软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(9):4242−4264 [doi: 10.13328/j.cnki.jos.007135]

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

Trie+结构函数式建模、机械化验证及其应用*

左正康, 柯雨含, 黄 箐, 王玥坤, 曾志城, 王昌晶

(江西师范大学 计算机信息工程学院, 江西 南昌 330022)

通信作者: 王昌晶, E-mail: wcj@jxnu.edu.cn

摘 要: Trie 结构是一种使用搜索关键字来组织信息的搜索树, 可用于高效地存储和搜索字符串集合. Nipkow 等人给出了实现 Trie 的 Isabelle 建模与验证, 然而其 Trie 在存储和操作时存在大量的冗余, 导致空间利用率不高, 且仅考虑英文单模式下查找. 为此, 基于索引即键值的思想提出了 Trie+结构, 相较于传统的索引与键值分开存储的结构能减少 50% 的存储空间, 大大提高了空间利用率. 并且, 对 Trie+结构的查找、插入、删除等操作给出了函数式建模及其严格的机械化验证, 保证操作的正确性和可靠性. 进一步, 提出一种匹配算法的通用验证规约, 旨在解决一系列的匹配算法正确性验证问题. 最后, 基于 Trie+结构与匹配算法通用验证规约, 建模和验证了函数式中英文混合多模式匹配算法, 发现并解决了现有研究中的基于完全哈希 Trie 的多模式匹配算法的模式串前缀终止的 Bug. 该 Trie+结构以及验证规约在提高 Trie 结构空间利用率和验证匹配算法中, 有一定的理论和应用价值.

关键词: Trie+; 函数式建模; 机械化验证; 多模式匹配算法

中图法分类号: TP301

中文引用格式: 左正康, 柯雨含, 黄箐, 王玥坤, 曾志城, 王昌晶. Trie+结构函数式建模、机械化验证及其应用. 软件学报, 2024, 35(9): 4242–4264. http://www.jos.org.cn/1000-9825/7135.htm

英文引用格式: Zuo ZK, Ke YH, Huang Q, Wang YK, Zeng ZC, Wang CJ. Trie+ Structural Functional Modeling, Mechanized Verification and Application. Ruan Jian Xue Bao/Journal of Software, 2024, 35(9): 4242–4264 (in Chinese). http://www.jos.org.cn/1000-9825/7135.htm

Trie+ Structural Functional Modeling, Mechanized Verification and Application

ZUO Zheng-Kang, KE Yu-Han, HUANG Qing, WANG Yue-Kun, ZENG Zhi-Cheng, WANG Chang-Jing

(School of Computer and Information Engineering, Jiangxi Normal University, Nanchang 330022, China)

Abstract: A Trie structure is a type of search tree that organizes information by search keywords and can be employed to efficiently store and search a collection of strings. Meanwhile, Nipkow *et al*. provided Isabelle modeling and verification for Trie implementation. However, there is a significant amount of redundancy in the Trie's storage and operation, resulting in poor space utilization, and only the English single-mode lookup is considered. Therefore, based on the idea that the index is the key value, this study proposes the Trie+ structure, which can reduce storage space by 50% compared to the traditional structure of storing the index and key value separately, and greatly improve space utilization. Furthermore, the Trie+ structure's lookup, insertion, and deletion operations are modeled as functions and rigorously mechanized to ensure their correctness and reliability. Additionally, a generalized verification protocol for matching algorithms is proposed to solve the correctness verification and problems of a series of matching algorithms. Finally, a functional Chinese-English hybrid multi-pattern matching algorithm is modeled and verified by the Trie+ structure and the matching algorithm's universal verification protocol, and the Bug of prefix termination of pattern strings in multi-pattern matching algorithms of existing research based on the full hash Trie is discovered and solved. The proposed Trie+ structure and verification protocol have theoretical and application significance in

* 基金项目: 国家自然科学基金 (62262031); 江西省自然科学基金 (20232BAB202010); 江西省教育厅科技重点项目 (GJJ210307, GJJ2200302); 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"专题特约编辑曹钦翔副教授、宋富研究员、詹乃军研究员推荐.
收稿时间: 2023-09-11; 修改时间: 2023-10-30; 采用时间: 2023-12-13; jos 在线出版时间: 2024-01-05
CNKI 网络首发时间: 2024-04-28

improving the space utilization of the Trie structure and verifying the matching algorithm.

Key words: Trie+; functional modeling; mechanized verification; multi-pattern matching algorithm

Trie 结构[1]是一种使用搜索关键字来组织信息的搜索树, 可用于高效地存储和搜索字符串集合[2]. 它在数据压缩、计算生物学、用于 IP 地址路由表的最长前缀匹配算法、字典的实现、模式搜索、存储/查询 XML 文档等方面有着广泛的应用[3−6]. 而且它提供了按字母顺序过滤的功能, 因此可以在字符串匹配过程中更容易地按字母顺序打印所有单词[2].

目前已有研究主要从在结构上进行改进, 在保证高效的搜索下提高存储空间的利用率, 提出了 Binary Trie[7]—种基于二进制字符集的 Trie 树结构, 它在每个节点上使用两个指针连接到子节点, 结构简单但是其不适合处理非二进制数据且维护成本较高; Set-Trie[8]是一种基于 Trie 树的数据结构, 用于存储和操作集合数据. 它将集合元素作为节点存储, 并提供高效的集合操作和查询功能, 但它无法直接支持连续范围或精确的浮点数比较; Structure-Shared Trie[9]是一种基于 256 元数组来实现的 Trie 结构, 通过共享和将显式无用空间压缩为位来减少未使用的空间, 但是他们将字母表的大小限制在 256, 这对于处理许多语言的表示来说是不够的[10]. 它们有一个共同的问题: 都是针对某种具体的数据类型而没有更好的泛化性, 并且由于缺少形式化验证导致无法保证其操作的正确性. 基于函数式编程思想, 用泛化的一阶数据类型 (包括参数化数据类型和嵌套数据类型) 来定义 Trie 结构和对 Trie 的操作进行函数式建模, 可解决抽象数据类型的问题. 并且通过将设计规范和验证条件形式化为逻辑表达式, 然后使用自动化工具进行符号推理和模型检查, 以验证程序是否符合预期的规范和性质[11], 可解决操作的正确性问题. 其中 Isabelle/HOL 作为当前被广泛使用、LCF 方法的函数式机械化定理证明器[12]. 为此, Nipkow 等人在 Isabelle 中提出了递归的 Trie 结构并实现了其查找、插入、删除等操作的函数式建模和验证[13−15].

但是 Nipkow 等人所实现的 Trie 结构在存储和操作上会有大量的冗余且仅考虑英文模式下的查找. 因此, 本文工作首先基于索引即键值的思想提出新 Trie+结构, 去除节点冗余信息, 并对 Trie+结构进行了查找、插入、删除等操作的函数式建模, 其中删除操作对比文献 [15] 去除了冗余结构, 有效压缩了存储空间, 提高了空间利用率. 利用结构和元素的不变性对基本操作进行机械化验证, 保证了操作的正确性和可靠性. 其次在函数式建模中, 首次通过将中英文内码值独立封装成归纳类型 Datatype, 并应用于 Trie+结构及其基本操作函数中的高阶泛化参数, 进而实现了函数式中英文混合多模式匹配算法 (functional multiple pattern matching on Chinese/English mixed texts, FMPM) 的建模. 最后基于 Locale 区域首次提出了满足逻辑规约的匹配算法通用验证规约, 此验证规约可通过区域解释实例化为本文的 FMPM 算法进而验证算法的正确性, 发现并解决了现有研究中基于完全哈希 Trie 的多模式匹配算法[16]存在模式串前缀终止的 Bug. 基于所提的验证规约, 有望解决一系列匹配算法的正确性验证问题, 本文工作路线图如图 1. 由此, 本文的贡献总结如下.

(1) 基于索引即键值的思想提出新的 Trie+结构, 并对其进行了函数式建模与机械化验证, 解决 Nipkow 等人的 Trie 结构[14]存在冗余和仅考虑英文单模式下查找的问题.

(2) 基于所提 Trie+结构及其基本操作, 实现了 FMPM 算法的建模, 在保证精确匹配前提下能大大提高空间的利用率, 并解决了中英文混合多模式匹配下的漏匹配或误匹配的问题.

(3) 提出了匹配算法通用的验证规约, 实例化后可保证 FMPM 算法正确性, 并发现现有研究中基于完全哈希 Trie 的多模式匹配算法的 Bug, 解决了中英文混合多模式匹配算法的模式串前缀终止问题. 基于此验证规约, 有望解决一系列的匹配算法正确性验证问题.

本文第 1 节对相关工作进行介绍. 第 2 节对 Trie+的结构进行函数式建模, 刻画了该结构以及查找、插入、删除操作的函数式建模. 第 3 节对 Trie+结构及其基本操作进行机械化验证, 通过结构不变性和元素不变性对 Trie+操作进行正确性验证. 第 4 节对 Trie+结构进行应用, 实现了基于 Trie+的 FMPM 算法的建模, 给出了匹配算法的通用验证规约, 此验证规约可通过区域解释实例化为本文的 FMPM 算法进而验证了算法的正确性, 同时发现基于完全哈希 Trie 的多模式匹配算法的 Bug. 第 5 节是对全文的总结以及未来工作展望. 本文对 Trie+结构及其基本操作、基于 Trie+的 FMPM 算法及匹配算法的通用验证规约, 给出了建模及验证脚本, 具体可以参阅 https://github.com/wuyin517/Trie_plus

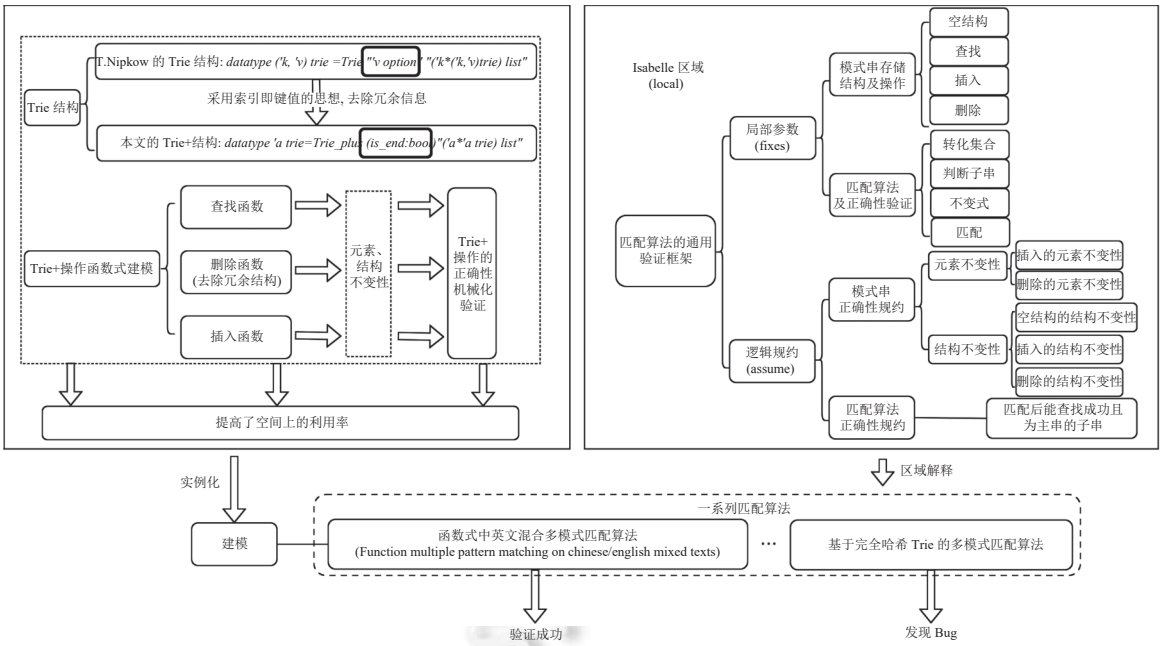


图 1 技术路线图

1 相关工作

在本节中, 将介绍与本文所提 Trie+结构相关的研究, 主要从 Trie 的结构及其变体、形式化验证、Nipkow 等人的 Trie 研究以及基于 Trie 结构的匹配算法这 4 个角度进行介绍.

