软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(2):508−542 [doi: 10.13328/j.cnki.jos.007560] [CSTR: 32375.14.jos.007560]

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

带递归定义的 SMT 公式求解技术综述*

冯维直[1,2,3], 刘嘉祥[1,2], 张立军[1,2,3], 吴志林[1,2,3]

[1](基础软件与系统重点实验室 (中国科学院 软件研究所), 北京 100190)

[2](计算机科学国家重点实验室 (中国科学院 软件研究所), 北京 100190)

[3](中国科学院大学, 北京 100049)

通信作者: 吴志林, E-mail: wuzl@ios.ac.cn

摘　要: 带有递归数据结构, 如列表 (*list*) 和二叉树 (*tree*) 等数据类型的程序, 在计算机领域被广泛使用. 程序验证问题通常将程序转换为可满足性模理论 (satisfiability modulo theories, SMT) 公式进行求解. 递归数据结构通常会转换为代数数据类型 (algebraic data type, ADT) 和整数等混合理论的一阶逻辑公式. 另外, 为表示递归数据结构的性质, 程序中通常需要包含递归函数, 递归函数在 SMT 中则需要通过包含量词和未解释函数的断言来表示. 关注带有 ADT 和递归函数这两类递归定义 SMT 公式的求解方法. 从 SMT 求解器、自动定理证明器和约束霍恩子句 (constrained Horn clause, CHC) 求解器这 3 方面对现有技术进行梳理和介绍. 同时, 对主流的求解工具进行统一实验对比, 探究现有求解工具和技术在各类问题上的优势和缺陷, 尝试寻找潜在的优化方向, 为研究者提供有价值的分析和参考.

关键词: 形式化方法; 递归函数; 可满足性模理论; 归纳推理; 引理合成; 约束霍恩子句

中图法分类号: TP301

中文引用格式: 冯维直, 刘嘉祥, 张立军, 吴志林. 带递归定义的SMT公式求解技术综述. 软件学报, 2026, 37(2): 508–542. http://www.jos.org.cn/1000-9825/7560.htm

英文引用格式: Feng WZ, Liu JX, Zhang LJ, Wu ZL. Survey on Solving SMT Formulas with Recursive Definitions. Ruan Jian Xue Bao/Journal of Software, 2026, 37(2): 508–542 (in Chinese). http://www.jos.org.cn/1000-9825/7560.htm

Survey on Solving SMT Formulas with Recursive Definitions

FENG Wei-Zhi[1,2,3], LIU Jia-Xiang[1,2], ZHANG Li-Jun[1,2,3], WU Zhi-Lin[1,2,3]

[1](Key Laboratory of Systems Software (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

[2](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

[3](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Programs with recursive data structures, such as *list* and *tree*, are widely used in computer science. Program verification problems are often translated into satisfiability modulo theories (SMT) formulas for solving. Recursive data structures are usually converted into first-order logic formulas combining algebraic data types (ADTs) and other theories such as integers. To express properties of recursive data structures, programs often include recursive functions, which in SMT are represented using assertions with quantifiers and uninterpreted functions. This study focuses on solving methods for SMT formulas with both ADTs and recursive functions. Existing techniques are reviewed from three perspectives: SMT solvers, automated theorem provers, and constrained Horn clause (CHC) solvers. Furthermore, the study conducts unified experiments to compare state-of-the-art tools on different benchmarks. It investigates the advantages and limitations of existing solving tools and techniques on various types of problems and explores potential optimization directions, providing valuable analyses and references for researchers.

Key words: formal method; recursive function; satisfiability modulo theories (SMT); inductive reasoning; lemma synthesis; constrained Horn clause (CHC)

---

* 基金项目: 中国科学院战略性先导科技专项 (XDA0320101)
收稿时间: 2025-02-18; 修改时间: 2025-07-23; 采用时间: 2025-09-26; jos 在线出版时间: 2025-12-10
CNKI 网络首发时间: 2025-12-11

随着计算机科学技术的发展, 计算机程序逐渐开始在社会各领域获得广泛应用. 对于安全攸关领域的程序, 代码中任何一点错误都可能造成极大的风险和经济损失. 程序形式化验证方法被提出用于提升程序可信度, 减少代码错误风险. 对程序进行形式化验证, 是使用数学方法将程序代码和程序需要满足的性质进行建模和表示, 然后从理论上证明程序代码满足或者违反性质. 程序形式化验证主要基于定理证明、模型检测、自动推理和约束求解等技术. 一般先在前端的程序代码层次进行分析, 基于程序语义将程序和待验证的性质编码为数学模型, 如迁移系统、自动机或霍尔三元组等, 然后生成使用一阶逻辑公式表示的验证条件 (verification condition), 最终调用后端的求解工具进行求解, 证明程序满足待验证性质或得到违反性质的反例.

对带有递归定义 (本文涉及的"递归定义"主要分为两类: 一类为递归数据结构, 另一类为递归函数) 的程序进行形式化验证, 是程序验证领域, 尤其是函数式程序验证, 广受关注且具有挑战性的重要问题之一. 该领域早期通常是在程序代码层次进行处理: 包括研究递归数据结构理论的判定算法、对递归函数进行展开、添加递归函数的前置-后置条件、基于定理证明工具手动进行归纳证明等. 最终生成尽量简单的逻辑公式作为验证条件, 再交由后端求解器完成求解. 随着后端求解技术的发展, 研究者逐渐开始直接在一阶逻辑公式层次对递归函数进行表示, 并研究在原一阶逻辑求解框架中加入处理递归数据结构和递归函数的技术. 相比在代码层次处理, 在逻辑公式层次的好处在于: 一阶逻辑公式的 SMTLIB 标准格式被形式化验证社区广泛认可和应用, 适合研究者在统一的样例上进行算法研究和实验对比; 后端求解器可以被应用于许多不同的编程语言形式化验证框架中, 具有更大的影响力和适用性; 另外在逻辑公式层次处理递归定义的特定技术可以更好地与原逻辑公式求解框架相结合. 本文将对这些在逻辑公式层次求解递归数据结构和递归函数的技术和主流工具进行介绍.

带背景理论的一阶逻辑公式可满足性问题: 命题逻辑是最基础的逻辑形式, 命题逻辑可满足性 (satisfiability, SAT) 问题是最早被证明的非确定性多项式完全 (non-deterministic polynomial-time complete, NPC) 问题. 但命题逻辑的表达能力有限, 在实际的程序验证问题中, 往往需要对特定背景理论下的计算进行描述和求解. 于是研究者将 SAT 问题扩展为可满足性模理论 (satisfiability modulo theories, SMT) 问题, 面向包含多种数据类型的一阶逻辑背景理论. 其背景理论涉及多种数学和计算机领域内的常用理论, 如布尔理论、线性整数/实数算术、位向量、未解释函数和数组理论等. 约束霍恩子句 (constrained Horn clause, CHC) 是另一种一阶逻辑公式表示形式, CHC 可满足性可以被视作 SMT 问题的特殊情况. 相比一般通用的 SMT 问题, CHC 的公式结构更适合表示带有循环结构的程序, CHC 公式的求解算法一般可以通过计算循环不变式来完成求解[1].

递归定义在程序代码和一阶逻辑公式层次的表示: 程序和一阶逻辑公式的递归定义主要分为如下两类.

1) 一类是递归数据结构, 递归数据结构由构造子 (constructor) 定义, 通常包含一个或多个基础情况 (base case) 和一个或多个递归情形 (recursive case), 通过引用自身的方式构造出复杂的数据对象. 通常包括树状结构 (一般由表示空树的基础构造子和表示非空树, 即根节点与左右子树的递归构造子定义)、列表结构 (一般由空列表的基础构造子和由头部元素与尾部列表组成的递归构造子定义) 以及基于 Peano 算术定义的自然数 (由表示零的基础构造子和表示后继函数的递归构造子定义) 等. 对带有递归数据结构的程序进行验证, 通常将递归数据结构转换为一阶逻辑公式中的代数数据类型 (algebraic data type, ADT), 通过求解 ADT 理论下的 SMT 问题进行验证 (详见第 1.3 节).

