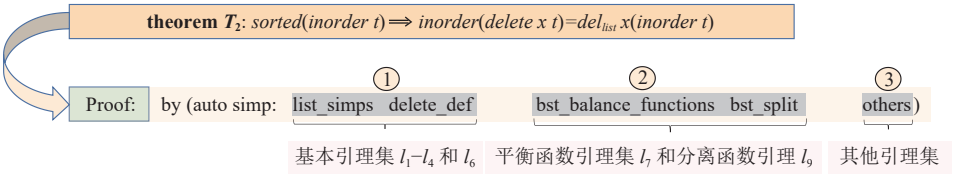


图 5　二叉搜索树删除操作维持 *bst* 在 Isabelle 中的证明

3.4　结构不变量 *inv* 的验证框架

附加性质的验证框架主要针对二叉搜索树变体, 其结构不变量 *inv* (如高度、颜色等) 在算法执行前后不会被破坏. 本节给出了一定的辅助引理集和引理选用策略, 下面是在 *insert* 和 *delete* 操作后, 维持结构不变量所需的规约:

$$\begin{cases} invar\ t \Rightarrow inv(insert\ x\ t) & T_4 \\ invar\ t \Rightarrow inv(delete\ x\ t) & T_5 \end{cases}$$

对于规约 T_4 和 T_5, 其前提都为 *invar t*, 代表 *t* 完全满足某一类二叉搜索树变体的所有不变量, 在执行插入和删除操作之后, 仍然满足某一结构不变量 *inv*, 这便是对函数 *insert* 和 *delete* 能维持某一附加性质的正确性证明. 其中, *inv* 可以实例化为高度不变量、颜色不变量等. 我们的策略是对附加性质进行模块化处理, 分别在 Isabelle 中验证它们的正确性.

在对二叉搜索树进行插入或删除操作时, 往往需要调用平衡函数去维持其结构不变量, 因此对函数 *insert* 和 *delete* 进行正确性证明时, 需要对平衡函数构造辅助引理. 这里把它归结为两种情况.

(1) 在插入或删除一个节点后, 某一结构不变量不会被破坏, 需要验证平衡函数不会破坏该结构不变量.

(2) 在插入或删除一个节点后, 某一结构不变量可能会被破坏, 需要验证平衡函数对它进行调整之后, 能够维持该结构不变量.

对于情况 (1), 下面给出了平衡函数的辅助引理集:

$$l_{10}: inv\ t \Rightarrow inv(balance\_fun\ l\ a\ r).$$

该灵感来自于 RB 插入算法的实现[8]中, 即总是将插入的新节点设置为红色, 那么 RB 的高度性质将不会被破坏, 因此在对它的高度性质进行证明时, 符合情况 (1). 在 l_{10} 中, *inv t* 表示 *t* 满足某一结构不变量 *inv*, 且被平衡函数调整后仍然满足该结构不变量.

对于情况 (2), 在插入或删除一个节点后, 某一结构不变量会被破坏, 那么在对平衡函数构造辅助引理的时候, 需要证明一种状态: 即某一结构不变量被破坏的状态 (在高阶逻辑中被形式化为弱化的谓词) 可通过平衡函数而维持该结构不变量. 下面给出情况 (2) 的平衡函数辅助引理集:

$$\begin{cases} l_{11}: inv\ t \Rightarrow inv\_weak\ t \\ l_{12}: inv\_weak\ t \Rightarrow inv(balance\_fun\ l\ a\ r) \end{cases},$$

其中, 引理集 l_{11} 表示若 *t* 满足某一结构不变量 *inv*, 则 *t* 必然也满足其弱化的结构不变量; 引理集 l_{12} 表示当树被插入或删除一个节点后, 某一结构不变量可能会被破坏, 因此用弱化的结构不变量 *inv_weak* 来尽可能表示所有被破坏情况 (如果未能表示完全, 则需要继续构造别的辅助引理, 当然这是平凡的情况), 并且通过平衡函数 *balance_fun* 之后, 这种被破坏的状态被修复, 从而仍然满足该结构不变量 *inv*.

4　LLRB 算法的函数式建模

LLRB 是 Sedgewick[5]提出的一种二叉搜索树变体, 相较于其他的 RBs 版本, 自平衡时需要处理的情形较少,

使得代码相对简洁. 本节选取了文献 [5] 中的 Top-down 2-3-4 版本, 并基于第 2 节的二叉搜索树函数式建模框架, 在 Isabelle 中对 LLRB 的定义、不变量、插入和删除操作进行建模. 本文对 LLRB 算法的建模及验证脚本可以参阅 https://github.com/Criank/LLRB_proof/tree/master.

4.1 LLRB 的定义及其不变量的函数式实现

LLRB 内部节点具有唯一的键, 每个节点具有颜色属性 (红色或者黑色), 并且满足以下性质[6].

性质 1: LLRB 节点的中序遍历会产生一个线性升序的列表, 称为基本不变量 *bst*.

性质 2: 任意节点到叶子节点的所有路径都具有相同数量的黑色节点, 称为高度不变量 *invh*.

性质 3: 根节点为黑色.

性质 4: 在所有内部节点所形成的路径中, 不能存在两个连续的红色节点.

性质 5: 对于任意的内部节点, 其右子节点和左子节点不能同时为右红左黑, 同时这也是左倾红黑树的来由, 性质 4 和 5 统称为颜色不变量 *invc*.

这些性质保证了 LLRB 的查找、插入和删除某个值的时间复杂度的最坏情况与树的高度成正比, 在理论上 LLRB 是高效的. 基于第 2 节中二叉搜索树结构的函数式定义框架, *BsTreeVar* 实例化为 *llrb*, 附加属性 *add_prop* 实例化为颜色属性 *color*, LLRB 定义的函数式实现如下.

```
//LLRB 颜色属性定义
datatype color = Red | Black
type_synonym 'a llrb = "('a * color) search_tree"
//颜色缩写的语法糖
abbreviation R where "R l a r ≡ Node l (a, Red) r"
abbreviation B where "B l a r ≡ Node l (a, Black) r"
```

上述性质 2–5 为 LLRB 不变量的非形式化描述, 在 Isabelle 中被形式化为谓词 *llrb*, 并对其附加性质进行模块化处理, 被分为高度不变量 *invh* 和颜色不变量 *invc*.

```
//LLRB 在 Isabelle 中的形式化:
definition llrb :: "'a llrb ⇒ bool" where
    "llrb t = (invc t ∧ invh t ∧ color t = Black)" //color t = Black 表示满足性质 3
//定义黑色高度 bheight
fun bheight :: "'a llrb ⇒ nat" where
    "bheight Leaf = 0" |
    "bheight (Node l (x, c) r) = (if c = Black then bheight l + 1 else bheight l)"
//定义高度不变量 invh, 满足性质 2
fun invh :: "'a llrb ⇒ bool" where
    "invh Leaf = True" |
    "invh (Node l (x, c) r) = (bheight l = bheight r ∧ invh l ∧ invh r)"
//定义颜色不变量 invc, 满足性质 4 和性质 5:
fun invc :: "'a llrb ⇒ bool" where //fun 定义的优势在于函数的终止性可以自动证明
    "invc Leaf = True" |
    "invc (Node l (a, c) r) = ((c = Red ⟶ color l = Black ∧ color r = Black) ∧
                               (c = Black ⟶ color r = Red ⟶ color l = Red) ∧ invc l ∧ invc r)"
//下面是有关颜色的两个辅助函数:
fun paint :: "color ⇒ 'a llrb ⇒ 'a llrb" where //设置节点颜色
```

```
  "paint c Leaf = Leaf" | "paint c (Node l (a, _) r) = Node l (a, c) r"
fun color :: "'a llrb ⇒ color" where //获取节点颜色, 且定义叶子节点为黑色
  "color Leaf = Black" | "color (Node _ (_, c) _) = c"
```

将不变量分解为 *invc* 和 *invh* 提高了模块化, 通常可以分别证明插入和删除操作在结构不变量 *invc* 和 *invh* 上的正确性, 有利于后期的验证工作.