Trie 结构和其变体: Trie 结构是一种经典的数据结构, 已被广泛研究和应用, 已有的工作主要集中在 Trie 结构的改进和变体上. 例如, Binary Trie[7]是一种基于二进制字符集的二叉 Trie 树结构, 它使用两个指针连接到子节点, 以实现高效的数据存储和检索. 尽管其结构相对简单, 但由于其特定于二进制字符集, 它在处理非二进制数据时存在限制, 这无疑增加了文本预处理的代价. 此外, 维护 Binary Trie 的成本较高, 需要考虑内存分配和指针管理等方面; Set-Trie[8]是一种基于 Trie 树的数据结构, 专门用于存储和操作集合数据. 它以集合元素作为节点存储, 提供高效的集合操作和查询功能. 然而, Set-Trie 无法直接支持连续范围或精确的浮点数比较. 这意味着在处理连续范围或浮点数时, 需要进行额外的转换或处理步骤; Structure-Shared Trie[9]是一种基于 256 元数组实现的 Trie 结构, 通过共享和位压缩来减少未使用的空间. 它通过将显式无用空间压缩为位信息来节省内存. 然而, 由于结构共享 Trie 将字母表大小限制在 256, 这导致不足以表达超过这一限制语言的字符集. 前面 3 种结构都是针对某一种单一的数据类型且大量使用“指针”, 同时也没有对其操作的正确性进行形式化验证, 从而导致 Trie 结构及其操作没有很好的泛化性且无法保证其操作的正确性.

Trie 结构形式化验证: 目前关于 Trie 的形式化验证主要分为手工和机械化两个方面. 在手工验证方面, 文献 [17] 通过手工将 Trie 扩展到术语索引, 其根据有限运算符签名构建的术语, 介绍了一些符号表示法, 并讨论了泛化 Trie 的属性, 包括它与其他数据结构的关系和潜在的应用. 但是它不仅工作量大, 而且因为无法处理更复杂的运算符和无限签名而无法覆盖所有情况. 在机械化验证方面, 已有文献[18,19]基于函数式编程实现了在 Coq 上的函数式建模与验证. 文献 [18] 在 Coq 上实现了对正数作为索引的 Trie 纯函数式的结构. 它们被用于编译器和静态分析器中的符号表、被用于表示从节点-ID 到边缘列表的映射有向图以及一般来说在命令式算法使用数组或需要可变指针的地方. 同时也给出了插入和查找的严格正确性验证, 但是缺少 Trie 结构的删除操作的函数式建模及正确性验证, 并且此结构只能用正数作为索引. 文献 [19] 同样用 Coq 实现了二进制 Trie 结构, 具有自然的扩展性, 有效地

支持稀疏性, 并且也提供了 Coq 中正确性的充分证明, 但是函数定义和证明的情况比较复杂, 导致证明成本较大.

Nipkow 等人的 Trie 结构及其机械化验证: Nipkow 等人在 Isabelle 中提出了 3 种不同递归的 Trie 结构并实现了其查找、插入、删除等操作的函数式建模和验证. 文献 [13] 这种结构利用字母和 Trie 递归结构的二元有序对来表示每个节点, 每个节点都包含一个将字母映射到子树的二元序对. 并且给出了对 Trie 结构操作的函数式建模及其相关性质的定理证明, 但是其节点存在冗余信息: 有效信息只存放在与字符串关联的最后一个节点上, 许多节点是不携带任何值的; 而且它的删除操作存在冗余结构现象: 旧的子 Trie 结构仍在关联表中, 这造成了大量存储空间浪费. 文献 [14] 对操作函数进行了优化, 去除了冗余结构, 但是只有在最后一个节点才携带有效信息, 大量节点存在冗余信息. 文献 [15] 对节点信息进行了改进, 使得每一个节点都携带有效信息, 但是仍存在冗余结构现象, 有大量存储空间的浪费.

Trie 结构应用于匹配算法: 文献 [20] 介绍了一种快速的多模式搜索算法, 即 Wu-Manber 算法. 该算法利用 Trie 结构在一段文本中同时搜索多个模式串, 提供了高效的字符串匹配和搜索功能. 文献 [21] 提出一种新型组合状态自动机模型. 在 Trie 结构基础上, 将中文字符高低字节拆分为虚、实两部分, 匹配时组合使用, 并结合 Quick Search (QS) 算法提升匹配效率. 以上算法仅通过测试, 无法对算法执行的所有可能做到全覆盖, 不能保证算法的正确性.

2 Trie+的函数式建模

本文提出了一种新的 Trie+结构, 这个结构是基于索引即键值的思想, 使得每一个节点都携带相应的有效信息, 通过使用终止符和一个带有映射信息的二元组列表可以去除大量的节点冗余信息, 利用了 HOL 标准库的 AList 包中的辅助函数可以去除冗余结构.

基于函数式程序设计的思想, 用泛化的一阶数据类型 (包括参数化数据类型和嵌套数据类型) 来定义 Trie+结构及 Trie+结构的基本操作函数式建模. 其中 Trie+结构利用自定义的递归函数来实现, Trie+结构的基本操作函数式建模包括查找、插入和删除函数.

2.1 Trie+结构的函数式定义

在 Isabelle 中提供归纳类型 datatype 来自定义自己所需的结构, 为此, 本文自定义的 Trie+结构如下所示, 其类型构造子 Trie_plus 由两个变量组合而成. 第 1 个变量表示字符串是否终止, 其中 is_end 是它的字段 (属性别名), 可直接访问其 bool 值, 第 2 个变量是一个二元组关联列表, 每一个二元组由一阶泛化类型'a 和'a trie 构成, 其中'a 类型可以实例化成任意所需的类型.

datatype 'a trie = Trie_plus (is_end : bool) "('a * 'a trie)list".

文献 [14] 中 Nipkow 等人提出的 Trie 结构如下:

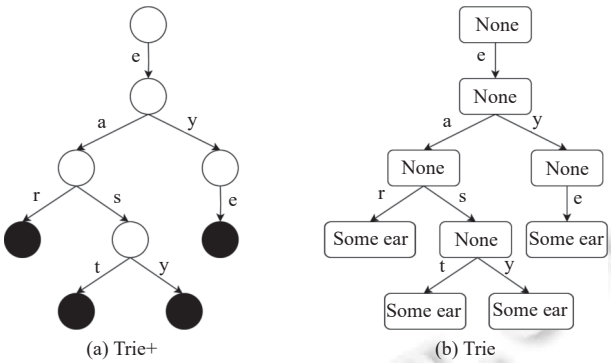
datatype ('k,'v) trie = Trie 'v option ('k*('k,'v)trie)list.

对比发现 Nipkow 等人的 Trie 结构[14]在存储时存在大量的'v option 类型, 且索引和键值分开存储导致同一个字符串会有两份同样的信息, 因此本文基于索引即键值的思想提出的 Trie+结构能减少 50% 的存储空间. 这里以一个简单的例子来进行对比, 例如, {ear, east, easy, eye}由图 2 中两种结构描述, 图 2(a) 是本文的 Trie+结构, 黑色节点 True 表示某一个字符串终止; 图 2(b) 是 Nipkow 等人的 Trie 结构, 当终止时'v option 返回值为 Some, 且重复存储当前字符串信息{Some ear, Some east, Some easy, Some eye}. 由图 2 可知本文的 Trie+结构在存储同一字符串的情况下能减少 50% 的空间.

在 Nipkow 等人的 Trie 结构中, 采用了'v option 类型, 这种设计非常灵活, 可以根据所需存放更多的信息, 但在存储字符串时可能会导致大量的冗余信息. 为了解决这个问题, 本研究将'v option 类型替换为 bool 类型, 同时将原本存储在'v option 中的索引信息作为键值来保存, 以保留'v option 中的有效相关信息. 因此, 本文提出的 Trie+结构在保留了相关的有效信息的情况下简化了其结构, 保证了其表达能力与 Trie 结构一致. 这种简化不仅提高了存储效率, 还为基于 Trie 类结构的较为复杂算法的形式化实现提供了坚实的基础.

同时, 本文定义了空的 Trie+结构, 该结构仅在 is_end 等于 False 且子 trie 列表为空列表时成立, 如下:

definition empty_trie :: “ ′a trie ” where “empty_trie = Trie_plus False []”.



(a) Trie+ (b) Trie

图 2 Trie+结构和 Trie 结构对比

2.2 Trie+基本操作的函数式建模

Trie+结构是一种支持查找、插入和删除操作的数据结构. 查找函数用于在 Trie+中搜索给定列表的值; 插入函数用于将待插入的列表的值添加到 Trie+中; 删除函数用于从 Trie+中删除指定列表的值. 给出这些操作的函数式建模, 使得其中的参数都是可泛化的.

2.2.1 查找函数

对于 Trie+结构的查找函数, 从根节点开始遍历到终止节点, 利用 map_of 函数将关联列表转化成偏函数映射关系, 便于快速查找当前节点是否在 Trie+结构上. 查找函数接受一个类型为′a trie 的 Trie+结构和一个类型为′a list 的键列表作为参数, 并返回一个 bool 值. 查找函数具体建模如下.

```
fun lookup_trie :: “ ′a trie ⇒ ′a list ⇒ bool” where
“lookup_trie (Trie_plus b m) [] = b” |
“lookup_trie (Trie_plus b m) (k#ks) = (case map_of m k of
                    None ⇒ False | Some st ⇒ lookup_trie st ks)”
```

上述函数是对类型为′a list 的键列表进行归纳, 当模式匹配到空列表时, 根据当前节点的 bool 值返回最终值, 当匹配到非空列表时, 只有当前字符匹配成功时, 才继续对下层 Trie+结构进行遍历查找, 如果出现一个在 Trie+树没有的节点值时, 则返回 False. 查找过程如图 3 所示.

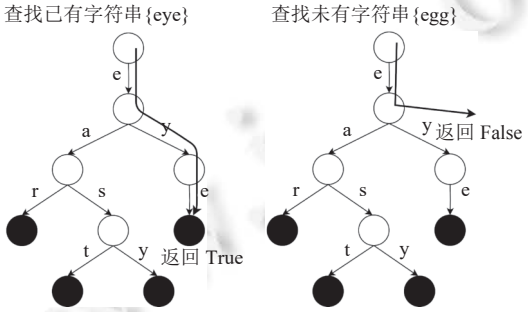


图 3 查找函数示例

2.2.2 插入函数

插入函数是将指定的列表键值插入到 Trie+树中得到新的 Trie+树, 这里利用 HOL 标准库中的 update_with_aux 函数将关联列表中根据指定索引键更新为对应的值, 并保证同一层的每个键只出现一次. insert_trie 接受一个

类型为′a list 的给定列表和一个类型为′a trie 的 Trie+作为参数. 插入函数具体建模如下.

```
fun insert_trie :: “ ′a list ⇒ ′a trie ⇒ ′a trie” where
“insert_trie [] (Trie_plus b m) = Trie_plus True m”|
“insert_trie (k#ks) (Trie_plus b m) = Trie_plus b
                        (AList.update_with_aux empty_trie k (insert_trie ks) m)”
```

上述函数对待插入列表递归的方式插入对应的节点. 根据给定列表的长度将对应的列表中的值逐层插入Trie+结构中, 直到到达给定列表的末尾, 其中, empty_trie 被用作在插入操作中的初始化状态. (1) 待插入列表为空时, 表示插入完成, 字符串已经终止并将最后一个节点的 is_end 字段设置为 True. (2) 待插入列表不为空时, 通过AList.update_with_aux 函数在 Trie_plus 结构的映射表中更新子节点的映射: 如果当前节点对应的键在映射表中不存在, 则新建一个子节点映射, 并继续递归调用 insert_trie 函数; 如果当前节点对应的键在映射表中存在, 则直接递归调用 insert_trie 函数, 再进行插入操作.