2) 另一类是递归函数, 递归函数是定义函数体中会调用自身的函数, 其函数结构通常包含一个用于终止计算的基础条件和一系列将复杂问题分解为更小同类问题的递归等式. 递归函数通常用来表示程序中一些需要通过循环或者递归计算的性质, 例如计算列表的长度等. 在一阶逻辑中通常将递归函数的定义分别转化为表示递归基础情形 (base case) 和递归情形 (recursive case) 的多条包含全称量词的逻辑公理, 该公理描述函数在所有可能输入上的行为. 递归函数 f 在逻辑上表示为如下的公理:

$$\forall x.\ (\varphi(x) \rightarrow f(x) = base(x)\ \wedge\ \neg\varphi(x) \rightarrow f(x) = rec(x, f(g(x)))),$$

其中, $\varphi(x)$ 表示终止条件, $base(x)$ 表示基础情形; $\neg\varphi(x)$ 表示未达到终止条件, $rec(x, f(g(x)))$ 定义递归情形下的公式, 表示递归情形中通过函数 g 构造规模更小的参数并递归调用 f 自身. 递归函数整体转换到在逻辑公式层次进

行表示, 通常是如下方式 (涉及的例子和细节详见第 1.5 节): 在 SMTLIB 标准格式中, 使用 declare-fun 声明函数名, 然后用带有全称量词和未解释函数 (uninterpreted function, UF) 的公理断言来表示函数体. 从 2016 年开始, SMTLIB 支持通过 define-fun-rec 的关键字直接定义递归函数, 主流 SMT 求解器比如 Z3 和 cvc5 在识别到输入带有 define-fun-rec 的函数定义时, 会采取一些特定求解策略. 验证目标被取反作为一条 assert 语句, SMT 求解器求解返回不可满足, 则认为验证目标得证. 两种方式在语义上可以认为是等价的.

通用的一阶逻辑公式求解框架: 现有的一阶逻辑公式自动求解工具在求解带有背景理论的一阶逻辑公式时, 通常是将可满足性判定算法和特定背景理论的求解技术相结合, 主要的一阶逻辑公式可满足性判定框架如下.

1) 影响力最大, 使用最广泛的是 SMT 求解器, 一般基于 DPLL (Davis-Putnam-Logemann-Loveland) 判定算法, 用于求解带背景理论的主流算法是 DPLL(T) 算法. 该算法求解原理是先不考虑背景理论, 将 SMT 公式视作 SAT 公式进行求解, 可满足时需再结合背景理论求解器验证解的相容性.

除 SMT 求解器外, 自动定理证明工具和 CHC 求解器同样可以进行一阶逻辑公式的求解, 现在主流的定理证明工具通常可以直接读入 SMTLIB 格式的 SMT 公式作为输入, 而 CHC 求解器则需要先将 SMT 公式转换成 CHC 形式再进行输入. 本文将不局限于 SMT 求解器, 还会介绍基于自动定理证明器和 CHC 求解器来处理带递归函数 SMT 问题的方法和工具.

2) 自动定理证明器一般基于 superposition 推理系统, 通过 saturation-based proof search 的证明框架来推导目标公式的可满足性. 对带有背景理论的一阶逻辑公式进行推理, 一般是将背景理论公理作为推理规则加入推理系统中. 用于推理带量词和背景理论的主流框架是 AVATAR 框架. 它结合了 SAT/SMT 求解器来提升在可满足性证明框架中进行公式推理的能力.

3) CHC 求解器的求解原理可以视作求解 CHC 公式所表达的迁移系统是否满足安全性质. 通常使用基于软件模型检测的方法, 结合谓词抽象、插值等技术来生成归纳不变式, 从而证明安全性质. CHC 求解器中一般也会集成通用 SMT 求解器, 并基于 SMT 求解器来提供对背景理论的支持, 不同的求解器则可能针对特定的背景理论问题实现专门的优化算法.

对递归定义问题的求解方法: 对于递归定义的处理, 递归数据结构和递归函数分别涉及对 ADT 理论和带有全称量词的未解释函数理论的求解. 在本文中我们将从 SMT 求解器、自动定理证明器和 CHC 求解器这 3 方面对这些特定背景理论的主流求解算法进行介绍.

1) 递归数据结构求解: 程序中的递归数据结构在逻辑公式层次一般用 ADT 理论公式进行表示.

SMT 求解器处理 ADT 理论公式的主流方法是基于 DPLL(T) 算法. 该算法将 SAT 的 DPLL 算法与 ADT 背景理论的判定算法相结合. ADT 理论的判定算法最早由 Oppen[2]提出, 其主要原理是将 ADT 结构进行展开, 对展开的项构造同余闭包和等价关系来完成可满足性的判定. Barrett 等人[3]在这一判定算法基础上引入一些启发式策略, 对计算效率进行了优化, 目前已经成为 SMT 求解器中实现 ADT 理论求解器的主流方法. Reynolds 等人[4,5]在上述方法上进一步扩展, 提出 codatatype 和共享选择子 (shared selector) 理论, 提升 ADT 理论的表达能力和求解效率. 上述基于 DPLL(T) 的判定算法被称为 lazy 方法, 作为主流 SMT 求解器如 Z3 和 cvc5 等用于进行 ADT 理论公式的主要方法. 此外有研究考虑先将 ADT 结构消去再进行求解 eager 方法[6,7], 主要原理是将带有 ADT 的一阶逻辑公式归约到等价的未解释函数和线性算术理论公式, 该方法实现在 SMT 求解器 Princess 中[6].

自动定理证明器求解 ADT 理论公式, 将基于 ADT 的构造子、选择子、子项等结构定义理论公理, 基于理论公理扩展推理系统的推理规则. 近年来比较经典的工作是 Cruanes[8] 和 Kovács 等人[9]对 superposition 推理系统进行扩展, 使其支持 ADT 理论和归纳推理的工作, 分别实现在自动定理证明工具 Zipperposition[10] 和 Vampire[11]中.

CHC 求解器求解 ADT 理论公式时, 需要将 CHC 求解框架和背景理论判定器结合, 其中背景理论判定器往往依赖 CHC 求解器中集成的 SMT 求解器. 为了提高求解效率, 有些 CHC 求解器中实现了专门针对 ADT 理论问题的优化算法. De Angelis 等人[12,13]提出一种转换算法, 可以将 CHC 公式中的 ADT 类型和对应的 CHC 公式转换成由其他基础类型, 如整数和未解释谓词表示的公式, 并保持可满足性, 该方法实现在 VeriMAP 求解系统中. Kostyukov 等人[14]通过将公式中的符号都转换为未解释函数, 然后将 CHC 求解中需要推理不变式的问题归约为在 tree auto-

maton 中寻找有限模型 (finite model finder) 问题进行求解, 该方法实现在 RInGen 求解器中.

2) 递归函数求解: 递归函数是程序中除了递归数据结构外的另一种常见递归定义. 递归函数在一阶逻辑公式层次一般表示为带有全称量词、ADT 理论、未解释函数理论和整数理论等混合理论的公式, 对这种混合理论公式的求解具有较大的挑战性. 近年来有不少相关的研究工作发表在形式化领域的重要会议或期刊上, 但即使是最好的算法和工具, 能处理的问题数目和形式也十分有限, 有较大的提升空间. 本文将分别介绍 SMT 求解器、自动定理证明器和 CHC 求解器所对应的一阶逻辑推理框架中对递归函数的处理方法.

SMT 求解器一般通过在 DPLL(T) 判定框架中加入归纳推理增强和自动生成辅助证明引理的方法来求解递归函数问题. 如前文所述, 表示带有递归函数问题的一阶逻辑公式一般为包括全称量词、未解释函数、ADT 理论和整数理论的混合理论公式. SMT 求解器在量词消去过程中引入表示归纳模式的断言, 然后通过一个专门的引理生成模块来提升理论判定过程中对复杂的未解释函数、ADT 理论和整数理论混合公式的处理能力.

自动定理证明器主要通过在推理系统中引入特定的归纳模式, 如结构归纳、整数归纳和基于函数定义的归纳模式等, 作为新的推理规则来提升推理能力. 并结合启发式优化方法提升推理过程的效率.