4.2 LLRB 插入和删除算法的函数式实现

LLRB 算法实现的难点在于插入和删除操作, 插入或删除某一节点后必须使整棵树仍然满足 LLRB 的性质. 由于其查找算法与标准二叉搜索树是完全相同的, 因此本文不予讨论. 本节主要阐述 LLRB 的插入和删除算法是如何基于第 2.3 节的高阶泛化函数实例化生成的, 以及平衡函数是如何维持不变量的.

(1) 插入

对于插入操作的函数式实现, 首先将区域 *BsTreeVar_op* 进行实例化, 其中 *BsTreeVar_op* 实例化为 *llrb_op*, *BsTreeVar* 实例化为 *llrb*.

```
locale llrb_op =
    and pre_invl :: "'a llrb ⇒ 'a llrb ⇒ 'a ⇒'a llrb ⇒ 'a llrb"
    and pre_invr :: "'a llrb ⇒ 'a ⇒ 'a llrb ⇒'a llrb ⇒ 'a llrb"
    and pre_invsplit :: "'a llrb ⇒ 'a ⇒ 'a llrb ⇒ 'a llrb"
```

LLRB 的插入操作按照二分法, 从根节点开始往下遍历直至搜索到合适位置, 替换叶子节点实现元素的插入. 在这个过程中可能会破坏高度不变量 *invh*, 为了避免这种情况, 采用的策略是将插入的新节点设置为红色并与它的父节点连接来避免破坏 *invh*, 从而使得在自平衡时只需考虑颜色不变量 *invc*. 由此基于本文第 2.3 节的二叉搜索树类结构插入操作的泛化函数, 可构造 LLRB 的插入操作的泛化函数 *ins'*.

```
fun ins' :: " 'a ⇒ 'a llrb ⇒ 'a llrb" where
//新节点插入空树中
  "ins' x Leaf = R Leaf x Leaf" |
//新节点插入黑色的非空子树中
  "ins' x (B l a r) = (case cmp x a of LT ⇒ pre_invl (ins' x l) l a r |
                                       GT ⇒ pre_invr l a r (ins' x r) |
                                       EQ ⇒ B l a r)" |
//新节点插入红色的非空子树中
  "ins' x (R l a r) = (case cmp x a of LT ⇒ R (ins' x l) a r |
                                       GT ⇒ R l a (ins' x r) |
                                       EQ ⇒ R l a r)"
```

当新节点插入红色非空子树中, 我们没有进行平衡操作且维持了子树节点的红色属性, 因为平衡会改变该子树的黑色高度, 从而导致整个树的 *invh* 性质被破坏. 而当新节点插入黑色非空子树中, 需调用接口 *pre_invl* 和 *pre_invr* 对其进行平衡, 因为此时通过平衡操作能够在不破坏 *invh* 性质的前提下维持新树的 *invc* 性质. 具体而言, 插入操作的平衡需要考虑以下两种情况.

1) 当插入红色节点的左侧, 出现两个连续的红色节点, 此时性质 4 被破坏, 可使用 *baliL'*进行翻转调整, 如图 6(c); 当插入到红色节点的右侧时, 同理使用 *baliR'*进行翻转调整, 如图 6(b).

2) 当插入到黑色节点的右侧或者插入完成后进行自平衡调整时, 可能会出现孤立的红色右子节点, 此时性质 5 被破坏, 可使用 *rightredB* 进行翻转调整, 如图 6(a).

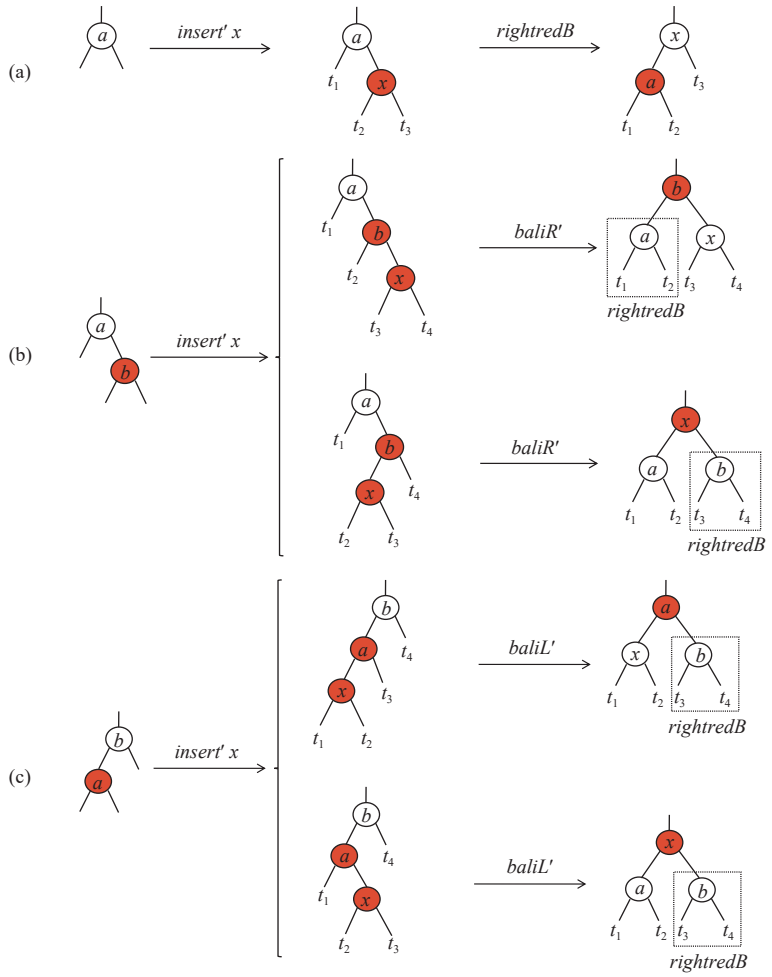


图 6 LLRB 插入操作的自平衡情况

如图 6 所示, 给出了插入操作的自平衡调整过程, 其中 *insert' x* 表示插入新节点后的状态, t_i 为任意子节点.

基于上述 LLRB 的插入操作的泛化函数 *ins'*和图 6 所示的 3 种自平衡情况, 接口 *pre_invl* 和 *pre_invr* 分别被实例化为自平衡调整函数 *baliL'*和 *baliR'*, 同时给出了内部平衡调整函数 *rightredB* 的定义.

---

locale *llrb_insert*

//解决红色右倾

fun *rightredB* :: "*'a llrb* ⇒ *'a* ⇒ *'a llrb* ⇒ *'a llrb* " **where**

"*rightredB* t_1 *a* (*R* t_2 *b* t_3) = (*B* (*R* t_1 *a* t_2) *b* t_3)" |

"*rightredB Leaf a* (*R* t_1 *b* t_2) = *B* (*R Leaf a* t_1) *b* t_2" |

"*rightredB* t_1 *a* t_2 = *B* t_1 *a* t_2"

fun *baliL'*:: "*'a llrb* ⇒ *'a llrb* ⇒ *'a* ⇒ *'a llrb* ⇒ *'a llrb*" **where**

"*baliL'* (*R* t_1 *a* (*R* t_2 *b* t_3)) _ c t_4 = *R* (*B* t_1 *a* t_2) *b* (*rightredB* t_3 *c* t_4)" |

"*baliL'* (*R* (*R* t_1 *a* t_2) *b* t_3) _ c t_4 = *R* (*B* t_1 *a* t_2) *b* (*rightredB* t_3 *c* t_4)" |