在图 4 中, 可以看到在已经包含字符串“ear”的 Trie+结构中插入另一个字符串“car”的过程, 这里可以将抽象的′a 实例化为实际的字符类型, 以便更好地表示. 由于字符“e”和字符“c”不同, 插入操作会在 Trie+结构中创建一个新的映射, 相当于在 Trie+树结构中另外开辟一条“支路”. 然后, 递归地向下继续插入字符, 直到最后一个字符, 将is_end 字段被设置为 True, 表示该字符串插入完成. 在这个基础上, 继续插入字符串“eye”. 当插入字符串中第 1 个“e”时, 由于 Trie+结构中已经存在字符“e”对应的映射, 插入操作不会创建新的节点, 在维持原有的映射上继续向下插入字符“e”之后的字符“y”. 按照同样的过程继续进行插入操作.

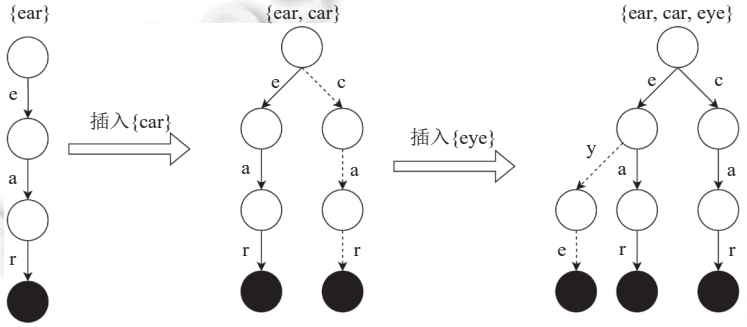


图 4 插入函数示例

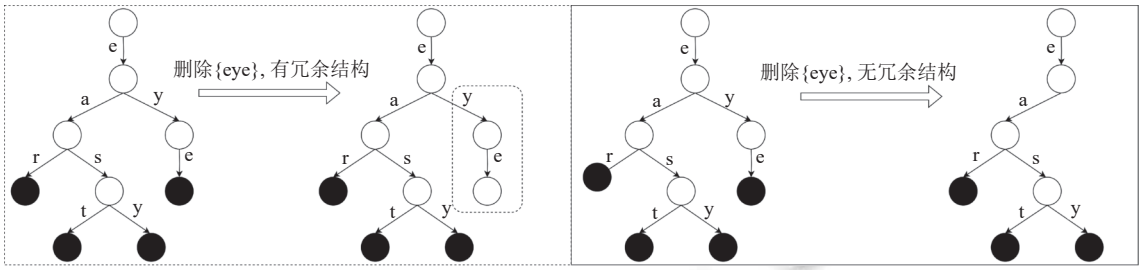
2.2.3 删除函数

删除函数是将 Trie+树中指定的列表键值删除后得到新的 Trie+树. 若删除后为空 Trie+结构, 则使用 HOL 标准库中的 delete_aux 函数, 该函数用于从关联列表中删除指定的键值对; 反则使用 HOL 标准库中的 update 函数, 该函数用于更新关联列表中的指定索引键对应的值. 这两个函数可以去除冗余结构, 从而减少了存储空间.

```
fun delete_trie :: “ ′a list ⇒ ′a trie ⇒ ′a trie” where
“delete_trie [] (Trie_plus b ts) = Trie_plus False ts” |
“delete_trie (k#ks) (Trie_plus b ts) = (case map_of ts k of
    None ⇒ Trie_plus b ts | Some t ⇒ let t′ = delete_trie ks t in if is_empty_trie t′
      then Trie_plus b (AList.delete_aux k ts) else Trie_plus b (AList.update k t′ ts))”
```

上述函数通过对待删除列表递归的方式删除对应的节点. (1) 如果待删除列表为空, 则表示已经到达目标节点, 将目标节点的 is_end 字段设置为 False. (2) 如果待删除列表不为空, 则继续递归下一层, 通过 map_of 函数在Trie+结构的映射表中找到对应的子节点, 然后根据子节点是否为空进行相应的操作: 如果子节点为空, 表示该键不存在于 Trie+中, 不进行任何操作, 直接返回原来的 Trie+结构; 如果子节点不为空, 则继续递归调用 delete_trie 函数, 最后根据递归的结果, 更新父节点的映射表: 如果子节点已经为空结构, 则删除该键值对, 如果子节点不为空结构, 则更新子节点对应的键值对.

与文献 [15] 相比, 本文的删除函数可在删除字符串时能够去除冗余结构. 图 5(a) 所示, 文献 [15] 中的删除操作仅仅只是将待删除字符串的终止节点染“白”, 从而以染“白”节点为终止的字符串无法再次被查找访问到. 在{ear, east, easy, eye}中删除{eye}, 仅仅将{eye}所在的支路的终止节点置为 False, 但是其相关信息存在此结构上导致有冗余结构. 图 5(b) 所示, 本文的 Trie+删除操作可以将{eye}中无用的支路去除.



(a) 文献[15] (b) 本文方法

图 5 文献 [15] 与本文的删除操作对比

2.3 Trie+与 Trie 的对比分析

本节对本文的 Trie+结构与文献 [14] 的 Trie 结构进行了深入的比较分析. Trie 结构作为一种经典的数据结构在字符串存储和检索中有广泛应用, 但其结构及算法实现存在冗余. 为解决这些问题, 在保持结构的表达能力一致且算法实现时间复杂度不变的前提下, 简化了结构及优化了算法实现中删除函数的建模, 显著提高了空间利用率. 我们将详细比较这两种结构的表达能力和时空复杂度, 如表 1 所示.

表 1 Trie 结构和 Trie+结构对比

对比	结构及操作	Trie	Trie+
表达能力对比	存储结构	索引和键值分开存储	索引即键值
时间复杂度对比	查找操作	O(n+m)	O(n+m)
	插入操作	O(n+m)	O(n+m)
	删除操作	O(n+m)	O(n+m)
空间复杂度对比	存储结构	有节点冗余信息	无节点冗余信息
	删除操作	有冗余结构	无冗余结构

表达能力比较分析: 与文献 [14] 的 Trie 结构的索引和键值分开存储相比, 本文引入的 Trie+结构, 其核心思想是基于索引即键值以保持必要信息, 保证了结构的表达能力与 Trie 结构一致.

时间复杂度比较分析: 本文中的 Trie+结构借鉴了 Trie 结构的递归方法, 对于 Trie+树的操作, 本文同样也采用了自上而下递归的函数式建模. 因此, 与文献 [14] 的 Trie 结构相比, 基本操作函数的时间复杂性上并没有增加额外的时间开销, 其各种操作的时间复杂度均相同. 在最坏情况下, Trie 结构和 Trie+结构的查找、插入和删除操作的时间复杂度均为 O(n+m), 其中 n 为查找字符的长度, m 为查找字符对应路径的长度.

空间复杂度比较分析: 本文引入的新 Trie+结构通过以保持必要信息的同时简化结构, 去除节点的冗余信息, 它仍然可以精确地表示存储的字符串, 如图 2 所示. 此外, 本文还优化了算法实现过程, 成功去除了删除操作中的不必要冗余结构, 提高了删除操作的空间利用率, 如图 5 所示.

综上所述, 本文的 Trie+结构通过以索引即键值为核心思想, 与文献 [14] 的 Trie 结构相比, 在保持结构的表达能力一致且算法实现时间复杂度不变的前提下, 简化了结构及优化了删除函数的建模, 显著提高了空间利用率.

3 Trie+的 Isabelle 机械化验证

机械化定理证明能够建立更为严格的正确性, 从而奠定系统的高可信性[12]. 在这个背景下, 基于 Isabelle 给出了 Trie+的机械化验证, 旨在验证 Trie+数据结构及其相关操作的正确性. 在第 2 节已经实现了 Trie+数据结构和基

本操作的函数式建模, 本节将利用 Isabelle 的推理规则和定理证明功能, 从结构不变性和元素不变性两个角度进行机械化验证, 以确保 Trie+中查找、插入、删除操作的正确性. 本节的所有定理和引理都通过 Isabelle 严格的机械化验证, 详见脚本: https://github.com/wuyin517/Trie_plus/blob/main/Trie_plus.thy

3.1 Trie+结构和元素不变性

在机械化验证过程中, 是通过验证数据结构或操作的不变性来保证其正确性的. 不变性是指在操作执行前后某些性质或特征始终保持不变. 对数据结构或操作进行不变性验证时, 可以通过证明不变式在操作前后为真或者满足某种等式来完成, 不变式[22]是指符合该数据结构的逻辑规则式.

结构不变性: 操作执行前后结构不变式为真. 本文给出的 Trie+结构不变式如下.

```
fun invar_trie :: " 'a trie ⇒ bool"where
"invar_trie (Trie_plus b m) = (distinct(map fst m)
                              ∧ (∀(k,t) ∈ set m. ¬ is_empty_trie t ∧ invar_trie t))"
```

结构不变式 invar_trie 利用 distinct 函数保证 m 中的键是互不相同的, 即不存在重复的键, 并通过递归使其对应的子 Trie+也满足结构不变式, 这确保每个节点都在 Trie+中存在唯一的映射. 特殊情况下空 Trie+结构也是满足这个结构不变式, 如下定理 1.

定理 1. 空 Trie+的结构不变性. theorem invar_empty: "invar_trie empty_trie".

元素不变性: 在集合中插入或删除元素后, 其他元素值和个数保持不变. 为此本文将 Trie+树转换成相应的集合, 利用集合简单的运算法则体现元素的不变性. 下面给出了集合转化函数, 其中 [simp] 表示该定义或者引理可用于之后化简证明.

```
definition trie_set :: " 'a trie ⇒ 'a list set" where[simp] : "trie_set t = {xs. lookup_trie t xs}"
```

对于空的 Trie+结构, 它的元素不变性如定理 2, 证明了空 Trie+对应的是空集.

定理 2. 空 Trie+的元素不变性. theorem empty_trie_set: "is_empty_trie t ⟹ trie_set t = {}".

定理 2 在证明过程中需要一些中间引理作为辅助引理, 其中它们的依赖关系如图 6.

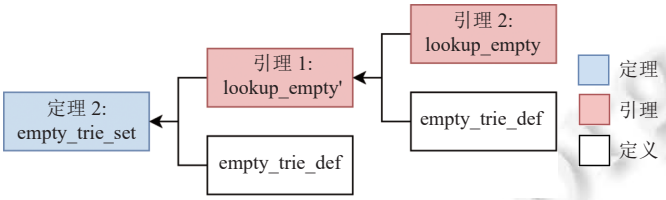


图 6 空 Trie+的元素不变性相关引理依赖关系

其中引理 1 作为主要的引理, 说明了对空 Trie+的形式化定义查找之后返回 False, 它是由引理 2 作为化简规则得到的, 其中 empty_trie_def 作为函数本身定义. 下面给出引理 1 和引理 2.

引理 1. lemma lookup_empty': "lookup_trie (Trie_plus False []) ks = False".

引理 2. lemma lookup_empty: "lookup_trie empty_trie = (λ_.False)".

3.2 Trie+操作函数的正确性验证

本节通过对 Trie+的结构不变性和元素不变性进行形式化的定义和验证, 结合归纳法对操作函数进行验证, 可在 Isabelle 中实现 Trie+操作函数的正确性验证. 这样的验证过程可以为 Trie+结构的使用提供更高的可信度, 确保操作函数的正确性和可靠性.

在证明过程中解决了以下难点, 首先是数据结构的复杂性, 证明中使用的数据结构是 Trie+, 它具有复杂的嵌套结构. 处理 Trie+数据结构的正确性和不变性需要深入理解该数据结构的内部工作方式, 并确保每一步都不会破

坏数据结构的完整性. 其次是分情况讨论, 证明中有多个情况需要讨论, 特别是在处理删除函数的不同情况时. 这需要详细考虑每种情况下数据结构的变化和属性的维护. 最后是构造多个关键引理, 证明需要多次引用先前证明过的引理, 以便推导更复杂的结果. 这要求清晰地理解不同引理之间的相互关系, 以确保它们在整个证明中协同工作.