CHC 求解器通过尝试生成归纳不变式来证明待验证目标. 这一类方法的思路可以理解为将带有递归函数的逻辑公式判定问题视作检测迁移系统是否满足安全性质的模型检测问题. 在求解带递归函数 SMT 公式时, 需要先将 SMT 公式转换为 CHC 形式, 然后再调用 CHC 求解器. CHC 求解器对带有递归函数公式的求解依赖于其处理量词、未解释函数、ADT 理论和整数理论的能力. 主流 CHC 求解器可以在一定程度上求解这类问题, 但求解能力有限 (见本文第 5 节实验部分). 2022 年 Govind 等人[15]通过基于展开和抽象等特定的递归函数求解技术来提升 CHC 求解器处理递归函数问题的能力. 该方法在基于 CHC 求解器 Spacer 开发的 Racer 系统中实现.

总的来说, SMT 求解器和自动定理证明器主要关注在原推理框架基础上进行自动归纳推理增强, 并通过引理生成方法来辅助提升求解效率. CHC 求解器直接通过 CHC 求解框架生成原递归函数的归纳不变式来进行验证, 并可以基于递归函数展开和抽象等技术来优化求解效率.

主流求解工具实验对比: 本文从现有文献中选择了包含整数理论和 ADT 理论的公开数据集, 并从实际的程序验证问题中构造了一部分数据集作为补充. 在这些数据集上我们对主流的求解工具进行了统一的实验对比和分析, 并根据实验结果评估和分析了主流工具在不同类型的递归函数问题求解能力优劣. 本文期望为关注递归定义求解、引理生成和 CHC 求解等领域的研究者梳理重点求解技术和主流工具, 提供潜在的改进优化思路, 探究可能的研究方向.

1 相关背景知识

我们首先对本文涉及的一阶逻辑相关背景知识进行简单介绍, 包括可满足性问题、代数数据类型理论和 superposition 演算等相关内容.

1.1 可满足性问题和可满足性模理论

命题逻辑可满足性问题 (propositional satisfiability problem) 是逻辑学和计算机科学中重要问题, 一般简称为 SAT 问题, 指对于给定的一组布尔逻辑公式, 判定是否存在一组变量赋值使得该公式为真, 如果存在, 则称该公式可满足 (SAT), 否则称为不可满足 (unsatisfiable, UNSAT). SAT 问题是第 1 个被证明的 NPC (non-deterministic polynomial-time complete) 问题[16]. SAT 被广泛应用于电子设计自动化 (electronic design automation, EDA) 和程序验证分析等领域. 2000 年左右, SAT 求解算法取得突破, 可以处理大规模命题逻辑公式求解的 SAT 求解器开始出现并成为研究热点[17–20], 并且开始应用于工业界解决实际问题.

但 SAT 问题只考虑命题逻辑, 在许多实际场景下表达能力有限, 研究者考虑使用一阶逻辑公式与特殊背景理论进行融合, 提出可满足性模理论 (SMT) 问题. SMT 的基本思想是针对多种数据类型和相应的一阶逻辑理论, 提出一个一般的框架, 从而可以求解涵盖多种特定背景理论的一阶逻辑公式的可满足性判定问题. SMT 涉及的理论为一些数学理论和计算机领域内用到的数据结构理论, 主要理论包括等式未解释函数 (equality uninterpreted function,

EUF)、位向量 (bit vector, BV)、数组 (array)、线性整数算术 (linear integer arithmetic, LIA)、线性实数算术 (linear real arithmetic, LRA)、非线性整数算术 (nonlinear integer arithmetic, NIA)、字符串 (string)、代数数据类型 (ADT) 等. 求解 SMT 公式可满足性问题的工具被称为 SMT 求解器, 目前, 主流的 SMT 求解算法是 DPLL(T) 算法[21,22], 主流求解器有美国微软公司开发的 Z3 求解器[23]、美国斯坦福大学和爱荷华大学开发的 cvc5 求解器[24] (注意这里关于求解器名称的大小写问题, 根据 cvc5 相关论文[24], 在 CVC4 以前的求解器名称均使用大写字母“CVC”, 而 cvc5 决定使用小写的“cvc”字母)、由 Armin Biere 团队开发的 Boolector 求解器等[25,26]. 另外基于 superposition 推理系统[27]主流自动定理证明工具, 如英国曼彻斯特大学的 Kovács 等人[11]开发的 Vampire, 也能够接收 SMT 公式作为输入, 通过自动推理技术来实现对 SMT 公式的证明和求解[28].

1.2 一阶逻辑

本节简单介绍本文可能涉及的一阶逻辑相关背景的基本概念和符号标记. 其中文字 (literal)、子句 (clause)、逻辑连接符 $\neg$, $\wedge$, $\vee$, $\leftarrow$, $\leftrightarrow$ 以及全称量词 $\exists$, $\forall$ 等基本概念本文不再赘述.

前文提到 SMT 相比 SAT 的主要区别在于考虑“多种数据类型”和相应的一阶逻辑理论, 这里我们定义多种数据类型记号 (multi-sorted signature) 为如下几种符号集合: 函数符号 (function symbol) 的集合 $\mathcal{F}$、谓词符号 (predicate symbol) 的集合 $\mathcal{P}$、类型的集合 $\mathcal{S}$. 每一种符号都具有一个元数 (arity). 其中 0 元的函数被称为常数 (constant). 我们使用 $\top$ 与 $\bot$ 表示 0 元谓词 true 与 false. 一般用字母 f, g 表示函数符号, p, q 表示谓词符号, x, y 表示变量. 使用符号集合与变量递归定义“项 (term)”, 一般用字母 s, t 表示. 当一个项不包含变量时我们称其为基项 (ground term). 一个被解释符号 (interpreted symbol) 是一个意义被定义的函数或谓词. 例如当我们引入等式理论 (equality), 则等号符号“=”是等式理论中的被解释符号.

等式理论 (equality): 在一阶逻辑的语言中加入表示项 (term) 相等的二元谓词: “=”. 这里“=”作为被解释符号, 含义由等式理论中的公理所定义. 我们有如下公理.

1) $\forall x.\ x = x$ (自反性, reflexivity).

2) $\forall x, y.\ x = y \rightarrow y = x$ (对称性, symmetry).

3) $\forall x, y, z.\ x = y \wedge y = z \rightarrow x = z$ (传递性, transitivity).

4) 对任意正整数 n 和 n 元函数符号 f, $\forall \bar{x}, \bar{y}.\ (\bigwedge_{i=1}^{n} x_i = y_i) \rightarrow f(\bar{x}) = f(\bar{y})$ (函数同余性, function congruence).

5) 对任意正整数 n 和 n 元谓词符号 p, $\forall \bar{x}, \bar{y}.\ (\bigwedge_{i=1}^{n} x_i = y_i) \rightarrow p(\bar{x}) \leftrightarrow p(\bar{y})$ (谓词同余性, predicate congruence).

其中, $\bar{x}$ 表示变量列表 $(x_1, \ldots, x_n)$.

一个解释 (interpretation) $\mathcal{M}$ 是公式 F 的一个模型 (model), 我们记作 $\mathcal{M} \vDash F$, 此时 F 在模型 $\mathcal{M}$ 中取值为 true. 进一步, 如果 F 存在至少一个模型, 则称其为可满足的; 反之则称其为不可满足的. 对于公式 F, 如果所有的解释都是它的模型, 则称 F 是有效的 (valid), 记作 $\vDash F$. 一个位置 (position) 是一个正整数的有限序列, 定义为 $n \cdot p$, 其中 p 是一个位置, n 是正整数; 用 ϵ 表示空序列, 称为根位置 (root position). 位置用于表示项的子项. 设 t 为项, p 为位置, 则 t 在 p 处的子项记作 $t|_p$. 其归纳定义如下: (1) 当 $p = \epsilon$ 时, $t|_\epsilon = t$; (2) 当 $p = i \cdot p'$ 且 $t = f(t_1, \ldots, t_n)$ 时, 其中 $1 \leqslant i \leqslant n$, $t|_p = t_i|_{p'}$. 例如对形如 $f(g(a,b))$ 的项, 有 $f(g(a,b))|_\epsilon = f(g(a,b))$, 表示该项取位置 ϵ 时得到它自身. $f(g(a,b))|_{1\cdot 2\cdot \epsilon} = b$, 表示对 $f(g(a,b))$ 取位置 $1 \cdot 2 \cdot \epsilon$ 得到 b, 即 $f(g(a,b))|_1 = g(a,b)$, $g(a,b)|_2 = b$, $b|_\epsilon = b$.