"*baliL'* t_1 _ *a* t_2 = *rightredB* t_1 *a* t_2"

fun *baliR'* :: "*'a llrb* ⇒ *'a* ⇒ *'a llrb* ⇒ *'a llrb* ⇒ *'a llrb*" **where**

"*baliR' t₁ a _ (R t₂ b (R t₃ c t₄)) = R (rightredB t₁ a t₂) b (B t₃ c t₄)*" |
"*baliR' t₁ a _ (R (R t₂ b t₃) c t₄) = R (B t₁ a t₂) b (rightredB t₃ c t₄)*" |
"*baliR' t₁ a _ t₂ = rightredB t₁ a t₂*"
//区域 *llrb_op* 中的接口 *pre_invl* 和 *pre_invr* 分别实例化为 *baliL'*和 *baliR'*
interpretation *llrb_op baliL' baliR'*

函数 *baliR'*和 *baliL'*可处理两个连续红色的情况, 并通过调用函数 *rightredB* 保证新树在满足基本红黑树性质的基础上同时解决红色单独右倾的情况. 同时, *baliL'*和 *baliR'*是单边自平衡的, 因为在插入元素之前, 树本身是满足 LLRB 的性质的. 比如当使用 *baliL'*对左子树进行调整时, 右子树要么本身就满足 LLRB 性质, 要么是调整操作后得到的满足 LLRB 性质的新子树. 连续红色只能发生在单边左子树上, 如图 6(c) 所示, 因此对右子树无需进行多余的操作, 即 *baliL'*只单边自平衡. *baliR'*也是同理. 因此, 通过 *baliL'*和 *baliR'*的单边自平衡调整, 可确保树的平衡性和性能.

我们基于 *locale* 给出了 LLRB 的插入实现, 在后续的正确性验证中, 为了方便验证并未直接在 *locale* 中进行, 而是在区域外定义了与之等价的 *ins* 函数 (见验证脚本 proof_insert.thy). 后续若能验证 *ins* 的正确性, 即表示区域中的 *ins'*是正确的, 同时这里给出函数 *ins'*和 *ins* 的等价证明, 如图 7 所示.

```
lemma locins_eq_ins:"ins' x t = ins x t"
  apply(induct t)
  apply simp
  apply(case_tac "x2")
  apply(case_tac "b")
  apply (simp only: ins'.simps ins.simps)
  by (metis ins'.simps(2) ins.simps(2) baliL'_eq_baliL baliR'_eq_baliR)
```

图 7　函数 *ins'*和 *ins* 的等价证明

当调整至根部时, 根节点可能会出现红色的情况, 为了不破坏 LLRB 的性质 3, 函数 *ins* 还要进行染黑 (*paint Black*) 操作.

definition *insert* :: "'*a* ⇒ '*a llrb* ⇒ '*a llrb*" **where**
"*insert x t = paint Black* (*ins x t*)"

至此, 已完成对 LLRB 插入操作函数式实现.

(2) 删除

LLRB 的删除操作整体上与标准二叉搜索树的删除方式大致相同: 需考虑空树和非空树的情况, 对于空树, 直接返回空即可; 对于非空树, 在寻找删除位置的过程中, 会进行自顶向下的二分查找, 找到指定节点后, 执行删除操作, 再逐层向上对不满足性质的节点或者子树结构进行调整. 整体的删除过程如图 8 所示.

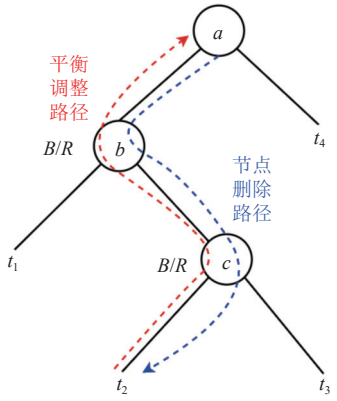


图 8　LLRB 的自底向上的自平衡调整过程

LLRB 的自平衡调整过程为: 当搜索到删除节点之后, 则从其右子树中把最小的元素替换到待删除节点的位置, 再逐层向上对不满足性质的节点或者子树结构进行自平衡调整. 遍历待删除节点是从根节点开始, 自顶向下地进行查找的, 找到待删除节点执行 *pre_invsplit* 函数, 然后自底向上执行 *pre_invl* 和 *pre_invr* 调整函数. 由于在对 LLRB 进行删除操作时, 只有从根到待删除节点路径上节点的 *bheight* 才会变化, 而其他节点的 *bheight* 是不变的, 这决定了自底向上自平衡调整更为有效.

基于 *cmp* 函数, 并结合递归自底向上调整, 删除操作可分为 3 种不同的情况. 基于本文第 2.3 节的二叉搜索树类结构的删除操作的泛化函数, 可构造 LLRB 的删除操作的泛化函数 *del'*.

```
fun del' :: "'a ⇒ 'a llrb ⇒ 'a llrb" where
  "del' x Leaf = Leaf" |
  "del' x (Node l (a, _) r) = (case cmp x a of LT ⇒ pre_invl (del' x l) l a r |
                                               GT ⇒ pre_invr l a r (del' x r) |
                                               EQ ⇒ pre_invsplit l a r)"
```

接着对区域 *llrb_op* 进行解释, *del'*中的接口 *pre_invl*、*pre_invr* 和 *pre_invsplit* 分别被实例化为 *pre_invl_llrb*、*pre_invr_llrb* 和 *pre_invsplit_llrb*.

```
locale llrb_delete
    definition pre_invl_llrb:: "'a llrb ⇒ 'a llrb ⇒'a ⇒ 'a llrb⇒ 'a llrb" where
      "pre_invl_llrb l' l a r = (if l ≠ Leaf ∧ color l = Black then baldL l' a r
                                 else rightredR l' a r) "
    definition pre_invr_llrb:: "'a llrb ⇒ 'a ⇒'a llrb ⇒ 'a llrb⇒ 'a llrb" where
      "pre_invr_llrb l a r r' = (if r ≠ Leaf ∧ color r = Black then baldR l a r'
                                 else R l a r' )"
    definition pre_invsplit_llrb:: "'a llrb ⇒ 'a ⇒ 'a llrb⇒ 'a llrb" where
      "pre_invsplit_llrb l a r = (if r = Leaf then l else let (a' , r') = split_min r in
                                    if color r = Black then baldR l a' r' else rightredR l a' r' )"
interpretation llrb_op pre_invl_llrb pre_invr_llrb pre_invsplit_llrb
```

① 函数 *pre_invl_llrb* 处理左子树"失衡"的情况, 它接受 4 个参数, 参数 *l'*用来记录下一层中已经过 *del'*调整过后返回的子树, 参数 *l*、*a* 和 *r* 分别对应调整前的左子树、根节点和右子树. 当 *l* 的根节点的颜色: 为黑, 说明执行删除操作后得到的 *l'*的 *bheight* 比 *r* 的 *bheight* 小 1, 此时需同时考虑 *invh* 和 *invc*, 调用 *baldL* 进行处理; 为红, 说明执行删除操作后得到的 *l'*的 *bheight* 与 *r* 的 *bheight* 相等, 此时只需考虑 *invc* 即可, 调用 *rightredR* 进行处理. 引理 *invh_del*:⟦*invh t*; *invc t*⟧⟹*invh* (*del x t*)∧(*color t*=*Red* ⟶ *bheight* (*del x t*)=*bheight t*)∧(*color t*=*Black*⟶*bheight* (*del x t*)=*bheight t*–1) 已证明上述结论, 详见验证脚本 proof_delete.thy; ② 同理, *pre_invr_llrb* 处理右子树"失衡"的情况; ③ *pre_invsplit_llrb* 的思想也类似, 不过它多一步找右子树中最小元素的操作, 调用 *split_min* 进行处理.

下面分别介绍 *split_min*、*baldL* 和 *baldR* 的具体实现思路及其功能.