3.2.1 查找函数正确性验证

在 Trie+满足结构不变式 invar_trie, 即结构保证了 Trie+的有效性和完整性, 对其进行查找, 结果应当只有查找成功和查找失败两种, 为此, 分别构造了定理 3 和定理 4. 定理 3 验证了查找成功的两种情况: 当查找的字符串为空且 Trie+的根节点为 True, 以及递归查找结束后到达 Trie+树的“黑色”节点. 同理, 定理 4 验证了查找失败的两种情况.

定理 3. 查找成功. theorem lookup_eq_True_iff: “invar_trie ((Trie_plus b kvs) :: 'a trie) ⟹ lookup_trie (Trie_plus b kvs) ks = True⟷(ks = []∧b = True)∨(∃k t ks′.ks = k # ks′∧(k, t)∈set kvs∧lookup_trie t ks′ = True)”.

定理 4. 查找失败. theorem lookup_eq_False_iff: “invar_trie ((Trie_plus b kvs) :: 'a trie) ⟹lookup_trie (Trie_plus b kvs) ks = False⟷ (ks = []∧b = False)∨(∃k ks′.ks = k # ks′∧(∀t.(k, t)∈set kvs ⟶ lookup_trie t ks′ = False))”.

图 7 给出了定理 3 和定理 4 相关的辅助引理以及它们之间的依赖关系.

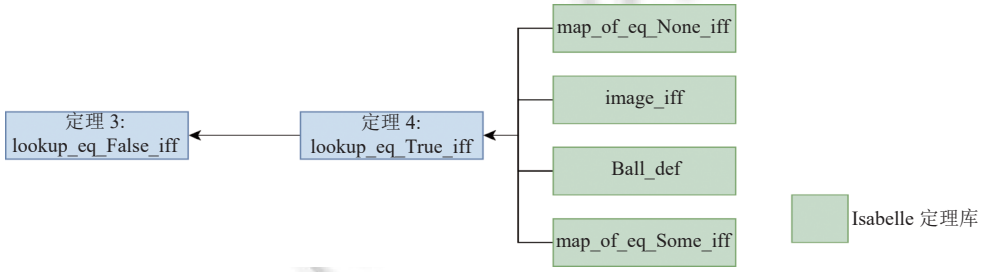


图 7 查找函数验证相关引理依赖关系

查找的结构不变性: 查找函数并没有对 Trie+结构中的值进行修改, 仅根据上述定理 3 和定理 4 返回查找后的结果, 因此其结构不变性显然成立.

查找的元素不变性: 查找某个元素成功时, 就意味着它是属于转化后的集合的, 同理, 失败等价于不属于. 以此构造的定理 5 和定理 6, 如下所示.

定理 5. 查找成功的元素不变性. theorem “lookup_trie t ls ⟹ ls ∈ (trie_set t)”.

定理 6. 查找失败的元素不变性. theorem “¬(lookup_trie t ls) ⟹ ls ∉ (trie_set t)”.

3.2.2 插入函数正确性验证

插入的结构不变性: 插入操作执行前后, 不破坏 Trie+的结构, 为此, 构造定理 7, 图 8 给出了证明定理 7 相关辅助引理以及它们之间的依赖关系.

定理 7. 插入的结构不变性. theorem invar_trie_insert: “invar_trie t ⟹ invar_trie (insert_trie ks t)”.

其中引理 3 说明了插入一个字符串到 Trie+之后, 一定不是空 Trie+结构.

引理 3. lemma insert_not_empty: “¬ is_empty_trie (insert_trie ks t)”.

这里给出了定理 7 在 Isabelle 中证明的具体过程, 如图 9 所示.

插入的元素不变性: 在集合中插入元素后, 其他元素值和个数保持不变, 在此基础上并上待插入的元素. 为此, 构造了定理 8, 图 10 给出了证明定理 8 所需的相关引理以及它们之间的依赖关系.

定理 8. 插入的元素不变性. theorem set_insert: “trie_set (insert_trie xs t) = trie_set t ∪ {xs}”.

定理 8 使用到的引理 4 和引理 5 如下.

引理 4. lemma lookup_trie_case: “lookup_trie (Trie_plus b m) xs = (case xs of [] ⇒ b |x # ys ⇒

(case (map_of m x) of None ⇒ False | Some t ⇒ lookup_trie t ys))”.

引理 4 表明在 Trie+数据结构中进行键值的查找操作, 逐层检索键列表, 并根据查找结果返回相应的值, False 表示查找失败.

引理 5. lemma lookup_insert: “lookup_trie (insert_trie ks t) ks′ = (if ks = ks′ then True else lookup_trie t ks′)”.

引理 5 表明如果插入的键与要查找的键完全相同, 则插入操作将确保查找结果为 True; 否则查找结果与原始 Trie+结构中的查找结果保持一致, 不受插入操作的影响.

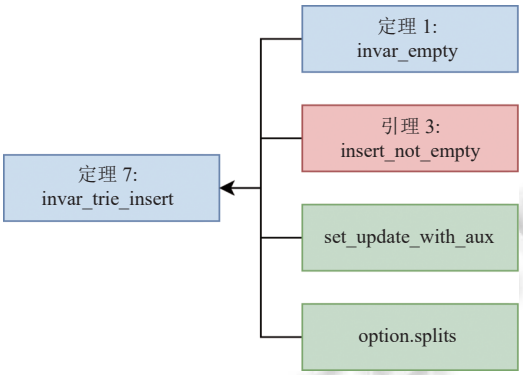


图 8　插入的结构不变性相关引理依赖关系

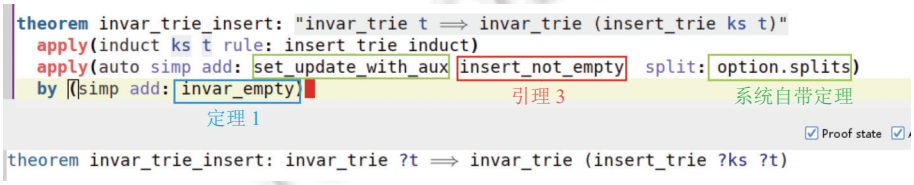


图 9　插入的结构不变性证明过程

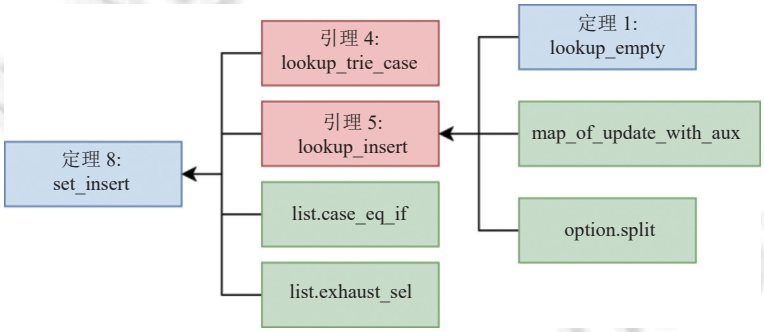


图 10　插入的元素不变性相关引理依赖关系

这里给出了定理 8 在 Isabelle 中的证明过程, 如图 11 所示.

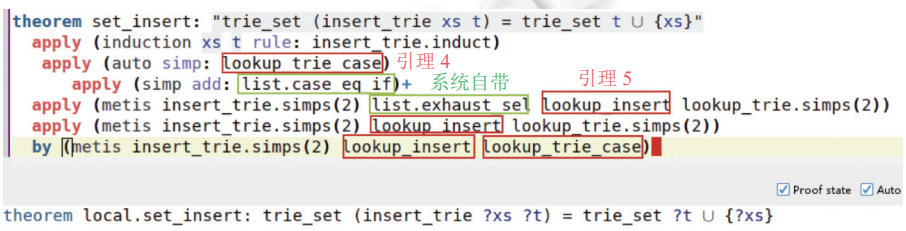


图 11　插入的元素不变性证明过程

3.2.3　删除函数正确性验证

删除的结构不变性: 同插入操作, 如定理 9 所示, 删除操作也不会破坏 Trie+的结构不变式.

定理 9. 删除的结构不变式. theorem invar_trie_delete: “invar_trie t ⟹ invar_trie (delete_trie ks t)”.

这个证明对比插入函数的结构不变性证明难度更大, 因为它涉及到的情况更多, 要对每一种情况分别证明. 在证明中, 首先考虑了删除空列表的情况, 该情况下不会改变 Trie+结构的性质. 然后对于非空列表的情况, 使用了模式匹配和条件分析, 分别考虑了映射中存在或不存在指定键的情况. 对于存在的情况, 通过递归调用删除操作并根据结果进行相应的更新, 确保了删除后的 Trie+结构满足不变性. 同时, 通过对映射和子结构的处理, 保证了 Trie+结构的有序性和节点的完整性. 最终, 根据不同的情况进行综合分析和推理, 得出了删除操作不会破坏 Trie+结构的性质.

这里给出了定理 9 在 Isabelle 中证明的具体过程, 如图 12.

```
theorem invar_trie_delete:
  "invar_trie t ⟹ invar_trie (delete_trie ks t)"
proof(induct ks t rule: delete_trie.induct)
  case 1 thus ?case by simp
next
  case (2 k ks vo ts)
  show ?case
  proof(cases "map_of ts k")
    case None
    thus ?thesis using "2.prems" by simp
  next
    case (Some t)
    with "2.prems" have "invar_trie t" by auto
    with Some have "invar_trie (delete_trie ks t)" by(rule 2)
    from "2.prems" Some have distinct: "distinct (map fst ts)" "¬ is_empty_trie t" by auto
    show ?thesis
    proof(cases "is_empty_trie (delete_trie ks t)")
      case True
      { fix k' t' [8 lines]
      with "2.prems" have "invar_trie (Trie_plus vo (AList.delete_aux k ts))" by auto
      thus ?thesis using True Some by(simp)
    next
      case False
      { fix k' t' [15 lines]
      thus ?thesis using Some "2.prems" False by(auto simp add: distinct_update)
    qed
  qed
qed
```

图 12　删除的结构不变性证明过程

删除的元素不变性: 同插入操作, 如定理 10 所示, 在集合中删除元素后, 其他元素值和个数保持不变, 在此基础上减去待删除的元素. 图 13 给出了证明定理 10 所需的相关引理以及它们之间的依赖关系.

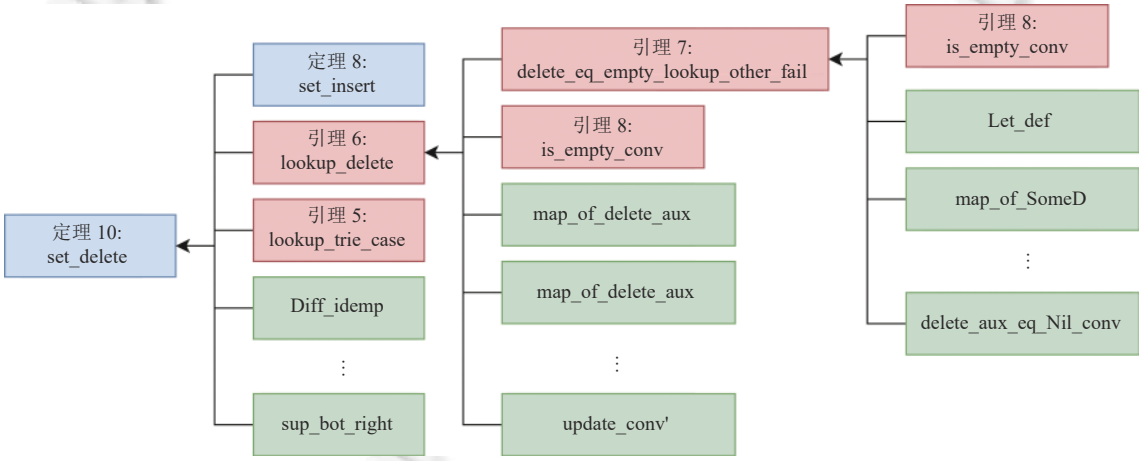


图 13　删除的元素不变性相关引理集关系

定理 10. 删除的元素不变性. theorem set_delete: “invar_trie t ⟹trie_set (delete_trie xs t) = trie_set t – {xs}”.