一个替换 (substitution) θ 是一个形如 $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ 的映射, 其中 x 和 t 分别为一阶逻辑理论中的变量和项, 且对任意 $1 \leqslant i, j \leqslant n$, $i \neq j$, 有 $x_i \neq x_j$. 在一个表达式 E 上, 一个替换的应用 (application) 记作 $E\theta$. 这里表达式 E 中所有的 x_i 都被 t_i 所替换. 对依赖某个变量 x 的表达式 E, 我们记作 $E[x]$. 当 x 被一个洞 (hole) (用于表示一个占位符) 或一个项 t 替换时, 我们分别记作 $E[\cdot]$ 和 $E[t]$. 对项 s 中某个位置 p 的子项被 t 替代时, 可以表示为 $s[t]|_p$, 或简写做 $s[t]$. 当一个替换 θ 满足 $s\theta = t\theta$ 时, 称其为两个项 s 和 t 的合一子 (unifier). 此时称 s 和 t 为可合一的 (unifiable). 例如两个项 $s = f(x,y)$ 和 $t = f(a,z)$, 其中 x、y、z 是变量, a 是常数, f 是函数. 定义替换 $\theta = \{x \mapsto a, y \mapsto z\}$, 那么应用替换 $s\theta = f(a,z)$, $t\theta = f(a,z)$, 有 $s\theta = t\theta$, 则 θ 是 s 和 t 的合一子. 对于项 s 和 t 的合一子 θ, 如果他们的每一个合一子 η, 都存在一个替换 μ 使得 $\eta = \theta\mu$, 则称这个合一子 θ 为 s 和 t 的“most general unifier”, 简称为“mgu”, 即 $\theta = mgu(s,t)$.

如果两个项是可合一的 (unifiable), 则它们存在一个唯一的 *mgu*, 写作 $mgu(s,t)$.

在集合 A 上的一个二元关系 (binary relation) R 是笛卡尔积 $A\times A$ 的一个子集. 常见的二元关系如非自反关系 (irreflexive relation): 满足 $a\,R\,b$ 蕴含对任意 $a, b\in A$, $a\neq b$; 传递关系 (transitive relation): 满足对任意 $a, b, c\in A$, $a\,R\,b$ 且 $b\,R\,c$ 可推出 $a\,R\,c$. 称集合 A 上的一个关系 R 是良基 (well-founded), 当 A 的所有非空子集都至少有一个 R 关系下的最小元.

1.3 代数数据类型理论

代数数据类型理论 (下面简称为 ADT 理论) 是函数式编程与类型论中的重要概念, 在有些文献中也被称为归纳数据类型或递归数据类型 (inductive or recursive datatype)[4]. ADT 理论的公式通常用来编码程序中需要使用递归定义来描述的数据结构, 例如列表 (*list*) 和二叉树 (*tree*), 以及该数据结构相关的运算. 形式化定义代数数据类型的符号集 Σ 为类型 (sort) 的序列 $\sigma_1^d,\ldots,\sigma_k^d$ 和构造子 (constructor) 的序列 $f_1,\ldots,f_m$. 通常将 n 元构造子写作 $f_i:\sigma_1\times\ldots\times\sigma_n\to\sigma_0$, 其中 $\sigma_0,\ldots,\sigma_n\in\{\sigma_1^d,\ldots,\sigma_k^d\}$, 将 0 元构造子称作常数 (constant) 或基础构造子 (base constructor). 若 f_i 返回类型 σ_j^d, 例如 $f_i:\sigma_1\times\ldots\times\sigma_n\to\sigma_j^d$, 则写作 $f_i:\sigma_j^d$. 另外定义选择子 (selector) f_i^j, 表示提取一个 f_i -项的第 j 个参数; 定义测试子 (tester) $is_{(f_i)}$, 用于决定一个项是否是 f_i -项 (在有些文献中也称为 destructor). 代数数据类型理论中项 t 和公式 ϕ 的语法定义为如下规则.

t	::=	x	variable
	|	$f_i(\bar{t})$	constructor
	|	$f_i^j(t)$	selector
ϕ	::=	$is_{f_i}(t)$	tester
	|	$t=t$	equality
	|	$\phi\wedge\phi\,\|\,\phi\vee\phi\,\|\,\neg\phi\,\|\ldots$	Boolean operator

例 1: 在 ADT 理论中递归定义自然数 $\mathbb{N}$ 如下:

$$nat := O|s(x):nat.$$

定义类型 α 的列表 ($list(\alpha)$) 为:

$$list := nil|cons(x:\alpha, y:list),$$

则 *nat* 中 O 为基础构造子, s 为一元构造子, 定义选择子 p, $p(s(x))=x$. 则可以定义自然数中的一些基础项, 例如 0 定义为 O, 1 定义为 $s(O)$, 2 定义为 $s(s(O))$ 等. 另外由选择子定义有 $p(s(s(O)))$ 等于 1. *list* 中 *nil* 为基础构造子, *cons* 为二元构造子, 其两个参数分别为类型为 α 的 x 和类型为 *list* 的 y. 在有些文献或表示习惯中也写成中缀表达式::, 即 $cons(x,y)$ 等同于 $x::y$. 定义选择子函数 $cons^1$ 和 $cons^2$ (通常写为 *head* 和 *tail*), 有 $head(cons(x,y))=x$, $tail(cons(x,y))=y$.

若 α 为 *nat* 类型, 我们可定义一些自然数的列表, 如空列表 *nil*, 只有 1 个元素 0 的列表 l_0 为 $cons(O,nil)$, 由 1、0 组成的列表 l_1 为 $cons(s(O), cons(O,nil))$ 等. 且 $(l_1)=s(O)$, $tail(l_1)=cons(O,nil)$.

例 2: 我们考虑一个代数数据类型理论中可满足赋值的例子, 如下定义 *color* 和 *CList* 类型:

$$\begin{cases} color := red|green|blue \\ CList := nil|cons(h:Color, t:CList) \end{cases}.$$

于是这里类型 *color* 构造子为 *red*、*green*、*blue*, 它们都是基础构造子, 也称为常数或 0 元构造子. *CList* 构造子为 *nil* 和 *cons*, 其中 *nil* 为基础构造子, *cons* 为二元构造子 $cons:Color\times CList\to CList$. 定义 *head* 和 *tail* 为选择子: $head(cons(h,t))=h$, $tail(cons(h,t))=t$. 若给定类型为 *CList* 的变量 x 和类型为 *Color* 的变量 y, 我们可以构建一个 ADT 公式为:

$$is_{cons}(x)\wedge\neg(y=blue)\wedge(head(x)=red\vee x=cons(y,nil)) \tag{1}$$

其中, $is_{cons}(x)$ 为 tester, 它为 true 当且仅当 x 是具有 *cons* 构造子的项. 求解公式 (1), 可以找到一组可满足的赋值 $\{x\mapsto cons(red,nil), y\mapsto green\}$. 具体的 ADT 理论公式求解算法将在第 2.2 节中进行介绍.

1.4 Superposition 演算

Superposition 演算是定理证明系统中最主流的推理系统 (inference system)[8,11,27]之一. 定理证明问题可以视作由特定理论下的一组公理 (axiom) 和一个待证明的断言 (conjecture) 组成.

推理 (inference): 大多数定理证明器基于一个推理系统来自动地化简和生成公式. 这一过程被称为推理 (inference), 由如下的规则 (rule) 表示:

$$\frac{F_1, F_2, \ldots, F_n}{G},$$

其中, $F_1, F_2, \ldots, F_n$ 被称为前件 (premise), G 被称为推理的结论 (conclusion). 没有前件的推理称为公理 (axiom). 一些推理规则的集合构成一个推理系统 (inference system). 在推理系统中作为输入的公式通常为合取范式 (clausal normal form, CNF). 一般用 $cnf(F)$ 来表示将公式 F 转为 CNF 形式的子句集合.