■ *split_min*

split_min 的作用是返回树的最小值及其分离最小值后的自平衡树. 下面是其函数式定义.

```
fun split_min :: "'a llrb ⇒ 'a * 'a llrb" where
  "split_min (Node l (a, _) r) = (if l = Leaf then (a, r) else let (x, l') = split_min l in
                                    (x, if color l = Black then (baldL l' a r) else (rightredR l' a r)))"
```

其具体思路为: 因为 LLRB 本身也是二叉搜索树, 所以其最小值节点是在左子树上. 当 *split_min* 找到该节点

时, 返回最小值 *x*, 并将其替代待删除节点上的值, 然后删除最小值节点. 根据最小值节点的颜色, 执行以下操作并返回调整后得到的树: ① 若为黑色, 说明删除的节点是黑色, 那么删除节点得到的左子树 *l'*, 其 *bheight* 比右子树 *r* 的 *bheight* 小 1. 对于当前以 *a* 为根节点的树来说, 其左、右子树不满足 *invh*, 需要调用 *baldL* 进行调整 (*baldL* 需同时考虑 *invh* 和 *invc*); ② 若为红色, 说明删除了左边的红色节点, 则可能不满足 *invc*, 调用函数 *rightredR* 进行调整.

■ *baldL* 和 *baldR*

针对 LLRB 删除操作不同的自平衡情况, *baldR* 和 *baldL* 给出了不同的匹配方案进行自平衡操作, 如图 9 所示.

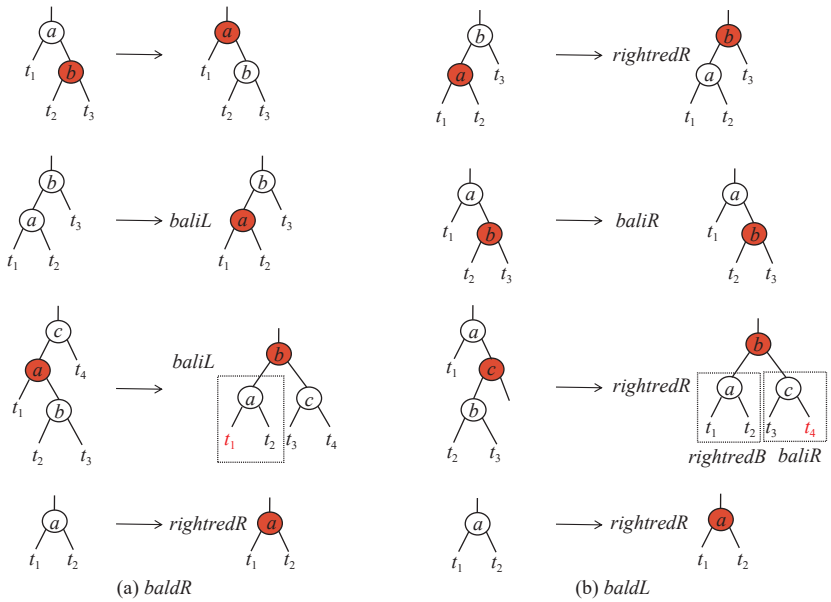


图 9 LLRB 删除操作的自平衡情况

下面给出了这两个函数的具体定义.

```
//左平衡函数 baldL 的定义
fun baldL :: "'a llrb ⇒ 'a ⇒ 'a llrb ⇒ 'a llrb" where
  "baldL (R t1 a t2) b t3 = rightredR (B t1 a t2) b t3" |
  "baldL t1 a (B t2 b t3) = baliR t1 a (R t2 b t3)" |
  "baldL t1 a (R (B t2 b t3) c t4) = rightredR (rightredB t1 a t2) b (baliR t3 c (paint Red t4))" |
  "baldL t1 a t2 = rightredR t1 a t2"
//右平衡函数 baldR 的定义
fun baldR :: "'a llrb ⇒ 'a ⇒ 'a llrb ⇒ 'a llrb" where
  "baldR t1 a (R t2 b t3) = R t1 a (B t2 b t3)" |
  "baldR (B t1 a t2) b t3 = baliL (R t1 a t2) b t3" |
  "baldR (R t1 a (B t2 b t3)) c t4 = R (baliL (paint Red t1) a t2) b (B t3 c t4)" |
  "baldR t1 a t2 = rightredR t1 a t2"
```

baldR 的具体思路如下, 4 种情况与代码行顺序对应.

1) 若匹配的右子树的根节点为红, 则直接把其染黑即可.

2) 若匹配的右子树的根节点为黑, 且左子树的根节点为黑, 则把左子树的根节点染红, 此时可能会出现连续红节点的情况, 需要调用 *baliL* 来调整.

3) 若匹配的右子树的根节点为黑, 且左子树的根节点为红, 则把左子树的左孩子染红, 然后进行翻转调整, 如图 9(a), 翻转后的左子树可能会出现连续红节点的情况, 需要调用 *baliL* 来调整.

4) 其他情况, 可能会出现左黑右红, 使用 *rightredR* 进行调整.

上述 4 种情况暂未考虑当前根节点的颜色. 若当前的根节点颜色为黑, 函数 *baldR* 调整后得到的树的整体 *bheight* 减 1 (具体可表现为将根节点染红等操作), 相较于执行删除操作前的树的 *bheight* 要小. 若为红色, 则调整后得到的树的 *bheight* 不变.

平衡函数 *baldL* 则处理左子树 *bheight* 减小的情况, 同理, 可将左子树染黑或右子树染红来维持 *invh*, 但将左子树染黑可能会出现左黑右红的情况, 故此时要调用 *rightredR* 来处理, 如图 9(b).

同插入一样, 我们在区域外定义了与 *del'*等价的 *del* 函数 (见验证脚本 proof_delete.thy), 并给出了它们的等价证明, 如图 10 所示.

```
lemma locdel_eq_del:"del' x t = del x t"
  apply(induct t)
  apply simp
  apply(case_tac "x2")
  apply (simp only: del'.simps del.simps)
  by (simp add: pre_invl_llrb_def pre_invr_llrb_def pre_invsplit_llrb_def)
```

图 10 函数 *del'*和 *del* 的等价证明

当调整至最顶部时, 根节点可能会出现红色的情况, 为了不破坏 LLRB 的性质 3, 函数 *del* 还要进行染黑 (*paint Black*) 操作.

---

definition *delete* :: "*'a ⇒ 'a llrb ⇒ 'a llrb*" **where**

"delete x t = paint Black (del x t)"

至此已完成对 LLRB 删除操作函数式实现.

5 LLRB 算法的正确性验证

LLRB 的插入和删除函数的验证分为终止性证明和功能正确性验证. 对于函数的终止性证明, 第 4 节给出的函数式实现均是通过 *fun* 和 *definition* 来定义的, 终止性在 Isabelle 中已被自动检查并验证; 对于函数的功能正确性验证, 通过定义二叉搜索树类结构的不变量, 即基本不变量 *bst* 及其结构不变量 (*invc* 和 *invh*), 它们在程序执行前后不会发生改变, 从而证明函数 *insert* 和 *delete* 的功能正确性. 本节阐述了如何使用验证框架构建引理, 以及引理与不变量之间的关系.

5.1 基本不变量 *bst* 的正确性验证

基本不变量 *bst* 表示二叉搜索树节点中序遍历的结果按照线性升序的方式排序. 基于第 3.3 节的规约 T_1, 可构造 *insert* 操作维持 *bst* 的定理 1.

定理 1. theorem *bst_insert*: "*sorted*(*inorder t*) ⟹ *inorder*(*insert x t*) = *ins_list x* (*inorder t*)".

定理 1 表示 LLRB 的 *insert* 执行前后中序遍历的列表是保持线性升序的. 图 11 给出了证明定理 1 相关的辅助引理以及它们之间的依赖关系.

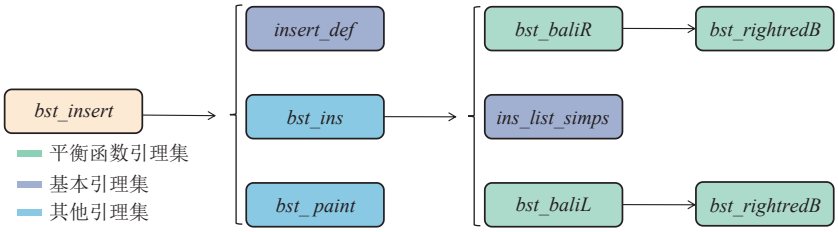


图 11 *insert* 操作维持 *bst* 的验证引理依赖关系

由图 11 可知, 定理 1 的证明主要由引理 1 bst_ins 来完成, 其定义如下.