其中使用到引理 6–引理 8 如下所示.

引理 6. lemma lookup_delete: “invar_trie t ⟹lookup_trie (delete_trie ks t) ks′ = (if ks = ks′ then False else

lookup_trie t ks′)”.

引理 7. lemma delete_eq_empty_lookup_other_fail: “⟦ delete_trie ks t = Trie_plus False []; ks′ ≠ ks ⟧ ⟹ lookup_trie t ks′ = False”.

引理 8. lemma is_empty_conv: “is_empty_trie ts ⟷ ts = Trie_plus False []”.

这里给出了定理 10 在 Isabelle 中证明的具体过程, 如图 14.

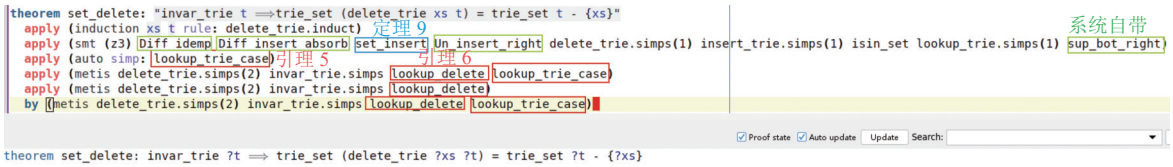


图 14 删除的元素不变性证明过程

4 基于 Trie+结构的应用

Trie 结构在数据压缩、计算生物学、IP 地址路由表的最长前缀匹配算法、字典的实现、模式搜索以及存储/查询 XML 文档等领域中有着广泛的应用[3−6], 尤其在匹配算法方面扮演着重要角色. 基于 Trie 实现的匹配算法在空间和时间复杂度上更加高效. 首先, Trie 结构以树状形式存储字符串, 利用共享公共前缀的方式节省了存储空间, 尤其在有大量重复前缀的情况下, 可以显著减少存储空间的使用. 其次, 基于 Trie 的匹配算法具有快速定位目标字符串或目标字符串前缀的能力, 从而提高了搜索的效率, 这种快速定位的特性使得在大规模数据集中进行搜索和匹配操作上更加高效.

4.1 函数式中英文混合多模式匹配算法

中英文混合多模式匹配算法[16]是一种特殊的多模式匹配算法, 专门用于在包含中文和英文的文本中同时查找多个目标模式. 可以帮助用户更有效地查找和分析包含多种语言的文本数据, 提高信息检索的广度和效率. 为此, 第 4.1 节从函数式编程的思想设计了函数式中英文混合多模式匹配算法. 与普通的 Trie 结构及其操作实现的匹配算法相比, 基于本文的 Trie+结构及其操作函数, 对函数式中英文混合多模式匹配 (FMPM) 算法进行了建模, 在存储大量的模式串时空间效率上有一定的优势. 为了验证包括 FMPM 算法在内的一系列匹配算法的正确性, 本文提出了匹配算法通用的验证规约. 利用这一规约, 对 FMPM 算法的正确性进行了严格验证, 并且还发现了现有研究中基于完全哈希 Trie 的多模式匹配算法中的模式串前缀终止的 Bug. 最后通过 Isabelle 中提供的自动转化 Haskell 代码方法将本文的 FMPM 算法更具可执行性, 从而达到工业界应用层面.

通过本小节的工作, 实现了 FMPM 算法, 并通过通用验证规约确保了其正确性. 这为处理混合语言文本的多模式匹配问题提供了一种可靠而高效的解决方案. 本文的研究工作不仅扩展了现有的匹配算法领域, 还提供了一种通用的验证方法, 可用于验证其他类型的匹配算法的正确性. 解决了一系列匹配算法中的正确性问题. 本节的 FMPM 算法建模及验证脚本见: https://github.com/wuyin517/Trie_plus/blob/main/match.thy. 匹配算法通用规约及其实例化脚本见: https://github.com/wuyin517/Trie_plus/blob/main/Locale.thy

4.1.1 中英文混合多模式匹配算法的函数式建模

已有的多模式匹配算法大都是面向单一字符环境, 应用于中英文混合环境时, 由于中文和英文的内码表示有时会发生歧义冲突, 导致存在漏匹配、误匹配等问题[23]. 为此, 本文基于 Datatype 方法来定义中英文统一的数据类型, 并将函数式中抽象数据类型实例化为中英文统一字符的方式解决漏匹配、误匹配等问题, 中英文统一的数据类型如下所示:

datatype ch = English nat | Chinese nat nat.

本文采用一种编码方案, 其中英文字符的内部表示通过单个自然数 (nat) 来表示, 并封装在一个名为“English”

的类型构造子内. 而中文字符则使用两个自然数表示其内部编码值, 并将它们封装在名为“Chinese”的类型构造子内. 这个编码方案的关键在于, 中文字符与英文字符之间以及中文字符与中文字符之间是相互独立且互不干扰的. 在文本处理过程中, 通过上述类型构造子将字符串的内部编码值进行封装, 以创建一种统一的“ch”类型的字符列表. 例如用{[Chinese 200 203, Chinese 195 241, Chinese 183 254, Chinese 206 241], [English 112, English 101, English 111, English 112, English 108, English 101]}表示{人民服务, people}, 这里的内码值用十进制表示. 匹配如图 15 所示, 如果在处理字符边界时出现错误, 导致匹配从“人”的低字节内码值 203 开始, 而使字节组合变为“203 195 | 241 183 | 254 206 | 241 112 | 101 | 111 | 112 | 108 | 101”, 显然会产生一连串的错位, 导致漏匹配; 如果错位的编码正好组成某个关键词, 则会导致误匹配. 而本文利用统一封装不会使一个 nat 值错位或重复表示而产生歧义的问题, 解决了中英文漏匹配或误匹配问题.

字符串	人		民		服		务		p	e	o	p	l	e
内码值	200	203	195	241	183	254	206	241	112	101	111	112	108	101
正确匹配	200	203	195	241	183	254	206	241	112	101	111	112	108	101
错误匹配	200	203	195	241	183	254	206	241	112	101	111	112	108	101
本文匹配 (正确)	Chinese	200	203	Chinese	195	241	Chinese	183	254	Chinese	206	241		
	English	112		English	101		English	111		English	112		English	108	101

图 15 错位匹配示意图

为了实现 FMPM 算法, 本节将模式串用上文的 Trie+结构存储, 可以有效提高存储空间的利用率. 在创建模式串时会大量涉及到上文的插入函数. 然后从目标文本的起始位置开始, 逐个字符地与模式串所在的 Trie+树结构进行匹配, 这个匹配过程主要是依赖于上文的查找函数. 继续递归遍历完整个目标文本, 返回匹配成功的字符串列表, 如有需要可通过删除函数实时修改 Trie+中的模式串, 使其更具灵活性. 由于这 3 个操作已在第 3 节进行了正确性验证, 这在较大程度上保证了 FMPM 算法的正确性.

下面给出了 FMPM 算法的具体函数式建模.

```
fun FMPM :: “ch trie ⇒ ch list ⇒ ch list list” where
“FMPM (Trie_plus b m) [] = []”|
“FMPM (Trie_plus b m) (x#xs) = (case prefix_lookup (Trie_plus b m) (x#xs) of []
                    ⇒ FMPM (Trie_plus b m) xs|sx ⇒ sx # FMPM (Trie_plus b m) xs)”
```

以上的 FMPM 函数, 匹配到两种模式的输入.

(1) 当目标文本列表为空时, 直接返回一个空列表.

(2) 当目标文本列表不为空时, 调用 prefix_lookup 函数获取以当前字符为前缀的目标文本列表在 Trie+中的路径, 并根据路径的结果进行匹配.

(a) 如果路径结果为空列表, 表示没有以此字符为前缀的完整匹配, 则跳过该字符继续递归调用 FMPM 函数, 在剩余的目标文本列表上进行匹配查找.

(b) 如果路径结果不为空列表, 则将此次匹配成功的字符串列表添加到返回的结果列表开头, 作为一次匹配成功的结构, 并继续递归调用 FMPM 函数, 在剩余的目标文本列表上进行匹配查找.

上述 FMPM 算法中使用到辅助函数 prefix_lookup、look_up. 下面给出了这两个函数的定义.

```
fun prefix_lookup :: “ch trie ⇒ ch list ⇒ ch list” where
 “prefix_lookup ks = (let l = (look_up t ks) in (if (lookup_trie t l) then l else []))”
fun look_up :: “ch trie ⇒ ch list ⇒ ch list” where
 “look_up (Trie_plus b m) [] = []”|
 “look_up (Trie_plus b m) (x#xs) = (case map_of m x of
        None ⇒ [] | Some st ⇒ (x # look_up st xs))”
```

函数 prefix_lookup 使用 look_up 函数获取目标文本列表在 Trie+中的路径, 其中 look_up 函数实际获取的是目标文本列表在 Trie+中能够匹配的最长前缀, 并使用 lookup_trie 函数验证该路径是否是一个完整的匹配. 如果验证结果为真, 则返回该路径, 否则返回一个空列表.

FMPM 函数匹配字符串的过程如图 16 所示.

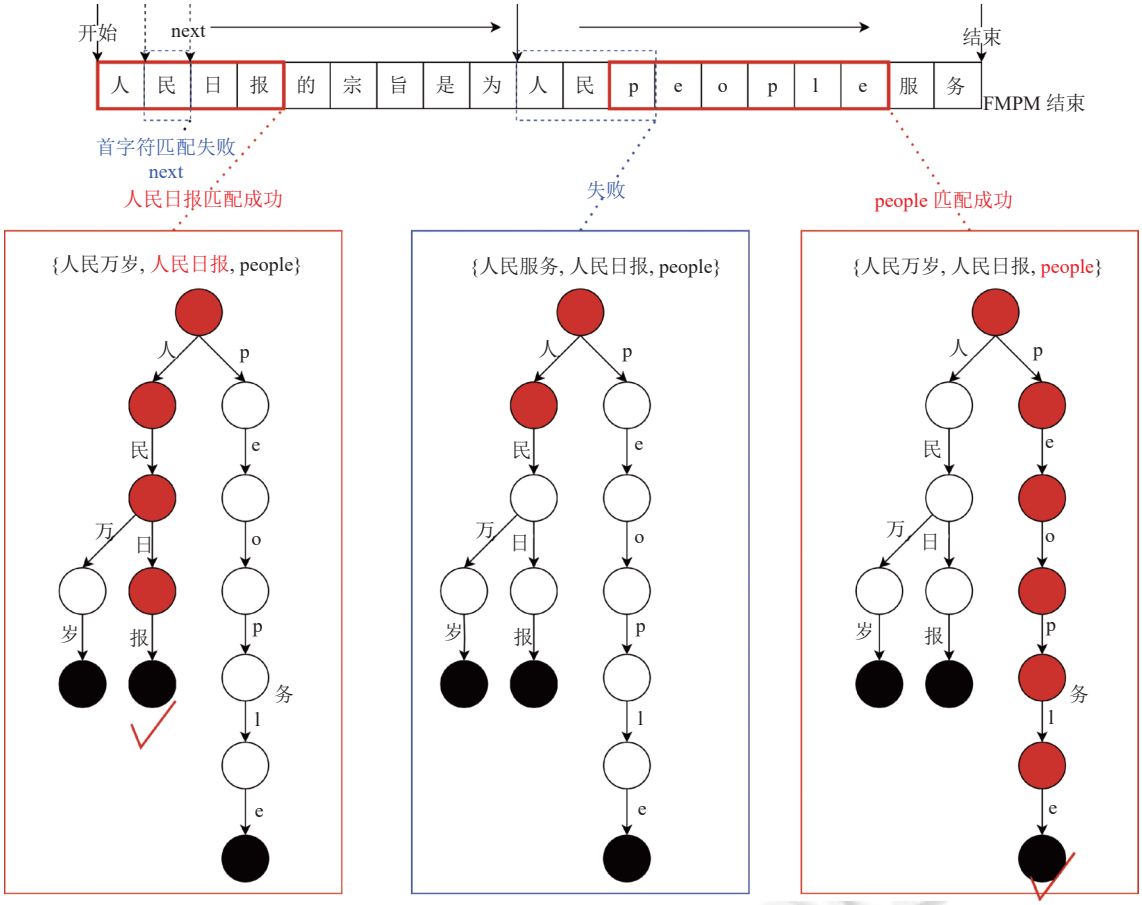


图 16 FMPM 算法匹配过程

图 16 中, 模式串为{人民万岁, 人民日报, people}, Trie+结构里本应存储的是内码 nat 值, 但是为了直观表示直接用字符表示, 目标串为“人民日报的宗旨是为人民 people 服务”, 首先从第 1 个字符“人”开始, 与 Trie+结构中的根节点匹配查找成功, 继续向下执行匹配, 直到“报” (叶子节点) 此次匹配结束, 返回第 1 次匹配成功的字符串“人民日报”. 接着递归执行 FMPM 算法从第 2 个字符“民”开始匹配, 首字符匹配失败直接跳过, 如此往复. 在匹配到第 2 个“人”时, 首字符匹配成功但是到了“p”当前匹配就结束了, 没到达叶子节点, 则此次匹配失败跳过. 后面的“people”匹配成功返回, 递归直到目标串结束. 所以最后匹配结果为{人民服务, people}.