化简 (simplifying): 当一个或多个前件因为他们对给出结论冗余, 能够从公式集合中去掉时, 称这个操作为化简 (simplify): 如 $\frac{F_1, F_2, \ldots, F_n}{G}$ 表示可将前件中的 F_1 消去.

合理和完备 (sound & complete): 当一个推理规则的结论在逻辑上由它的前件推出, 则该规则被称为合理的 (sound), 当一个推理系统的所有规则都是合理的, 则称这个推理系统为合理的; 当一个推理系统所有有效的公式集合都能在该系统中被证明为 true, 则该推理系统被称为完备的 (complete).

反证完备 (refutationally complete): 我们称在推理系统中结论为否定 (一般用 $\perp$ 表示) 的推理为“反证 (refutation)”. 反证完备指对任意不满足的公式集合, 都可以推理出空子句.

化简序 (simplification ordering): 在对 superposition 推理系统的具体推理规则进行介绍前, 我们首先引入化简序 (simplification ordering) 概念. 化简序是通过在推理系统中引入一个“优先级”来在指导推理过程中如何选择和化简子句: 选择子句指的是决定推理过程中需要被优先处理哪些子句, 化简子句指的是如何基于一些规则来删除冗余子句或者简化复杂子句, 从而减少搜索空间.

我们用符号 $\succ$ 来表示化简序, 并将其扩展到文字、子句和等式上. 一个等式两边的 l 和 r, 如果有 $l \succ r$, 则它们的方向为 $l = r$. 项上的一个序 $\succ$ 若满足以下 4 种性质, 则称之为化简序: 1) $\succ$ 是良基的, 即存在项的有限序列 $t_0, \ldots, t_n$, 使得 $t_0 \succ t_1 \succ \ldots \succ t_n$. 2) $\succ$ 是单调的, 即如果 $l \succ r$, 那么对任意项 s, $s[l] \succ s[r]$. 3) $\succ$ 在替换下保持稳定, 即如果 $l \succ r$, 那么对任意替换 θ, $l\theta \succ r\theta$. 4) $\succ$ 具有子项性质, 如果 r 是 l 的子项, 且 $l \neq r$, 那么 $l \succ r$. 定义化简序的直观意义是为了描述表达式之间“谁更加简单”, 从而更适合被推理系统优先处理.

例如当使用字典序作为化简序时, 基于符号字母顺序来定义序关系, 对于项 $f(a,b)$ 和项 $f(a,c)$, 字典序 $a \succ b \succ c$, 因此 $f(a,b) \succ f(a,c)$.

一种常用的化简序是 KBO (Knuth-Bendix ordering), 它基于权重和符号优先级. 预先定义符号的权重和优先级, 先比较权重, 相同时再比较符号优先级来确定项之间的化简序.

例如定义符号优先级 $f \succ g \succ a \succ b \succ c$, 权重 $w(f) = 2$, $w(g) = w(a) = w(b) = w(c) = 1$, 表示令二元函数 f 的权重要大于一元函数和常量. 那么对于项 $f(a,b)$ 和 $g(g(c))$, $w(f(a,b)) = w(f) + w(a) + w(b) = 4$, $w(g(g(c))) = w(g) + w(g) + w(c) = 3$, 于是 $f(a,b) \succ g(g(c))$.

Superposition 演算: 大多数现代一阶定理证明器使用 superposition 演算作为他们的推理系统, 通常将 superposition 推理系统简称为 $\mathbb{S}$up. $\mathbb{S}$up 是合理 (sound) 且反证完备 (refutationally complete) 的. $\mathbb{S}$up 基于反证法来证明公式成立, 称为 saturation-based proof search 过程 (将在第 3.1 节进行介绍).

$\mathbb{S}$up 推理系统的核心规则是 superposition 规则, 该规则最早提出是为了扩展归结 (resolution) 规则, 引入基于等式的重写操作来化简公式. Superposition 名称的来源没有官方公认的说法, 原词字面意思是“叠加”或“重叠”, 可能是由于 superposition 规则一般通过“等式理论”, 将公式项中某一位置的子项替换为另一个项, 然后基于归结原理进行推理. 这一个操作是多个步骤的叠加, 也是将信息重叠到逻辑公式中某一位置的项进行传递.

Superposition 演算的推理规则如下.

𝕊up 推理系统一般包含如下推理规则: superposition 规则 (Sup)、binary resolution 规则 (Bin)、 equality resolution 规则 (ER) 等. 这里我们简单介绍这些规则对应的推理公式.

● Binary resolution 规则: $\dfrac{A\vee C\ \ \neg B\vee D}{(C\vee D)\theta}$ (Bin), 其中 θ 是 A 和 B 的 mgu.

● Equality resolution 规则: $\dfrac{l\neq r\vee C}{C\theta}$ (ER), 其中 θ 是 l 和 r 的 mgu.

● Superposition 规则: $\dfrac{l=r\vee C\ \ t[s]=u\vee D}{(t[r]=u\vee C\vee D)\theta}$ (Sup1), $\dfrac{l=r\vee C\ \ t[s]\neq u\vee D}{(t[r]\neq u\vee C\vee D)\theta}$ (Sup2), $\dfrac{l=r\vee C\ \ L[s]\vee D}{(L[r]\vee C\vee D)\theta}$ (Sup3), 其中 θ 是 l 和 s 的 mgu, $l\theta\succ r\theta$, $t[s]\theta\succ u\theta$ 且 L 不是等式.

● Demodulation 规则: $\dfrac{l=r\ \ \cancel{C[l\theta]\vee D}}{C[r\theta]\vee D}$ (Dem), 其中 $l\theta\succ r\theta$ 且 $C[l\theta]\vee D\succ l\theta=r\theta$. 这一条规则是化简规则, 是 superposition 规则的一个特例.

这里我们给出一些直观解释来理解上述规则.

Binary resolution 规则, 即归结规则 (或翻译为消解规则), 是一种基本的推理规则, 表示若 A 和 B 存在合一子替换, 由于 $A\theta=B\theta$, 那么可以将 $A\theta$ 和 $(\neg B)\theta$ 消解, 于是 $(C\vee D)\theta$ 必然成立.

Equality resolution 规则类似归结规则, 但它引入了等式符号. 该规则比较简单: 如果 $l\neq r\vee C$ 成立, 那么对任意模型, 要么 C 成立, 要么 $l\neq r$ 成立. 如果 $l\neq r$ 成立, 那么 $l\theta\neq r\theta$ 成立, 若 l 和 r 存在 mgu 为 θ, 那么 $l\theta=r\theta$, 出现矛盾. 于是此时必然有 $C\theta$ 成立.

Superposition 规则以 Sup1 为例, 它通过等式 $l=r$ 来做重写. 等式 $t[s]=u$ 表示某个位置的子项为 s 的项 t, 有 $t=u$. 如果 s 和 l 是可合一的, 存在 mgu 为 θ, 即 $l\theta=s\theta$. 该规则表示如下推理: 对前提子句的任意模型, 如果上下文 C 和 D 都是 false, 那么必然有 $l=r$ 和 $t[s]=u$ 同时为 true. 由于 $s\theta=l\theta$, 那么将 s 用 θ 替换后, 基于等式关系有 $s\theta=l\theta=r\theta$. 于是 $t[s]\theta=t[r]\theta$, 从而 $t[r]\theta=u\theta$, 即 $(t[r]=u)\theta$ 在 C 和 D 都为 false 时必然为 true. 这一个规则可以被视作一种有条件重写 (conditional rewriting), 即假设 C 和 D 均为 false 时, t 的子项 s 可被 $l=r$ 重写. Superposition 规则可以认为是提供了一种应用等式关系和变量替换来对原公式中的子句进行化简和消去的方法.

基于推理系统对公式进行证明需要一个合适的算法来进行反证的过程, 即如何在前提条件构成的搜索空间中组织对空子句的搜索. 进行证明推导一般使用 saturation-based proof search 推理框架, 我们将在第 3.1 节中进行介绍.