引理 1. lemma *bst_ins*: "*sorted*(*inorder t*) ⟹ *inorder*(*ins x t*) = *ins_list x* (*inorder t*)".

引理 1 表示 *ins* 函数执行前后中序遍历的列表是保持线性升序的. 由于 *ins* 函数的实现调用了平衡函数 *baliL* 和 *baliR*, 根据辅助引理集 l_7, *balance_fun* 可实例化为这两个函数, 从而构造辅助引理 *bst_baliR* 和 *bst_baliL*. 它们的构造是类似的, 以引理 2 *bst_baliR* 为例.

引理 2. lemma *bst_baliR*: "*inorder*(*baliR l a r*) = *inorder l* @ *a* # *inorder r*".

引理 2 表示执行前后树的中序遍历结果是一致的, 从而维持 *bst* 性质. 对于引理 2 的证明, *baliR* 函数在解决两个连续红色节点的同时, 其内部结构依赖于 *rightredB* 函数, 需证明辅助引理 3 *bst_rightredB*.

引理 3. lemma *bst_rightredB*: "*inorder*(*rightredB l a r*) = *inorder l* @ *a* # *inorder r*".

引理 3 表示 *rightredB* 函数在处理红色单独右倾的情况时, 维持 *bst* 性质.

上述引理在 Isabelle 中被机械证明之后, 定理 1 可被验证, 定理 1 在 Isabelle 中的证明脚本如图 12 所示.

```
lemma bst_ins:
  "sorted(inorder t) ⟹ inorder(ins x t) = ins_list x (inorder t)"
  by(induction x t rule: ins.induct)
  (auto simp: ins_list_simps bst_baliL bst_baliR)

theorem bst_insert:
  "sorted(inorder t) ⟹ inorder(insert x t) = ins_list x (inorder t)"
  by(auto simp: insert_def bst_ins bst_paint)
```

图 12 *insert* 操作维持 *bst* 的证明

同理, 我们可在 Isabelle 中证明 *delete* 操作维持 *bst* 性质的正确性, 略.

5.2 结构不变量 *inv* 的正确性验证

(1) 高度不变量 *invh* 的正确性验证

LLRB 的高度不变量 *invh* 表示从根到叶子节点的所有路径都具有相同数量的黑色节点. 基于 3.4 节的规约 T_4, 其中 *invar t* 实例化为 *llrb t*, *inv* 实例化为 *invh*, 可构造 *insert* 操作维持 *invh* 的定理 2.

定理 2. theorem *invh_insert*: "*llrb t* ⟹ *invh*(*insert x t*)".

定理 2 表示 LLRB 执行 *insert* 函数后仍满足高度不变量 *invh*. 图 13 给出了证明定理 2 相关的辅助引理以及它们之间的依赖关系.

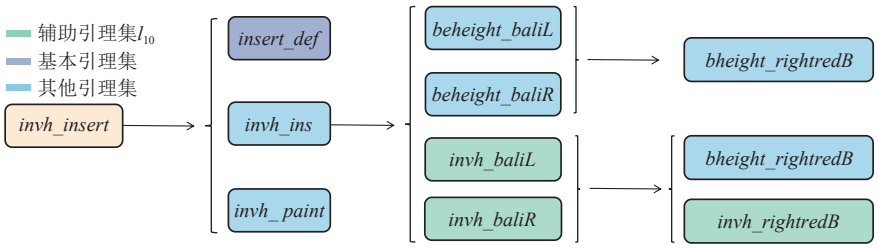


图 13 *insert* 操作维持 *invh* 的验证引理依赖关系

由图 13 可知, 定理 2 的证明主要由引理 4 invh_ins 来完成, 其定义如下.

引理 4. lemma *invh_ins*: "*invh t* ⟹ *invh*(*ins x t*) ∧ *bheight*(*ins x t*) = *bheight t*".

引理 4 表示 *ins* 函数执行前后不会破坏高度不变量 *invh*. 同时进行 *ins* 操作后, LLRB 的黑色高度不变.

对于引理 4 的证明, 由于 *ins* 函数的实现通过调用 *baliL* 和 *baliR* 函数. 基于 3.4 节的辅助引理集 l_{10}, *balance_fun* 被分别实例化为这两个函数, *inv t* 被实例化为 *invh t*, 并根据第 4.1 节中 *invh t* 的定义可构造辅助引理 *invh_baliL* 和 *invh_baliR*, 另外, 由于结论的增强, 需证明 *ins* 函数不会改变黑色高度, 由此构造辅助引理 *bheight_*

baliL、*bheight_baliR* 和 *bheight_rightredB*. 以引理 5 *invh_baliL* 和引理 6 *bheight_baliL* 为例:

引理 5. lemma *invh_baliL*: "⟦*invh l*; *invh r*; *bheight l* = *bheight r*⟧ ⟹ *invh*(*baliL l a r*)".

引理 6. lemma *bheight_baliL*: "*bheight l* = *bheight r* ⟹ *bheight* (*baliL l a r*) = *Suc*(*bheight l*)".

引理 5 表示 *baliL* 函数执行前后不会破坏高度不变量 *invh*. 引理 6 表示对于任意 LLRB, 若其左右子树的黑色高度相等, 那么经过函数 *baliL* 处理之后, 该树的黑色高度比子树多 1.

上述引理在 Isabelle 中被机械证明之后, 定理 2 可被验证, 定理 2 在 Isabelle 中的证明脚本如图 14 所示.

```
lemma invh_ins: "invh t ⟹ invh (ins x t) ∧ bheight (ins x t) = bheight t"
  by(induct x t rule: ins.induct)
  (auto simp: invh_baliL invh_baliR bheight_baliL bheight_baliR)

theorem invh_insert: "llrb t ⟹ invh (insert x t)"
  by (auto simp: insert_def invh_ins invh_paint llrb_def)
```

图 14 *insert* 操作维持 *invh* 的证明

基于第 3.4 节的规约 T_5, 可构造 *delete* 操作维持 *invh* 性质的定理 3.

定理 3. theorem *invh_delete*: "*llrb t* ⟹ *invh*(*delete x t*)".

定理 3 表示 LLRB 执行 *delete* 函数后仍满足高度不变量 *invh*, 定理 3 与定理 2 的验证过程类似, 略.

(2) 颜色不变量 *invc* 的正确性验证

LLRB 的颜色不变量 *invc* 表示在所有内部节点所形成的路径中, 不能存在两个连续的红色节点, 且对于任意的内部节点, 黑色节点的子节点不能同时为右红左黑. 基于第 3.4 节的规约 T_4 和 T_5, 可构造 *insert* 和 *delete* 操作维持 *invc* 的定理 4 和 5.

定理 4. theorem *invc_insert*: "*llrb t* ⟹ *invc*(*insert x t*)".

定理 4 表示 LLRB 执行 *insert* 函数后仍满足 *invc*. 由于 LLRB 的颜色结构是左倾的非对称结构, 平衡后的 LLRB 不能有两个连续的红色节点以及不存在红色节点右倾的情况. 为了证明定理 4 的正确性, 我们将 LLRB 的颜色不变量 *invc* 进行弱化, 即允许存在两个连续的红色节点或红色节点右倾的情况, 从而定义了一系列弱化的颜色不变量 *invc*2、*invc*3、*invc_red*, 用来表示被破坏的状态.

```
//invc2 表示两个连续红色节点
abbreviation invc2 :: "'a llrb ⇒ bool" where
  "invc2 t ≡ invc(paint Black t)"
//LLRB 是 RB 的特殊版本, invc3 表示 RB 的颜色不变量
fun invc3 :: "'a llrb ⇒ bool" where
  "invc3 Leaf = True" |
  "invc3 (Node l (a, c) r) = ((c = Red ⟶ color l = Black ∧ color r = Black) ∧ invc l ∧ invc r)"
//invc_red 表示红色节点右倾
fun invc_red :: "'a llrb ⇒ bool" where
  "invc_red Leaf = True" |
  "invc_red (Node l (a, c) r) = (invc4 (Node l (a, c) r) ∧ invc l ∧ invc r)"
```

根据辅助引理集 l_{11}, *inv t* 实例化为 *invc t*, *inv_weak t* 实例化为上述一系列的弱化颜色不变量, 包括 *invc*2、*invc*3 和 *invc_red*, 从而得到引理 7–引理 9.