4.1.2 中英文混合多模式匹配算法的机械化验证

4.1.2.1 匹配算法的通用验证规约

为了验证一系列匹配算法的正确性, 在本文中, 首次提出了一种通用的匹配算法的验证规约, 该规约可用于实例化验证各类匹配算法. 该规约基于 Isabelle/HOL 证明助手, 通过使用函数式编程思想和抽象数据类型的特性, 可以灵活地应用于不同类型的匹配算法, 无论是单模式匹配或多模式匹配, 还是英文或中英文混合模式匹配等, 并通

过严格的机械化验证过程, 证明匹配算法的正确性和可靠性.

区域 (Locale) 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[24]. 通过区域声明可定义通用的验证规约, 以规范定理和证明的作用范围, 并可对其进行实例化. 在 Isabelle 中, 使用 locale 关键字可以提供一个清晰、模块化和可维护的规约, 使得更加可读和易于理解且具有灵活性和可重用性.

因此, 基于 Locale 给出了匹配算法通用验证规约 Match, 构成了对匹配算法的严格约束, 确保了匹配算法在满足这些规约才能正确实现. 本文的通用验证规约是在 Nipkow 等人关于 Trie 结构及其操作验证规约的基础上, 根据匹配算法应满足的正确性规约扩展形成匹配算法通用验证规约 Match. 如图 17, 其中 Match 为区域的名字, 定义的局部变量用 fixes 表示, 用于存储模式串的结构及其操作的高阶泛化局部参数: 空结构 (empty)、插入 (insert)、删除 (delete) 、查找 (lookup); 用于匹配算法及其验证的高阶泛化局部参数: 结构不变式 (invar)、判断子串性质 (is_sub)、转化集合 (to_set) 以及匹配 (match). 它们之间满足的逻辑规约用 assumes 关键字表示: 空结构需要满足结构不变量 (invar_empty), 当存储模式串为空时, 其 Trie 结构也要满足结构不变式; 对结构进行插入操作, 要同时满足结构不变性 (invar_insert) 和元素不变性 (inser_set), 其中插入操作的结构不变性是指当存储模式串的 Trie 结构满足结构不变式时, 插入一个新的字符串后也应满足这个结构不变式, 插入操作的元素不变性是指在集合中插入元素后, 其他元素值和个数保持不变, 在此基础上并上待插入的元素; 删除操作同理; 最后匹配算法应满足的正确性规约: 匹配算法的返回值都能在模式串的结构中查找成功且是主串的子串 (match_correctness). 这 6 项逻辑规约同时为存储模式串的结构和匹配的局部参数定义了通用的接口要求, 这有助于确保匹配算法在不同场景下的正确性和可靠性.

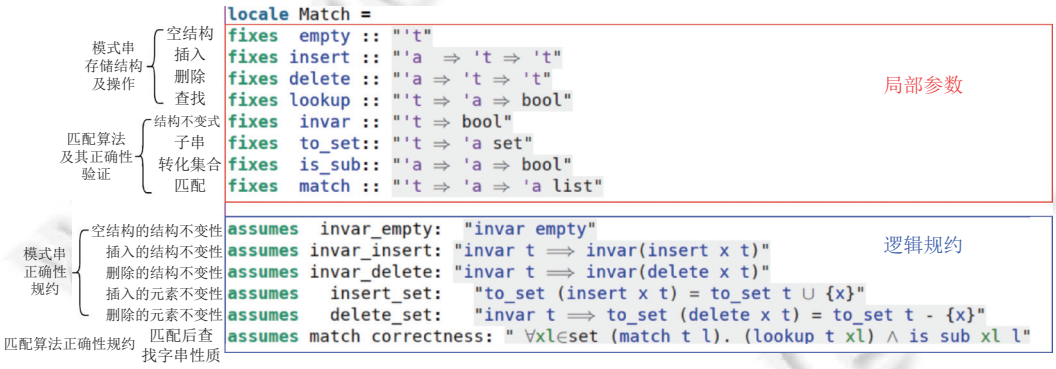


图 17 匹配算法通用验证规约

这个证明规约中的局部变量是高阶泛化的, 可以根据实际情况实例化为所需的数据类型和函数类型. 并且验证规约的优势在于它提供了一个清晰、模块化和可维护的方法来验证算法的正确性, 它将验证的关注点限定在特定的局部上下文中, 使得验证过程更加可读和易于理解. 因此, 这个规约具有灵活性和可重用性, 可以根据需要扩展和修改, 以适应不同的匹配算法或验证需求.

4.1.2.2 基于通用验证规约的 FMPM 算法验证

区域定义后, 可以在程序的上下文中进行动态地解释, 这样区域中的所有信息 (包括内部函数等) 都被传递到当前上下文, 从而可以在当前上下文中进行复用[25]. 通过 interpretation 命令可将区域及其内部信息传递到当前上下文将其实例化. 因此匹配算法通用验证规约 Match 可通过区域解释应用于本文的 FMPM 算法. 通过严格的机械化验证, 能够确保匹配算法的正确性和可靠性, 而且当前文献中尚未有此类算法的机械化验证案例, 这赋予了验证 FMPM 算法重要的理论和实际意义. 并且, FMPM 算法的建模采用 if、case 和 let 等语句复杂的嵌套递归, 同时引入了多个新的函数. 这意味着在证明其性质时需要全面考虑各种情况, 深入研究函数之间的潜在关系, 以提炼出多个关键引理.

在匹配算法通用验证规约中的局部变量是高阶泛化的, 可以根据实际情况实例化成所需的数据类型和函数类型: 空结构 (empty) 实例化为空 Trie+结构; 匹配 (match) 实例化为 FMPM 算法; 查找 (lookup) 实例化为 Trie+结构的查找函数; 其他参数类似, 如图 18 所示. 这样的模式串为 Trie+结构, 通过其基本操作可以构建模式串, 匹配的字符可以是自定义的中英文统一字符, 匹配状态可以是一个 bool 值, 匹配的结果可以是一个双重 list 等.

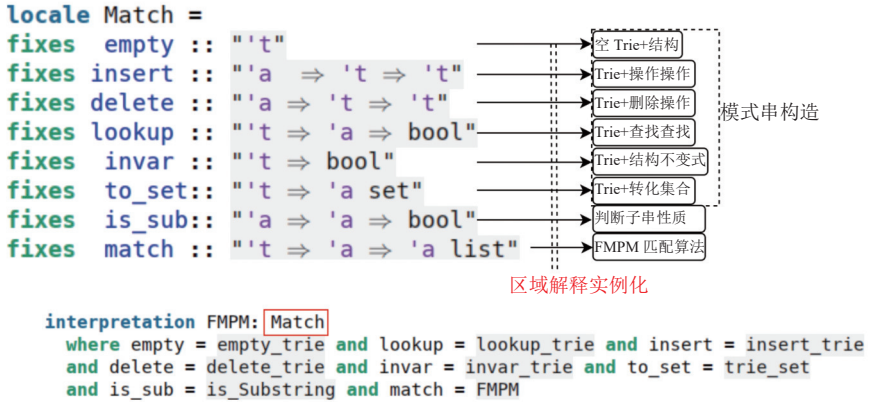


图 18　区域解释验证 FMPM 算法

在这个实例化过程中, 判断子串性质的具体函数如下所示, 其中 startWith 函数检查第 1 个串是否以第 2 个串开头. 而主函数 is_Substring 检查第 1 个串是否是第 2 个串的子串.

```
fun is_Substring :: " 'a list ⇒ 'a list ⇒ bool " where
    "is_Substring x [] = (x = [])" |
    "is_Substring x (y # ys) = (startsWith x (y # ys) ∨ is_Substring x ys)"
```

为此, 本文首先对 is_Substring 函数进行正确性验证, 采用与数学上的定义等价的方式如定理 11. 判断子串函数在数学上的定义如下, 定义中的等式 $s_2=s_3@s_1@s_4$ 表示存在两个列表 s_3 和 s_4, 使得将 s_1 插入到 s_3 和 s_4 之间后, 得到的结果与 s_2 相等. 这意味着 s_1 在 s_2 中以某种方式出现, 并且 s_3 和 s_4 分别是 s_1 的前缀和后缀. 如果存在这样的 s_3 和 s_4, 那么函数返回 True, 否则返回 False.

```
definition is_substring :: "'a list ⇒ 'a list ⇒ bool" where
                    "is_substring s₁ s₂ = (∃s₃ s₄. s₂ = s₃ @ s₁ @ s₄)"
```

定理 11. 子串等价定义. theorem is_Substring_equiv_is_substring: "is_Substring x y = is_substring x y".

引理 9–引理 11, 有助于理解字符串匹配和子串检查之间的关系, 并可以用于证明函数 is_Substring 的正确性.

引理 9. lemma is_sub_is_sub: "startsWith xs ys = (∃ls. xs@ls = ys)".

引理 10. lemma startsWith_sub: "startsWith x ys ⟹ is_substring x ys".

引理 11. lemma is_substring_add: "is_substring x ys ⟹ is_substring x (y#ys)".

图 19 给出了证明定理 11 所需的相关引理以及它们之间的依赖关系.

这里给出了定理 11 在 Isabelle 中证明的具体过程, 如图 20.

在匹配算法通用验证规约中的逻辑规约是需要通过严格的机械化证明的, 这 6 条规约中的前 5 条已经在第 3 节中通过了机械化验证. 按照规约中的顺序分别是定理 1 空 Trie+的结构不变性; 定理 7 插入的结构不变性; 定理 9 删除的结构不变性; 定理 8 插入的元素不变性; 定理 10 删除的元素不变性. 那么第 6 条匹配正确性规约是待证明的最后一个定理. 匹配规约中的逻辑规约验证过程如图 21 所示.