1.5 在 SMT 中表示带有递归定义函数的问题

对于带有递归定义函数的程序验证问题, 传统方法是通过程序分析技术来处理递归定义, 生成尽可能简单的 SMT 公式 (一般为无量词不包含递归函数定义的简单公式) 交给底层 SMT 求解器进行求解 (在本文第 4.5 节将对这类方法进行介绍). 近些年来, 研究者开始关注直接在求解器中处理带有递归结构的 SMT 公式.

基于 SMTLIB 标准语法对递归定义函数进行表示, 通常是将函数体表示为全称量词和未解释函数组成的公理断言. 从 2016 年开始, SMTLIB 标准语法支持使用 define-fun-rec 直接定义递归函数, 下面分别展示使用公理断言和 define-fun-rec 在 SMT 公式中表示递归定义函数的例子, 并假设待验证性质为对任意 $list$ 类型的参数 x, 其长度大于等于 0, 即 $\forall x:list.\ len(x)\geqslant 0$. 在本文的例子中我们两种写法都会有所涉及.

使用公理表示递归定义函数: 例如我们在 SMT 的 ADT 理论中定义 $list$ 类型: declare-datatypes $list=nil|cons(head:int,tail:list)$. 然后在 SMT 中定义 $list$ 类型的长度函数 $len:list\rightarrow int$, 使用 SMTLIB 标准格式进行公理定义如下.

```
1 (declare-fun len(list))
2 (assert (= (len nil) 0))
3 (assert ((forall ((x Int) (y Int)) (= (len(cons x y)) (+ (len y) 1)))))
4 (assert (not ((forall (x list) (>= (len x) 0)))))
```

使用 define-fun-rec 表示递归定义函数: 同样定义 *list* 类型的长度函数, 可以表示如下.

```
1 (declare-fun-rec len
2    ((x list)) Int
3       (match x
4          ((nil 0)
5          ((cons x y) (+ (len y) 1))
6       )))
7 (assert (not ((forall (x list) (>= (len x) 0)))))
```

以上两种表示方法均为分别从递归函数的基本情况 (base case) 和递归情况 (recursive case) 进行构造. 表示令空列表 *nil* 的长度 *len* 为 0, 对任意列表 L, 当 L 由类型为 *Int* 的 x 和类型为 *list* 的 y 构造时, 即 $L = cons\ x\ y$ 时, L 的长度为 y 加 1. 并对待验证公式取反, 当 SMT 求解返回不可满足时, 则表示原性质必然成立.

SMT 求解器对于 define-fun-rec 的函数定义会采取一些特定的策略进行求解, 如 Z3 对于使用 define-fun 的函数定义更倾向于“eager”的展开策略, 将所有对应函数的出现用函数体进行代换; 而对于 define-fun-rec 则更倾向先作为未解释函数进行保留. 如 cvc5 实现了针对递归定义函数的求解优化选项“--fmf-fun”, 对于 define-fun-rec 的递归定义函数生效, 该技术主要用于找到可满足模型, 下文将进行详细介绍.

对于带递归定义的 SMT 公式求解, 待验证断言通常用带全称量词的 SMT 公式表示, 例如若希望验证对任意 *list* 类型, 其长度均大于等于 0, 则待验证断言表示为 $\psi := \forall x.\ len(x) \geqslant 0$. 其中 x 类型为 *list*. 主流工作分为两类: 一类是基于 SMT 求解器的验证工具, 需要先将待验证公式取反, 然后使用 SMT 求解器求解, 若返回 SAT, 则表明存在一组赋值使该断言不成立, 即存在一组反例; 若返回 UNSAT, 则证明原公式成立. 另一类也是基于自动定理证明系统, 待验证断言作为结论, 若能够在证明系统中推理出该结论, 则成功证明.

由于递归定义的结构, 待验证公式的验证往往依赖于归纳推理等技术, 因此在 SMT 求解器或定理证明系统中引入归纳推理能力是该领域的研究重点, 目前的主要工作可以分为两类: 一类是基于 SMT 求解的 DPLL(T) 框架, 在全称量词实例化的过程中对公式进行归纳增强, 并结合子目标生成技术提高求解效率; 另一类是在自动定理证明器的证明系统中引入归纳推理规则, 基于自动定理证明系统对 SMT 公式进行验证. 下面我们分别对两类工作进行介绍.

2 SMT 求解器递归定义求解技术

基于 SMT 求解算法验证带递归定义公式的技术包括验证技术 (即返回 UNSAT 求解进行证明) 和找错技术 (即找到可满足解返回 SAT). 基于 SMT 求解的框架, cvc5 在求解带递归定义函数的 SMT 问题时可以利用现代求解器在代数数据类型和整数等背景理论下的高效求解能力. 本节将先介绍用于递归函数问题返回 UNSAT 的归纳定义增强和引理生成技术, 然后简要介绍和讨论用于递归函数问题返回 SAT 的模型寻找方法.

2.1 DPLL(T) 判定算法

DPLL(T) 是目前 SMT 求解的主流判定算法. 它由用于推理特定理论背景可满足性的理论求解器和基于 DPLL 算法来高效推理命题逻辑可满足性的 SAT 求解器组成. 由于求解时先将 SMT 公式视作 SAT 公式进行求解, “按需”使用理论求解器, 因此该方法被称为 lazy 方法.

DPLL(T) 伪代码如算法 1. 其中输入为公式 φ_{in}. *quant_elim* 为量词消去函数. *get_model* 表示基于 DPLL 算法的 SAT 求解器, 它接受一个命题逻辑公式 F 作为输入, 如果 F 不可满足则返回空 (none); 如果 F 可满足则返回一个可满足文字的合取子句 A, 使得 $A \vDash F$. $check\_sat_T$ 表示背景理论 T 的求解器, 它接受一个符号表 Σ 中文字的合取子句 ψ 作为输入, 返回可满足或 ψ 中不可满足文字的合取子句 μ. 算法中上标为 c, 如第 8 行 A^c, 和上标为 a, 如第 2 行 φ^a, 分别表示具体化 (concretization) 和抽象化 (abstraction) 函数, 其中抽象化函数将一个带背景理论的无量词 SMT 公式 φ 映射到命题逻辑公式, 具体化表示抽象化函数的逆函数, 即 $(\varphi^a)^c = \varphi$. 算法原理是先将原公式做量词消

去得到无量词公式, 然后将其抽象为命题逻辑公式, 判断其可满足性, 如果不可满足则原公式不可满足, 如果可满足则通过理论求解器检查背景理论是否可满足. 由于现代 SAT 求解算法的发展, 许多优化技术可以极大提升 SAT 判定的效率, 因此 DPLL(T) 算法优先进行 SAT 判定, 必要时才进行难度更高的背景理论判定.

算法 1. DPLL(T) 算法基本框架.

输入: 一个背景理论 T 的公式 φ_{in}, 符号表为 Σ;
输出: 当 φ_{in} 在背景理论 T 下可满足, 则输出 SAT, 否则输出 UNSAT.

```
1. φ := quant_elim(φ_in)
2. F := φ^a
3. while true do
4.     A := get_model(F)
5.     if A == none then
6.         return UNSAT
7.     else
8.         μ := check_sat_T(A^c)
9.         if μ == SAT then
10.            return SAT
11.        else
12.            F := F ∧ ¬μ^a
13.        end
14.    end
15. end
```

DPLL 算法: 这里我们简单介绍 SAT 判定的 DPLL 算法. DPLL 算法得名于该算法发明人: Davis-Putnam-Logemann-Loveland. 它的基本思路可以理解为通过不断地尝试给变量赋值, 在发现冲突 (即某一个赋值使得公式出现矛盾) 时回溯, 逐步缩小搜索空间, 最终找到满足布尔公式的解或证明原公式不可满足. 20 世纪 90 年代开始, 在 DPLL 框架上涌现出许多优化技术, 极大提升了算法效率. 最显著一个优化是被称为冲突子句学习 (conflict driven clause learning, CDCL) 的技术. 它相比 DPLL 框架最大的区别在于对冲突分析和回溯的技术. 当它发现冲突时, 不会简单回溯尝试其他赋值, 而是从冲突中学习一个新的子句, 将该子句添加到原问题公式中, 从而避免未来出现类似冲突, 减少搜索空间. 另外 CDCL 算法优化了原来根据决策变量顺序回溯的机制, 而是根据冲突分析进行计算, 基于学习到的子句直接跳到导致冲突的决策层.