引理 7. lemma *invc2I*: "*invc t* ⟹ *invc*2 *t*".

引理 8. lemma *invc3I*: "*invc t* ⟹ *invc*3 *t*".

引理 9. lemma *invc_redI*: "*invc t* ⟹ *invc_red t*".

引理 7–9 表示若 t 满足颜色不变量 *invc*, 则必然也满足弱化的颜色不变量 *invc_weak*. 基于上述引理, 图 15 给出了证明定理 4 相应的辅助引理以及它们之间的依赖关系.

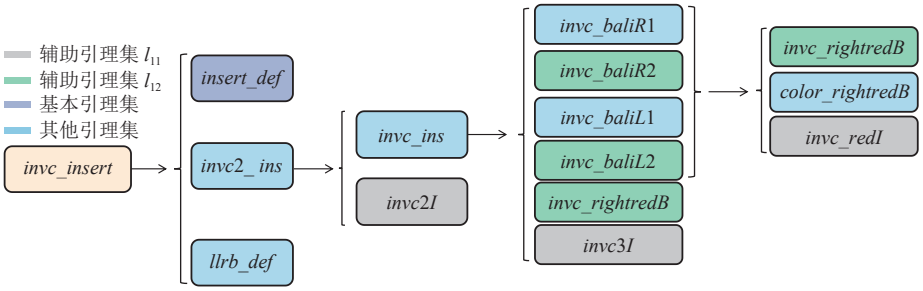


图 15 *insert* 操作维持 *invc* 的验证引理依赖关系

由图 15 可知, 定理 4 的证明主要是由引理 10 *invc2_ins* 来完成, 其定义如下.

引理 10. lemma *invc2_ins*: "*invc t* ∧ *invh t* ∧ *color t* = *Black* ⟹ *invc2* (*ins x t*)".

引理 10 表示 LLRB 进行 *ins* 操作后, 满足弱化的颜色不变量 *invc2*, 这是因为 *ins* 函数最后返回的根节点可能是红色的, 从而出现两个连续的红色节点. 对于引理 10 的证明, 可对根节点的颜色情况进行划分, 还需证明引理 11 *invc_ins*.

引理 11. lemma *invc_ins*: "*invc t* ⟶ *invc_red* (*ins x t*) ∧ (*color t* = *Black* ⟶ *invc* (*ins x t*))".

引理 11 表示 LLRB 进行 *ins* 操作的两种情况: 若树的根节点为黑色, 则进行 *ins* 操作后仍能满足 *invc*; 若树的根节点为红色, 则进行 *ins* 操作后只能满足弱化的颜色不变量 *invc_red*, 这是因为 *ins* 函数对于根节点为红色的情况暂不进行平衡操作, 比如插入一个新节点的树只有一个红色的根节点, 此时完成 *ins* 操作后会出现红色节点右倾的情况.

对于引理 11 的证明, 根据辅助引理集 l_{12}, *inv_weak t* 被分别实例化为条件 *invc l* ∧ *invc_red r*、*invc r* ∧ *invc_red l* 和 *invc l* ∧ *invc r*, 代表颜色不变量 *invc* 被破坏或者被弱化的 3 种情况, *balance_fun* 分别被实例化为 *ins* 函数内部调用的 *baliL*、*baliR* 和 *rightredB* 函数. 为判断被弱化的颜色不变量能否被 *balance_fun* 修复, 构造辅助引理 *invc_baliR2*、*invc_baliL2* 和 *invc_rightredB*, 以引理 12 *invc_baliR2* 为例:

引理 12. lemma *invc_baliR2*: "⟦*invc l*; *invc_red r*⟧ ⟹ *invc* (*baliR l a r*)".

引理 12 表示右子树弱化的颜色不变量 *invc_red*, 能被平衡函数 *baliR* 修复.

另外, 由于 LLRB 结构的特殊性, 我们还构造了一些在 *invc* 证明过程中需要的其他引理 (见图 15). 它们是基于子目标的提示来构造的. 上述引理在 Isabelle 中被机械证明之后, 定理 4 可被验证, 定理 4 在 Isabelle 中的证明脚本如图 16 所示.

```
lemma invc_ins: "invc t ⟶ invc_red (ins x t) ∧ (color t = Black ⟶ invc (ins x t))"
  apply(induct x t rule: ins.induct)
  by(auto simp: invc_baliR1 invc_baliR2 invc3I invc_baliL1 invc_baliL2 invc_rightredB)

lemma invc2_ins:"invc t ∧ invh t ∧ color t = Black ⟹ invc2 (ins x t)"
  by (simp add: invc2I invc_ins)

theorem invc_insert: "llrb t ⟹ invc (insert x t)"
  by(simp add: llrb_def insert_def invc2_ins)
```

图 16 *insert* 操作维持 *invc* 的证明

定理 5. theorem *invc_delete*: "*llrb t* ⟹ *invc* (*delete x t*)".

定理 5 表示 LLRB 执行 *delete* 函数后仍满足 *invc*. 图 17 给出了证明定理 5 相关的辅助引理以及它们之间的依赖关系.

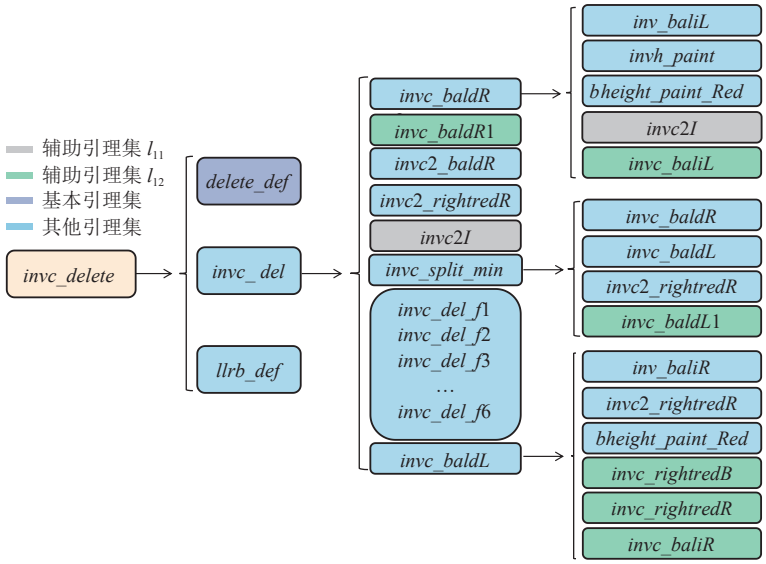


图 17 *delete* 操作维持 *invc* 的验证引理依赖关系

由图 17 可知, 定理 5 的证明主要是由引理 *invc_del* 来完成, 基于 *del* 函数和颜色属性的关系, 定义如下.

引理 13. lemma *invc_del*: "⟦*invh t*; *invc t*⟧ ⟹ (*color t* = *Red* ⟶ *invc* (*del x t*)) ∧ (*color t* = *Black* ⟶ *invc2* (*del x t*))".