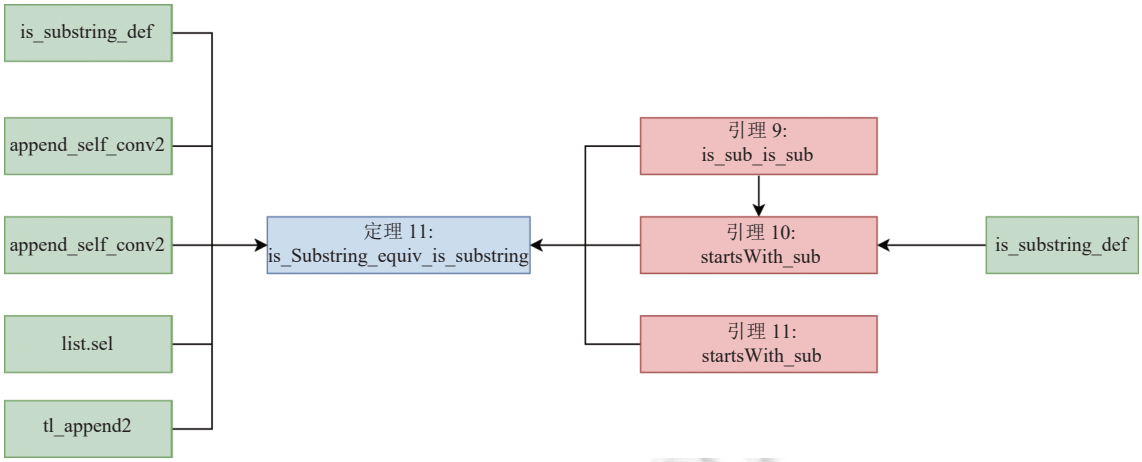


图 19 判断子串函数验证相关引理依赖关系

```
theorem is_Substring_equiv_is_substring:
  "is_Substring x y = is_substring x y"
proof(induct x y rule: is_Substring.induct)
  case (1 x)
  then show ?case
    by (simp add: is_substring_def)
next
  case (2 x y ys)
  then show ?case
  proof(cases)
    assume 01:"startsWith x (y#ys)" thus ?case using 2 is_Substring.simps(2)[of x y ys]
      by (metis  startsWith_sub)
  next
    assume 02:"¬ startsWith x (y#ys)" thus ?case using 2 is_substring_add is_Substring.simps(2)
      by (smt (verit, best) append_self_conv2 is_sub_is_sub is_substring_def list.sel(3) tl_append2)
  qed
qed
```

图 20 判断子串函数正确性证明过程

因此, 构造定理 12, 此定理说明 FMPM 算法匹配出来的结果列表中的每一个元素, 都能通过查找函数查找成功且都是主串的子串, 图 22 给出了证明定理 11 所需的相关引理以及它们之间的依赖关系.

定理 12. 匹配正确性. match_correct: “∀x∈set (FMPM t l). (lookup_trie t x)∧is_Substring x l”.

为正确证明定理 12, 首先将逻辑交运算拆开, 分别得到关键引理 12 和引理 13, 如下.

引理 12. lemma match_lookup_trie: “∀x∈set (FMPM t l). lookup_trie t x”.

引理 13. lemma match_is_sub: “∀x∈set (FMPM t l). is_Substring x l”.

同理构造了引理 14–引理 17, 如下.

引理 14. lemma prefix_lookup_lookup_trie: “prefix_lookup t l ≠ [] ⟹ lookup_trie t (prefix_lookup t l)”.

引理 14, 说明 prefix_lookup 前缀匹配成功时, 能通过查找函数查找出来. 它需要引理 10 和引理 11 作为辅助引理.

引理 15. lemma prefix_lookup_nil: “¬(lookup_trie t (look_up t ks)) ⟹ prefix_lookup t ks = [] ”.

引理 15 表明, 如果在给定的输入字符列表 ks 中没有找到匹配的前缀, 则 prefix_lookup 函数将返回一个空列表. 这个引理确保了在没有匹配时 prefix_lookup 函数的正确性. 引理 16 则表示, 如果在给定的输入字符列表 ks 中找到了匹配的前缀, 则 prefix_lookup 函数将返回与 look_up 函数完全相同的前缀. 这个引理保证了在匹配成功的情况下, prefix_lookup 函数会保持与 look_up 函数相同的结果.

```
assumes  invar_empty:  "invar empty"
assumes invar_insert: "invar t ⟹ invar(insert x t)"
assumes invar_delete: "invar t ⟹ invar(delete x t)"
assumes   insert_set:   "to_set (insert x t) = to_set t ∪ {x}"
assumes   delete_set:   "invar t ⟹ to_set (delete x t) = to_set t - {x}"
assumes match_correctness: " ∀xl∈set (match t l). (lookup t xl) ∧ is_sub xl l"
```

逻辑规约

通过严格的机械化验证

```
proof (standard, goal_cases)
  case 1
  then show ?case by (simp add:invar_empty)
next
  case (2 t x)
  then show ?case using invar_trie_insert by blast
next
  case (3 t x)
  then show ?case using invar_trie_delete by blast
next
  case (4 x t)
  then show ?case using set_insert by blast
next
  case (5 t x)
  then show ?case using set_delete by blast
next
  case (6 t l )
  then show ?case using match_correct by blast
qed
```

定理 1

定理 7

定理 9

定理 8

定理 10

验证过程

图 21　实例化为 FMPM 的逻辑规约证明

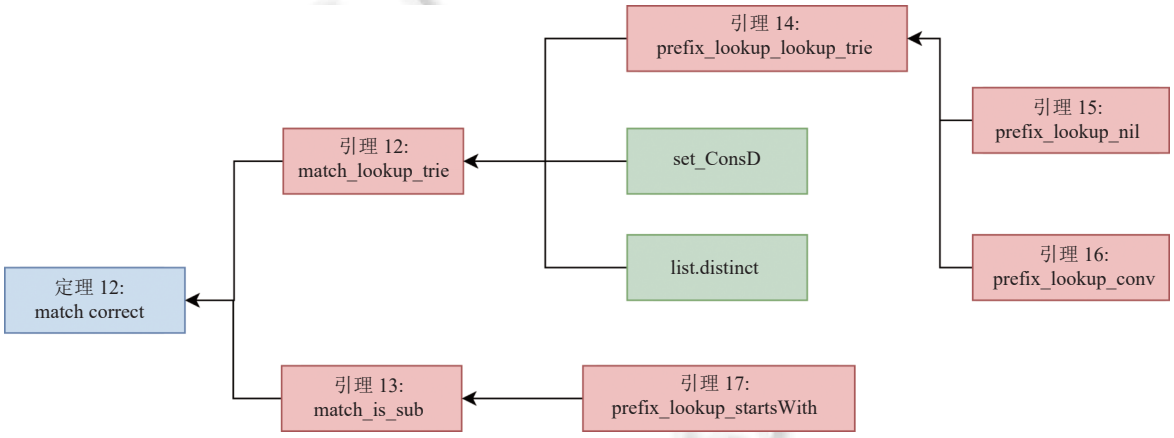


图 22　匹配后查找相关引理集关系

引理 16. lemma prefix_lookup_conv: “(lookup_trie t (look_up t ks)) ⟹ prefix_lookup t ks = (look_up t ks)”.

引理 17. lemma prefix_lookup_startsWith: “prefix_lookup t l = str ⟹ startsWith str l ”.

引理 17 表明 prefix_lookup 函数来判断某个字符串是否在 Trie+ 树中, 并返回匹配的前缀. 这个引理的证明确保了这个前缀是原始字符串的一部分.

关键引理 12 和引理 13 在 Isabelle 中的已通过证明, 如后文图 23 所示.

最后, 定理 12 是最终 FMPM 算法的正确性, 在 Isabelle 中的已通过证明, 如后文图 24 所示.

4.1.3　中英文混合多模式匹配算法的 Haskell 代码生成

Isabelle 中设计的函数具有高度抽象性和形式化的特点, 这使得它们在理论研究和形式化验证方面非常强大. 然而, 要将这些函数投入实际的工业应用中, 需要将它们转化为更具可执行性和效率的编程语言, 如 Haskell 或

OCaml. 这种转换不仅将形式方法的理论应用于实际问题, 还使其能够达到工业级的要求. 如图 25 所示, 在 Isabelle 中定义的函数 empty_trie、insert_trie、lookup_trie、delete_trie、FMPM 通过 Code Generation 方法[24]中 export_code 命令自动生成为 Haskell 可执行程序.

```
lemma match_lookup_trie:"∀x∈set (FMPM t l). lookup_trie t x"
  proof(induction t l rule: FMPM.induct)
    case (1 b m)
    then show ?case by simp
  next
    case (2 b m x xs)
    show ?case
    proof(cases "prefix_lookup (Trie_plus b m) (x#xs)" )
      case Nil
      then show ?thesis
        using "2.IH"(1) by fastforce
    next
      case (Cons a list)
      then show ?thesis
        by (metis "2.IH"(2) FMPM.simps(2) list.distinct(1) list.simps(5)
              prefix_lookup_lookup_trie set_ConsD)
    qed
  qed
```

引理 14

```
lemma match_is_sub:"∀x∈set (FMPM t l).  is_Substring  x l"
proof(induction t l rule: FMPM.induct)
  case (1 b m)
  then show ?case
    by auto
next
  case (2 b m x xs)
  then show ?case
    proof(cases "prefix_lookup (Trie_plus b m) (x#xs)" )
      case Nil
      then show ?thesis
        by (simp add: "2.IH"(1))
    next
      case (Cons a list)
      then show ?thesis  apply auto
        apply (metis Cons prefix_lookup_startsWith startsWith.simps(3))
        using "2.IH"(2)Cons apply blast
         apply (metis prefix_lookup_startsWith Cons startsWith.simps(3))
        using "2.IH"(2) Cons by blast
    qed
qed
```

引理 17

图 23　引理 12 和引理 13 正确性证明过程

```
theorem match_correct:"∀x∈set (FMPM t l). (lookup_trie t x)∧is_Substring x l"
  using match_lookup_trie match_is_sub by blast
```

引理 12　　引理 13

图 24　FMPM 算法正确性证明过程

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

module Match(Ch, fmpm) 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;
import qualified HOL;
import qualified Option;
import qualified Map;
import qualified List;
import qualified Trie_plus;
import qualified Arith;

data Ch = English Arith.Nat | Chinese Arith.Nat Arith.Nat;

equal_ch :: Ch -> Ch -> Bool;
equal_ch (English x1) (Chinese x21 x22) = False;
equal_ch (Chinese x21 x22) (English x1) = False;
equal_ch (Chinese x21 x22) (Chinese y21 y22) =
  Arith.equal_nat x21 y21 && Arith.equal_nat x22 y22;
equal_ch (English x1) (English y1) = Arith.equal_nat x1 y1;
```

```
instance Eq Ch where {
  a == b = equal_ch a b;
};

look_up :: Trie_plus.Trie Ch -> [Ch] -> [Ch];
look_up (Trie_plus.Trie_plus b m) [] = [];
look_up (Trie_plus.Trie_plus b m) (x : xs) = (case Map.map_of m x of {
       Nothing -> [];
       Just st -> x : look_up st xs;
     });

prefix_lookup :: Trie_plus.Trie Ch -> [Ch] -> [Ch];
prefix_lookup t ks = let {
                       l = look_up t ks;
                     } in (if Trie_plus.lookup_trie t l then l else []);

fmpm :: Trie_plus.Trie Ch -> [Ch] -> [[Ch]];
fmpm (Trie_plus.Trie_plus b m) [] = [];
fmpm (Trie_plus.Trie_plus b m) (x : xs) =
  (case prefix_lookup (Trie_plus.Trie_plus b m) (x : xs) of {
    [] -> fmpm (Trie_plus.Trie_plus b m) xs;
    a : list -> (a : list) : fmpm (Trie_plus.Trie_plus b m) xs;
  });

}
```

图 25　自动转换成 Haskell 代码

4.2　完全哈希 Trie 多模式匹配算法的 Bug 发现

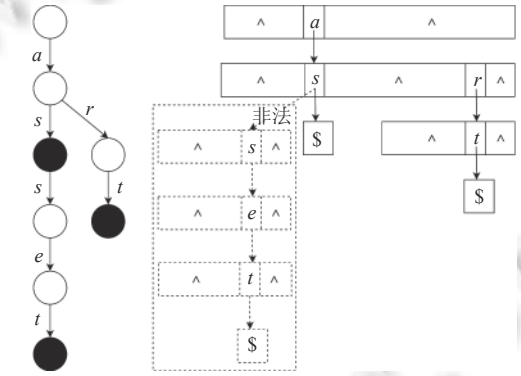
我们发现已有文献 [16] 中提出的完全哈希 Trie 多模式匹配算法违背了第 4.1.2.1 节给出的匹配算法的通用验证规约中插入的元素不变性的逻辑规约, 存在一个模式串前缀终止的 Bug. 具体地, 在构造模式串的过程中的代码如算法 1 所示.

算法 1. 完全哈希 Trie 匹配机构造算法.