2.2 代数数据类型理论的判定算法

本节主要介绍无量词代数数据类型理论的判定算法. 公式只含有代数数据类型中的构造子、选择子等符号. 即本节研究的问题中待求解公式的形式如例 2 中的公式 (1).

目前主流 SMT 求解器对无量词 ADT 公式的基本求解框架是先对 ADT 公式的结构进行展平, 逐步猜测变量对应构造子, 然后构建同余闭包进行判定. 在实现中一般采用一些启发式优化策略, 在猜测对应构造子、消去选择子等基本结构这些过程中进行剪枝, 从而缩减由于复杂结构带来的搜索空间爆炸问题. 但面对公式规模较大且包含多种数据类型的情况时求解效率依然有限. 下面具体介绍相关算法.

对于无量词代数数据类型理论公式的判定, 最早由 Oppen 等人[2,29]在 1980 年提出了一种线性时间判定算法, 通过代数数据类型公式构造有向图, 计算图中节点的同余闭包 (congruence closure) 从而得到公式中相关项 (term) 的等价关系来进行判定, 但该算法只适用于单个归纳数据类型且带有单个构造子 (constructor) 的递归定义数据结构.

对于更一般的问题, 如允许互递归数据类型以及数据类型中包含多个构造子的情况, 其判定问题被证明是

NP 完全的[29], 对于该问题, Barrett 等人[3] 于 2007 年在 Oppen 判定算法的基础上提出了一种有效的判定算法, 该方法首先展平公式子句, 对展平得到的子项, 当其数据类型定义包含多个构造子的情况, 猜测其对应的顶层构造子, 逐步地构建约束变量值, 当发现不一致 (inconsistency) 时则回溯, 然后尝试不同的构造子直到判定约束满足或无法再选择, 且为了进一步提升计算效率, 算法中引入了启发式策略来优化顶层构造子的猜测来进行剪枝. 该方法目前已经成为 SMT 求解器对于代数数据类型理论的基本判定方法.

例 3: 我们用一个简单的例子来解释代数数据类型理论求解基本算法的思路. 假设在 *list* 类型上有如下公理:

$$\begin{cases}\forall x,y.\ head(cons(x,y))=x\\ \forall x,y.\ tail(cons(x,y))=y\end{cases},$$

其中, $cons$ 是构造子, $head$ 和 $tail$ 是选择子. 待求解公式 φ 是 $l=cons(u,v)\wedge cons(head(l),tail(l))\neq l$. 那么一个判定过程是基于上述公理构造同余闭包来逐步判断项的等价关系. 首先由 $l=cons(u,v)$ 和公理, 我们有等价类 $\{cons(u,v),l\}$、$\{head(l),u\}$、$\{tail(l),v\}$. 于是由同余关系, 应该有 $cons(u,v)$ 和 $cons(head(l),tail(l))$ 在同一等价类中, 即 $cons(u,v)=cons(head(l),tail(l))$, 这与 $cons(head(l),tail(l))\neq l$ 矛盾. 于是待求解公式为不可满足的 (UNSAT) .

Reynolds 等人[4]对 Barrett 等人[3]的方法进一步扩展, 引入一种统一的判定算法, 用于同时支持数据类型 (datatype) 理论和对偶数据类型 (codatatype) 理论求解.

由于 DPLL(T) 框架在求解 ADT 问题时, 算法中学习到的引理子句可能存在选择子 (selector), 但每个选择子只与一个构造子相关联, 这将造成学习到的引理子句通用性较差, 针对这一问题, Reynolds 等人[5]提出共享选择子 (shared selector) 理论用于减少判定过程中项的计算数量, 从而加速求解时间, 但实验表明该方法的适用性不够强, 尽管可以显著提升语义引导合成 (SyGus) 问题相关的求解效率, 但对于更一般且公式规模增大的测例集, 该方法提升效果有限.

上述这些经典的判定方法主要基于 SMT 求解 lazy 方法, 近年来, 有一些研究者考虑通过 eager 方法来求解代数数据类型公式[6,7]. Hojjat 等人[6]提出将代数数据类型公式归约到等价的未解释函数和线性算术理论公式的方法, 并实现在 Princess 求解器中; 类似的, Shah 等人[7]将代数数据类型公式归约到等价的未解释函数理论的公式进行求解, 但使用与 Hojjat 等人[6]不同的归约技术去避免引入线性算术理论造成的求解困难, 他们的方法相对传统方法在实验效果上具有一些提升, 但对于包含互递归定义以及多种类型定义且长度较大的复杂公式, 该方法与现有 state-of-the-art 的 SMT 求解器 Z3、cvc5 和 Princess 均表现出求解困难, 表明现有的代数数据类型理论求解方法在可扩展性和求解效率上依然存在较大提升空间[7].

2.3 归纳推理增强技术

下面我们介绍基于引理生成的归纳推理增强技术. 带有递归定义函数的问题求解效率十分依赖于在求解方法中引入的归纳推理模式和引理生成技术的效率. 本节首先介绍在量词消去过程中引入归纳推理模式的方法. 第 2.4 节我们将介绍辅助证明的引理生成技术.

为了便于理解, 例 4 中我们用更接近数学语言的写法替代 SMT 语法, 表示上文中对 *list* 定义的长度函数.

例 4: 假设长度函数 $len: list\to Int$:

$$\begin{cases} len(nil)=0 & (A_1)\\ \forall x,y.\ len(cons(x,y))=1+len(y) & (A_2)\end{cases},$$

要证明断言 $\psi:=\forall x.\ len(x)\geqslant 0$.

为证明断言成立, 我们求解 $F:=\{A_1,A_2,\neg\psi\}$ 的可满足性, 若 F 可满足, 表示我们找到一个使断言 ψ 不成立的反例, 若 F 不可满足, 表示我们证明断言 ψ 成立.

SMT 求解器处理上述问题首先会调用量词消去模块, 对于全称量词, 一般基于 instantiation (实例化) 技术; 对于存在量词, 则一般基于 Skolemization (斯科伦化) 技术[30]. 由于待验证断言是一个否定的全称量词公式 (SMT 求解器中一般基于德摩根律将其转换为存在量词公式), SMT 求解器首先会尝试使用 斯科伦化技术对其进行量词消去: 首先可推出引理 $(\forall x.\ P(x))\vee\neg P(k)$, 这里 k 是一个常数 (这一引理的成立直观上可以理解为等价于

$\exists\neg P(x)\to\neg P(k)$, 这里不展开介绍). 对 $\neg\forall x.\ P(x)$ 进行斯科伦化, 得到 $\neg P(k)$, 称 k 为斯科伦 (Skolem) 常数. 然后 $\neg P(k)$ 将被加入 F 中, 为便于讨论, 我们假设这里 $\neg P(k)$ 已经是一个无量词公式. 于是 SMT 求解器将尝试调用内置决策过程求解 $\neg P(k)$ 的可满足性. 而在例 4 中我们将看到在没有引入归纳推理能力时, SMT 求解器无法完成对该问题中 $\neg P(k)$ 的求解.

一个失败的推理过程: 对 ψ 进行斯科伦化之后得到公式集合 $F:=\{A_1,A_2,\neg\psi,\neg len(k)\geqslant 0\}$. 由 $\{A_1,\neg len(k)\geqslant 0\}$, SMT 求解器找到一个模型 $k=cons(h,t)$ 且 $len(k)=-1$. 这里表示 k 是一个由 h 与 t 构造的类型为 $list$ 的常数, 且长度为 -1. 将这一模型代入 A_2 进行实例化, 得到 $len(cons(h,t))=1+len(t)$. 于是 $len(t)=-2$. 再一次进行实例化, 将会得到存在 hh 和 tt, 使得 $t=cons(hh,tt)$, 且 $len(t)=1+len(tt)$. 从而 $len(tt)=-3$. 这一过程将会无限循环进行, 使求解失败. 其原因在于当带量词的代数数据类型理论公式为 false 时, SMT 进行判定的公理模型是不标准模型 (non-standard model), 即 SMT 求解器中包含的相关理论公理不足以满足我们对形如例 4 的问题进行证明.