引理 13 表示 *del* 函数不会破坏颜色不变量 *invc*, 在结论中我们给出了两种情况: 若树的根节点为红色, 则进行 *del* 操作后其颜色不变量不被破坏; 若树的根节点为黑色, 则进行 *del* 操作后只能满足弱化的颜色不变量 *invc2*, 这是因为在自平衡过程中会把树的根节点暂时染成红色.

对于引理 13 的证明, *del* 函数调用 *baldL*、*baldR* 和 *split_min* 函数, 基于 *invc* 和 *invc2* 与它们之间的关系, 可构造引理 *invc_baldR*、*invc_baldL* 和 *invc_split_min*, 引理 *invc_baldR* 和 *invc_baldL* 的构造是类似的, 下面我们给出了引理 14 *invc_baldR* 和引理 15 *invc_split_min* 的具体定义.

引理 14. lemma *invc_baldR*: "⟦*invh l*; *invh r*; *bheight l* = *bheight r* + 1; *invc l*; *invc2 r*⟧ ⟹ *invc2* (*baldR l a r*) ∧ (*color l* = *Black* ⟶ *invc* (*baldR l a r*))".

引理 15. lemma *invc_split_min*: "⟦*split_min t* = (*x*, *t'*); *t* ≠ *Leaf*; *invh t*; *invc t*⟧ ⟹ (*color t* = *Red* ⟶ *invc t'*) ∧ (*color t* = *Black* ⟶ *invc2 t'*)".

引理 14 和 15 的结论都划分了两种情况, 与引理 13 类似, 使用弱化的颜色不变量表示一些特殊的情形, 它们在 Isabelle 中归纳证明时会被考虑, 而实际在的定义中是不允许的. 比如我们定义的 LLRB 树的根节点总是黑色的, 但在 Isabelle 定理证明器中根为红色的情况也需要被归纳证明, 这是需要分情况的原因.

除了上述引理 13–15 等关键引理外, 在定理 5 的证明过程中也构造了一些其他平凡引理 (如 *invc_del_f*1–*invc_del_f*6), 限于篇幅这里不予讨论. 上述引理及其他相关辅助引理在 Isabelle 中被机械证明之后, 定理 5 可被验证, 定理 5 在 Isabelle 中的证明脚本如图 18 所示.

```
lemma invc_del:
"⟦ invh t; invc t ⟧ ⟹ (color t = Red ⟶ invc (del x t)) ∧ (color t = Black ⟶  invc2 (del x t))"
apply(induction x t rule: del.induct)
apply(auto simp: invc_baldR invc2_baldR invc_baldR1 invc_baldL
                 invc2_baldL invc_baldL1 invc2I invc2_rightredR
          dest!: invc_split_min
           dest: neq_LeafD
          split!: prod.splits if_splits)
   by(auto simp: invc_del_f1 invc_del_f2 invc_del_f3 invc_del_f4 invc_del_f5 invc_del_f6)

theorem invc_delete: "llrb t ⟹ invc (delete x t)"
   by(simp add: delete_def invc_del llrb_def)
```

图 18 *delete* 操作维持 *invc* 的证明

至此, 已完成对 LLRB 插入和删除操作维护结构不变量 *invc t*、*invh t* 的模块化证明, 且在建模过程中的染黑操作保证了 *color t* = *Black*, 根据 LLRB 所需满足结构不变量性质: "*llrb t* = (*invc t* ∧ *invh t* ∧ *color t* = *Black*)", 可以证明 *insert* 和 *delete* 操作能维持 LLRB 的所有结构不变量, 如图 19 所示.

```
theorem llrb_insert: "llrb t ⟹ llrb (insert x t)"
  by(metis invc_insert invh_insert llrb_def color_paint_Black insert_def)

theorem llrb_delete: "llrb t ⟹ llrb (delete x t)"
  by(metis invc_delete invh_delete color_paint_Black delete_def llrb_def)
```

图 19 *insert* 和 *delete* 操作维持 LLRB 结构不变量的整体证明

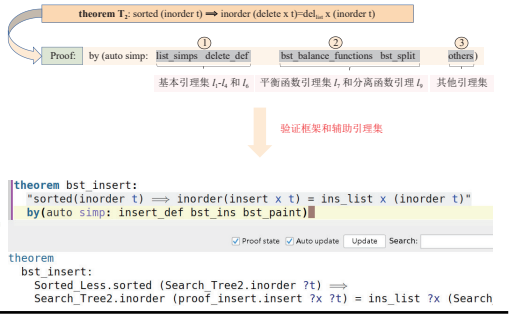
6 实验比较

文献 [6] 利用 Dafny 平台验证了 LLRB 插入和删除操作, 证明了相应函数的终止性和正确性. 本节将从验证过程、验证脚本的模块化和代码的可执行性 3 个方面与文献 [6] 进行比较.

(1) 验证过程

对于验证过程, 本文针对二叉搜索树类算法插入和删除算法的功能正确性, 提出了基于不变量的 Isabelle 验证框架, 并给出了相应的引理集, 从而克服引理选用的盲目性. 以验证 LLRB 为例, 表 1 中给出了本文构造的引理数与文献 [6] 的规约和断言数的对比, 数量大幅度减少, 验证过程示例如表 1 所示.

表 1 验证过程的对比

对比项	文献[6]	本文
中间断言数 (assert)	62	0
定理数 (requires+ensures)	158	84
过程示例		

(2) 验证脚本的模块化

在验证脚本的模块化方面, Dafny 的定理 (requires+ensures) 主要通过断言 (assert) 来描述, 难以继承 (如表 1 左边所示). 而 Isabelle 定理 (theorem+lemma) 支持模块化的设计, 可通过定义新的类型、常量和规则对父理论进行扩充, 有效地避免了重复的理论定义. 在与本文工作相关的 LLRB_SET.thy、proof_insert.thy 和 proof_delete.thy 这 3 个文件中, 也可以体现相互的复用关系. 如图 20 所示, proof_delete.thy 导入了 proof_insert.thy, proof_insert.thy 导入了 LLRB_SET.thy, 而 LLRB_SET.thy 又导入了 Isabelle 类库的 5 个已有的.thy 文件.

(3) 函数式程序设计的可执行性

文献 [6] 中 Dafny 设计的函数式程序只能为算法进行证明推理, 不可执行. 而 Isabelle 中设计的函数可自动转换为函数式编程语言 Haskell、OCaml 等, 可达到工业级应用. 如图 21 所示, 在 Isabelle 中定义的函数 *ins* 转换为 Haskell 可执行程序.

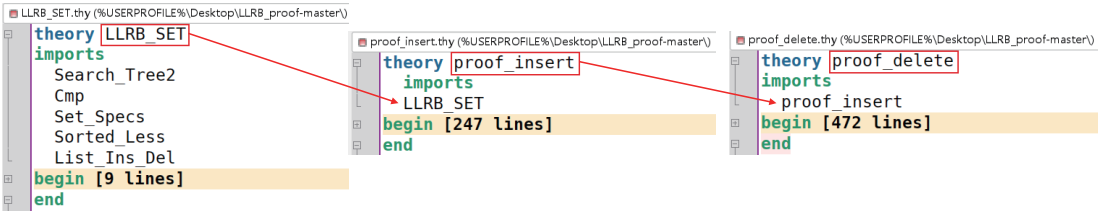


图 20　LLRB 算法在 Isabelle 中的验证脚本模块化

Example.hs (isabelle-export:Draft.proof_delete/code/example/)

```
{-# LANGUAGE EmptyDataDecls, RankNTypes, ScopedTypeVariables #-}

module Example(Linorder, Cmp_val, Color, Search_tree, ins) where {
import Prelude ((==), (/=), (<), (<=), (>=), (>), (+), (-), (*), (/), (**),
  (>>=), (>>), (=<<), (&&), (||), (^), (^^), (.), ($), ($!), (++), (!!), Eq,
  error, id, return, not, fst, snd, map, filter, concat, concatMap, reverse,
  zip, null, takeWhile, dropWhile, all, any, Integer, negate, abs, divMod,
  String, Bool(True, False), Maybe(Nothing, Just));
import qualified Prelude;
class Ord a where {
  less_eq :: a -> a -> Bool;
  less :: a -> a -> Bool;};
class (Ord a) => Preorder a where {};
class (Preorder a) => Order a where {};
class (Order a) => Linorder a where {};
data Cmp_val = LT | EQ | GT;
data Color = Red | Black;
data Search_tree a = Leaf | Node (Search_tree a) a (Search_tree a);
ins ::
  forall a. (Eq a, Linorder a) => a -> Search_tree (a, Color) -> Search_tree (a, Color);
ins x Leaf = Node Leaf (x, Red) Leaf;
ins x (Node l (a, Black) r) = (case cmp x a of {
                                 LT -> baliL (ins x l) a r;
                                 EQ -> Node l (a, Black) r;
                                 GT -> baliR l a (ins x r); });
ins x (Node l (a, Red) r) = (case cmp x a of {
                               LT -> Node (ins x l) (a, Red) r;
                               EQ -> Node l (a, Red) r;
                               GT -> Node l (a, Red) (ins x r);});};
```