```
typedef struct Head_Hash_Table { //首字符哈希表
    struct Trie_Node *head;
} Head_Index[CODE_LEN][CODE_LEN];
typedef struct Trie_Node { //完全哈希 Trie 结点
    struct Trie_Node *next[CODE_LEN];
```

```
} Trie_Node;
for (i=0;i<key_sum;i++){
  if (kw[i][0]>128) { //中文模式串
    len=strlen(kw[i]);
    if (!head_index[kw[i][0]][kw[i][1]].head) {
      p_1=malloc(sizeof(struct Trie_Node));
      head_index[kw[i][0]][kw[i][1]].head=p_1;}
    else
      p_1=head_index[kw[i][0]][kw[i][1]].head;
  for (k=2;k<len;k++){
    if (k<len−1){
      if (!p→1next[kw[i][k]]){
        p_2=malloc(sizeof(struct Trie_Node));
        p_1→next[kw[i][k]]=p_2;p_1=p_2;}
      else p_1=p_1→next[kw[i][k]];}
    else p_1→next[kw[i][k]]=END_FLAG;}}
  else { //英文模式串, 处理与中文模式串相似}
}
```

算法 1 使用指针作为映射且用特殊符号表示终止符, 无法在一个字符串结束终止时同时指向两个不同的内存空间, 这导致无法处理一个字符串是另一个字符串的前缀的情况. 以图 26 为例说明.



(a) Trie+ 结构 (b) 完全哈希 Trie 结构

图 26 Trie+结构与完全哈希 Trie 结构对比

假设已有的模式串为{as, asset, art}. 字符串{as}后面的指针已经指向终止符 END_FLAG('$'), 无法再指向有效的字符. 如果强行插入新的字符串{asset}, 导致之前的字符串无法引用, 也就无法通过查找函数查找到之前的字符串, 即"to_set (insert x t) = to_set t ∪ {x}", t 中原有的{as}在插入{asset}时被强行覆盖了, 元素的个数减少, 不满足通用验证规约中插入的元素不变性逻辑规约. 因此, 在匹配过程中会出现无法正确返回完整匹配结果的问题, 存在一个不满足匹配算法正确性的 Bug. 而在本文的 Trie+结构中, 采用了关系二元序对组成的列表来表示映射且通过 bool 类型作为终止符. 这种表示方法允许终止字符的节点还能继续映射下一个的节点, 解决了文献 [16] 中算法存在的问题. 因此, 通过使用 Trie+结构及其操作函数, 能够正确地处理中英文混合多模式匹配中模式串前缀终止的情况, 确保匹配结果的完整性和正确性.

这一发现不仅对于验证中英文混合多模式匹配算法的正确性具有重要意义, 也对于改进和优化其他基于完全

哈希 Trie 的多模式匹配算法提供了指导和启示.

4.3 基于 Trie+结构的 KBPs 系统应用

文献 [26] 提出了一种基于知识的程序 (knowledge-based programs, KBPs) 系统, 用于将一个代理的知识与其行为直接联系起来. KBPs 被用于描述那些需要基于其对环境的了解来做出决策的智能体或代理程序的行为. 文中提供了一个全面的解决方案, 它能够将基于知识的程序编译成可执行的自动机, 并使用 Isabelle/HOL 进行正确性证明. 这个算法是通过自上而下的方法开发的, 它使用了 Isabelle 的 Locale 机制来构建证明. 其中关键脚本 ClockView 用于记录了最近状态的当前时间和行为结果, 是一种具体的同步视图; SPRViewDet 和 SPRViewSingle 记录了代理程序在给定跟踪中所做的所有观察. 这 3 个脚本都通过 Maps 映射来表示有限环境中的状态和行为转换, 且均通过使用标准库中 Nipkow 等人的 Trie 结构, 用于表示状态和行为的集合及模型的构建, 如图 27 所示.

图 27 KBPs 主程序的依赖关系

本文的 Trie+结构在保留必要信息的前提下去除了不必要的冗余信息, 这种简化的 Trie+结构允许在表达能力不受损害且时间复杂度保持一致的情况下减少存储空间, 使其更适用于复杂数据处理和算法形式化建模需求. 因此, 将上述的 Trie 结构替换成 Trie+结构, 它消除了 Trie 结构中存在的大量冗余信息, 可以节省大量内存和存储成本. 其次, Trie+结构的操作算法经过优化, 删除函数的建模得到了改进, 提高了操作的空间利用率. 所以, 基于 Trie+结构的 KBPs 系统在保证正确性的同时能更加高效地运行, 同时也有望改善其他基于 Tire 结构的系统应用.

5 总结与展望

本文首次基于索引即键值的思想提出了新型 Trie+结构, 通过对 Trie+结构的插入、删除和查找等操作进行函数式建模与严格的机械化验证, 保证了这些操作的正确性和可靠性. 相较于 Nipkow 等人的 Trie 结构, Trie+结构能够显著减少存储空间, 提高空间利用率, 并且支持中英文混合多模式下的查找. 基于 Trie+结构应用在一种函数式中英文混合多模式匹配算法 (FMPM) 中, 实现了其函数式建模, 解决了中英文漏匹配和误匹配的问题. 此外, 本文还首次提出了通用的匹配算法验证规约, 为解决匹配算法的正确性验证问题提供了一种统一的方法. 通过将函数式建模和验证规约应用于 FMPM 算法, 本文不仅证明了该算法的正确性, 还发现了现有研究中基于完全哈希 Trie 的多模式匹配算法存在的 Bug, 并通过 Isabelle 中自动转化功能将 FMPM 算法转成 Haskell 代码, 使其更具可执行性. 最后可将 Trie+结构应用于 KBPs 等系统, 改善系统的空间效率.

本文的贡献在于引入了 Trie+结构和匹配算法通用验证规约, 解决了 Trie 结构存储空间冗余和中英文混合多模式查找限制的问题, 并为匹配算法的正确性验证提供了一种可行的方法. 基于这些贡献, 本文的工作具有一定的

理论和应用价值, 为提高 Trie 结构空间利用率和验证匹配算法提供了有效的解决方案. 未来的研究可以进一步拓展验证规约的应用范围, 并探索更多类型的匹配算法的正确性验证.

References:

[1] Fredkin E. Trie memory. Communications of the ACM, 1960, 3(9): 490–499. [doi: 10.1145/367390.367400]

[2] Hinze R. Generalizing generalized tries. Journal of Functional Programming, 2000, 10(4): 327–351. [doi: 10.1017/S0956796800003713]

[3] Ghuge S. Map and trie based compression algorithm for data transmission. In: Proc. of the 2nd Int'l Conf. on Innovative Mechanisms for Industry Applications (ICIMIA). Bangalore: IEEE, 2020. 137–141. [doi: 10.1109/ICIMIA48430.2020.9074934]

[4] Diop L, Diop CT, Giacometti A, Soulet A. Trie-based output space itemset sampling. In: Proc. of the 2022 IEEE Int'l Conf. on Big Data (Big Data). Osaka: IEEE, 2022. 6–15. [doi: 10.1109/BigData55660.2022.10020843]

[5] Indira B, Valarmathi K, Devaraj D. A trie based IP lookup approach for high performance router/switch. In: Proc. of the 2019 IEEE Int'l Conf. on Intelligent Techniques in Control, Optimization and Signal Processing (INCOS). Tamilnadu: IEEE, 2019. 1–6. [doi: 10.1109/incos45849.2019.8951425]

[6] Song HZ, Sun WT, Zhang MZ, Lu Y. XML index algorithm based on Patricia tries. In: Proc. of the 2009 Int'l Conf. on Web Information Systems and Mining. Shanghai: IEEE, 2009. 289–293. [doi: 10.1109/wism.2009.67]

[7] Gupta S, Garg ML. Binary trie coding scheme: An intelligent genetic algorithm avoiding premature convergence. Int'l Journal of Computer Mathematics, 2013, 90(5): 881–902. [doi: 10.1080/00207160.2012.742514]

[8] Savnik I, Akulich M, Krnc M, Škrekovski R. Data structure set-trie for storing and querying sets: Theoretical and empirical analysis. PLoS ONE, 2021, 16(2): e0245122. [doi: 10.1371/journal.pone.0245122]

[9] Tanhermhong T, Theeramunkong T, Chinnan W. A structure-shared trie compression method. In: Proc. of the 15th Pacific Asia Conf. on Language, Information and Computation. Hong Kong: City University of Hong Kong, 2001. 129–138.

[10] Huet G. A functional toolkit for morphological and phonological processing, application to a Sanskrit tagger. Journal of Functional Programming, 2005, 15(4): 573–614. [doi: 10.1017/S0956796804005416]

[11] Wang J, Zhan NJ, Feng XY, Liu ZM. Overview of formal methods. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 33–61. (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5652.htm [doi: 10.13328/j.cnki.jos.005652]

[12] 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]

[13] Nipkow T, Wenzel M, Paulson LC. Isabelle/HOL: A Proof Assistant for Higher-order Logic. Berlin: Springer, 2002. [doi: 10.1007/3-540-45949-9]

[14] Lochbihler A, Nipkow T. Trie. Archive of Formal Proofs. 2015. https://www.isa-afp.org/browser_info/current/AFP/Trie/document.pdf

[15] 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/

[16] Sun QD, Huang XB, Wang Q. Multiple pattern matching on Chinese/English mixed texts. Ruan Jian Xue Bao/Journal of Software, 2008, 19(3): 674–686 (in Chinese with English abstract). https://www.jos.org.cn/jos/article/abstract/20080318 [doi: 10.3724/SP.J.1001.2008.00674]

[17] Connelly RH, Morris FL. A generalization of the trie data structure. Mathematical Structures in Computer Science, 1995, 5(3): 381–418. [doi: 10.1017/S0960129500000803]

[18] Appel AW. Verified Functional Algorithms. Software Foundations Vol. 3. 2023. https://softwarefoundations.cis.upenn.edu/vfa-current

[19] Appel AW, Leroy X. Efficient extensional binary tries. Journal of Automated Reasoning, 2023, 67(1): 8. [doi: 10.1007/s10817-022-09655-x]

[20] Wu S, Manber U. A fast algorithm for multi-pattern searching. Tucson, AZ: University of Arizona, 1994.

[21] Shen Z, Wang YC, Liu GS. Improved multiple pattern algorithm for Chinese string matching. Journal of the China Society for Scientific and Technical Information, 2002, 21(1): 27–32 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-0135.2002.01.006]

[22] 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]

[23] Aho AV, Corasick MJ. Efficient string matching: An aid to bibliographic search. Communications of the ACM, 1975, 18(6): 333–340. [doi: 10.1145/360825.360855]

[24] Haftmann F, Bulwahn L, Nipkow T. Code generation from Isabelle/HOL theories. 2023. https://isabelle.in.tum.de/dist/Isabelle2023/doc/codegen.pdf

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

[26] Gammie P. Verified synthesis of knowledge-based programs in finite synchronous environments. 2023. https://www.isa-afp.org/browser_info/current/AFP/KBPs/outline.pdf

附中文参考文献:

[11] 王戟, 詹乃军, 冯新宇, 刘志明. 形式化方法概貌. 软件学报, 2019, 30(1): 33–61. http://www.jos.org.cn/1000-9825/5652.htm [doi: 10.13328/j.cnki.jos.005652]

[12] 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]

[16] 孙钦东, 黄新波, 王倩. 面向中英文混合环境的多模式匹配算法. 软件学报, 2008, 19(3): 674–686. https://www.jos.org.cn/jos/article/abstract/20080318 [doi: 10.3724/SP.J.1001.2008.00674]

[21] 沈洲, 王永成, 刘功申. 改进的中文字串多模式匹配算法. 情报学报, 2002, 21(1): 27–32. [doi: 10.3969/j.issn.1000-0135.2002.01.006]

[25] 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, 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.

王玥坤(1997－), 女, 硕士生, 主要研究领域为定理证明, 形式化方法.

柯雨含(1998－), 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.

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.