图 21　自动转换为 Haskell 可执行程序

7　总结与展望

高效的数据结构在实际应用中起着重要的作用, LLRB 是由 Sedgewick[5]提出的一种二叉搜索树变体, 其结构比传统的红黑树多了“左倾”的约束条件, 在验证时无法使用常规的证明策略, 需要更多的人工干预和努力, 其正确性验证是一个公认的难题.

定理证明方法将程序和系统的正确性表达为数学命题, 然后使用逻辑推导的方式证明正确性, 可覆盖所有边缘情况. 由于 Isabelle 定理证明器具有优良的逻辑框架和可扩展性, 本文基于 Isabelle 主要完成以下工作: (1) 探究二叉搜索树类算法之间共性, 用区域刻画了二叉搜索树类结构插入和删除高阶泛化函数, 实现了程序的复用; (2) 挖掘验证规约与辅助引理之间的关系, 对 Nipkow 提出的二叉搜索树类算法 Isabelle 验证框架附加性质部分进行细化, 从而给出具体化的验证方案; (3) 实例化区域后, 生成 LLRB 的函数式算法, 基于二叉搜索树算法的验证框架将 LLRB 的不变量划分为基本不变量 *bst*、高度不变量 *invh* 和颜色不变量 *invc*, 并给出了函数式 LLRB 插入和删除算法的正确性证明.

由于二叉搜索树变体的复杂性以及附加性质的差异性, 未来可将本文提出的建模和验证框架应用到更多二叉搜索树类变体结构中, 从而进一步检验和完善本文所提的建模和验证框架, 也可将本文所提的建模和验证框架思路推广到其他树结构算法的函数式建模及验证.

致谢 在此我们向对本文工作给予建议的江西师范大学计算机信息工程学院 2022 级硕士研究生柯雨含、刘增鑫等同学致以谢意, 同时对审稿人提出的宝贵建议表示感谢.

References:

[1] Leavens GT, Leino KRM, Müller P. Specification and verification challenges for sequential object-oriented programs. Formal Aspects of Computing, 2007, 19(2): 159–189. [doi: 10.1007/s00165-007-0026-7]

[2] Leino K, Moskal M. VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In: Proc. of the 2010 Tools and Experiments Workshop at VSTTE. 2010. 221–331.

[3] Song LH, Wang HT, Ji XJ, Zhang XY. Verification of file comparison algorithm fcomp in Isabelle/HOL. Ruan Jian Xue Bao/Journal of Software, 2017, 28(2): 203–215 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5098.htm [doi: 10.13328/j.cnki.jos.005098]

[4] Jiang N, Li QA, Wang LM, Zhang XT, He YX. Overview on mechanized theorem proving. Ruan Jian Xue Bao/Journal of Software, 2020, 31(1): 82–112 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5870.htm [doi: 10.13328/j.cnki.jos.005870]

[5] Sedgewick R. Left-leaning red-black trees. In: Proc. of the 2008 Dagstuhl Workshop on Data Structures, 2008. 17–25.

[6] Peña R. An assertional proof of red-black trees using Dafny. Journal of Automated Reasoning, 2020, 64(4): 767–791. [doi: 10.1007/s10817-019-09534-y]

[7] Appel A W. Efficient verified red-black trees. 2011. https://www.cs.princeton.edu/~appel/papers/redblack.pdf

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

[9] Reade CMP. Balanced trees with removals: An exercise in rewriting and proof. Science of Computer Programming, 1992, 18(2): 181–204. [doi: 10.1016/0167-6423(92)90009-Z]

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

[11] Andersson A. Balanced search trees made simple. In: Proc. of the 3rd Workshop on Algorithms and Data Structures. Montreal: Springer, 1993. 60–71. [doi: 10.1007/3-540-57155-8_236]

[12] Zhao YW. Functional programming and proof. Electronic textbook. 2021. https://www.yuque.com/zhaoyongwang/fpp/

[13] Okasaki C. Red-black trees in a functional setting. Journal of Functional Programming, 1999, 9(4): 471–477. [doi: 10.1017/S0956796899003494]

[14] Filliâtre JC, Letouzey P. Functors for proofs and programs. In: Proc. of the 2004 European Symp. on Programming. Barcelona: Springer, 2004. 370–384. [doi: 10.1007/978-3-540-24725-8_26]

[15] Bobot F, Filliâtre C, Marché C, Paskevich A. Why3: Shepherd your herd of provers. In: Proc. of the 1st Int'l Workshop on Intermediate Verification Languages. Wroclaw, 2011. 53–64.

[16] Nipkow T, Blanchette J, Eberl M, Gómez-Londoño A, Lammich P, Sternagel C, Wimmer S, Zhan BH. Functional algorithms, verified! 2021. https://functional-algorithms-verified.org

[17] Ahmadi R, Leino KRM, Nummenmaa J. Automatic verification of Dafny programs with traits. In: Proc. of the 17th Workshop on Formal Techniques for Java-like Programs. Prague: ACM, 2015. 4.

[18] Nipkow T, Klein G. Concrete Semantics: With Isabelle/HOL. Springer Int'l Publishing, 2021.

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

[20] Ballarin C. Tutorial to locales and locale interpretation. Contribuciones Científicas en honor de Mirian Andrés Gómez. Universidad de La Rioja, 2010. 123–140.

[21] Hoare CAR. Proof of correctness of data representations. In: Gries D, ed. Programming Methodology. New York: Springer, 1978. [doi: 10.1007/978-1-4612-6315-9]

[22] Jones CB. Program specification and verification in VDM. In: Logic of Programming and Calculi of Discrete Design. Berlin: Springer, 1987. 149–184. [doi: 10.1007/978-3-642-87374-4_7]

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

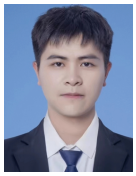
附中文参考文献:

[3] 宋丽华, 王海涛, 季晓君, 张兴元. 文件比较算法 fcomp 在 Isabelle/HOL 中的验证. 软件学报, 2017, 28(2): 203–215. http://www.jos.org.

cn/1000-9825/5098.htm [doi: 10.13328/j.cnki.jos.005098]
[4] 江南, 李清安, 汪吕蒙, 张晓瞳, 何炎祥. 机械化定理证明研究综述. 软件学报, 2020, 31(1): 82–112. http://www.jos.org.cn/1000-9825/5870.htm [doi: 10.13328/j.cnki.jos.005870]

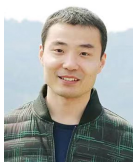
左正康(1980－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为形式化方法, 智能化软件.

曾志城(1999－), 男, 硕士生, 主要研究领域为定理证明, 形式化方法.



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

胡颖(1998－), 女, 硕士, 主要研究领域为定理证明, 形式化方法.



黄箐(1984－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为智能化软件.

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.

孙欢(1997－), 女, 硕士生, CCF 学生会员, 主要研究领域为定理证明, 形式化方法.