归纳推理增强: 在 Reynolds 等人[31]通过对斯科伦化过程中引入归纳推理增强来解决上述问题. 假设当前类型的公式项存在一个良基序 (well-founded ordering) R, 那么存在 k, 使如下公式成立:

$$(\forall x.\ P(x))\vee(\neg P(k)\wedge\forall x.\ (R(x,k)\to P(x))) \tag{2}$$

此时称 $\forall x.\ R((x,k)\to P(x))$ 是 $\neg P(k)$ 基于 R 的归纳增强 (inductive strengthening).

这一归纳增强方法可以视作将一种归纳模式 (induction schema) 引入 SMT 求解过程. 公式成立的具体证明过程可参考文献 [31] 第 2 节 Remark 1.

从直观上看, 若对任意 x, $P(x)$ 成立, 则公式 (2) 成立; 否则存在 y 使得 $P(y)$ 不成立, 即 $\neg P(y)$. 我们假设所有这样的 y 构成集合 S, 则 S 不为空集, 对 S 中任意的一个元素 y_0, 考虑从 y_0 出发的极大良基关系序列 $y_0,y_1,\ldots\in S$, 其中任意下标 i, 满足 $R(y_{i+1},y_i)$. 那么由良基关系, 该序列必然是有限的, 令其在某个 y_n 终止. 那么令 k 为 y_n, 满足 $\neg P(k)$, 且由于 y_n 是序列中最后一个元素, 满足 $\forall x.\ (R(x,k)\to P(x))$.

通常在 SMT 求解中根据待求解公式的背景理论, 有两种典型的良基关系 R 被使用: 在代数数据类型理论中, 对于代数数据类型的项 s 与 t, $R(s,t)$ 当且仅当 s 是 t 的子项. 这对应使用结构归纳法的求解模式. 在整数理论中, 对于整数 s 和 t, 则通常 $R(s,t)$ 当且仅当 $0\leqslant s\leqslant t$. 这对应使用数学归纳法的求解模式. 为了简化求解, 在实际的 SMT 求解器 cvc5 中, 实现了弱归纳法的求解模式: 对于代数数据类型的项 s 和 t, $R(s,t)$ 当且仅当 s 是 t 的直接子项, 例如 $list$ 类型中的 $tail$ 项与原项. 对于整数 s 和 t, 则 $R(s,t)$ 当且仅当 $0\leqslant s=t-1$.

成功求解例 4: 引入归纳推理增强之后, 例 4 可以被成功求解. 此时对于 ψ, 将产生公式 $\neg len(k)\geqslant 0\wedge\forall y.\ (y=tail(k)\to len(y)\geqslant 0)$. 该公式化简为 $len(k)<0\wedge len(tail(k))\geqslant 0$. 而由 A_2 可推知 $len(tail(k))<len(k)$. 于是矛盾, 求解器返回 UNSAT, 原命题 ψ 得证.

2.4 辅助证明引理生成技术

我们分别介绍目前 SMT 求解器 cvc5 中所实现的证明引理生成技术和近年来提出的其他引理生成技术.

2.4.1 cvc5 求解器引理生成技术

尽管在 SMT 求解的量词消去过程中引入归纳推理增强可以成功解决例 4, 但实际在许多情况下, 只引入归纳推理增强依然不足以完成公式求解, 还需要结合启发式的子目标生成技术自动在公理集合中引入“中间引理”或“子目标”来完成求解. 这一过程类似于程序验证问题中为循环程序引入循环不变式, 或补充前置或后置条件, 可以视作一种对原已有条件的加强. 如何更好地自动生成引理以辅助 SMT 公式的求解是领域内一个重点研究方向, 在第 3 节将对现有的重点方法进行梳理和介绍. 这里我们介绍 Reynolds 等人[31]的子目标生成方法, 该方法实现在 cvc5 求解器中. 通过自动生成子目标来证明待验证公式 ψ 的基本思路如下: 1) 首先确定相关 (relevant) 子目标 φ_1. 2) 证明 φ_1 成立. 3) 将 φ_1 放入前提公式集合, 在 φ_1 成立的前提下证明 ψ.

实际的 SMT 求解过程在找到一个相关子目标 $\forall x.\ f(x)=g(x)$ 之后, 会在公式集合中加入分离引理 (splitting lemma): $\neg\forall x.\ f(x)=g(x)\vee\forall x.\ f(x)=g(x)$. 然后分别考虑 $\neg\forall x.\ f(x)=g(x)$ 和 $\forall x.\ f(x)=g(x)$ 两种可能分支的情况. 在进行子目标生成时, Reynolds 等人[31]的基本方法是先基于已知符号枚举可能的公式, 然后通过启发式过滤方法

来寻找相关子目标公式, 此时枚举的候选公式中剩下的公式称为被过滤 (filtered) 公式. 下面分别介绍枚举的基本方法和过滤候选公式的启发式技术.

枚举子目标公式: 首先定义公式项的 *size* 为项中函数应用次数加上重复变量的数量. 例如对于 $f(g(x,y))$, 函数 g 作用于参数 x 与 y, f 作用于 $g(x,y)$, 没有重复变量, 则 $f(g(x,y))$ 的 *size* 为 2. 而 $g(x,f(x))$, 出现函数 f 和 g, 且 x 重复出现, 于是 *size* 为 3. 令形如 $\forall \bar{x}.\ f(\bar{x}) = g(\bar{x})$ 的 *size* 是 $\max(size(f(\bar{x})), size(g(\bar{x})))$. 例如 $\varphi_1 = \forall x,y.\ f(g(x,y)) = h(f(x), f(y))$, 左边的 *size* 为 2, 右边 $f(x)$ 和 $f(y)$ 两次函数应用, 加上函数 h, *size* 为 3, 于是 φ_1 的 *size* 为 3. 从 *size* 为 0 开始, 对 *size* 为 n 枚举所有可能的子目标集合 S_n, 称其为候选子目标集合. 对每个 n, 通过启发式技术确定一个子集 $S_n^R \subseteq S_n$, 称该子集为相关子集, 剩下的公式集合称为被过滤子集. 随着 n 增长不断构造相关子集作为子目标引理, 用于辅助归纳推理, 直到达到某个固定的 n 的上界 (实际中一般使用 3). 下面介绍 Reynolds 等人[31]所使用的过滤技术.

过滤候选子目标: 主要有 3 种启发式过滤技术.

1) 基于激活断言 (active conjecture) 过滤. 对于一个项 t 如果对当前的文字集合 M 和原问题符号表 Σ, 可以推出存在某些 Σ 中的项 s, 使得 $t = s$, 则称 t 是非激活的 (inactive), 否则为激活的 (active). 在 M 中出现的形如 $f(t_1,\ldots,t_n)$ 的项, 若其中至少有一个项 t_i 是激活的, 则称这个 f 项为 M 中基础相关 (ground-relevant) 的项. 如果一个 Σ 中的项 t 能够被泛化为一个基础相关项 s, 则称 t 是相关项, 这里的泛化指在 M 中可推出 $(t = s)\sigma$ 成立, σ 是某个将 t 中自由变量映射为基项的替换. 最后在枚举生成候选子目标的过程中, 只保留相关的项. 进行这一步过滤的直观是对原命题进行斯科伦化后会在原问题的符号表 Σ 中引入斯科伦常数 k, 构成扩展符号表 Σ'. 此时原问题的公理和 SMT 对应的代数数据类型或整数背景理论无法推导扩展后的符号表 Σ' 所构成的公式. 基于这一观察, 在进行过滤时应该尽可能生成仅可泛化到扩展符号表 Σ' 项的候选子目标, 尤其是那些无法在当前上下文中推出等价于原符号表 Σ 项的子目标公式.

例 5: 考虑代数数据类型和等式的混合理论 T. 定义了数据类型 *nat* 和 *list*. 并假设符号表 Σ 中包含函数 *plus*、*app*、*rev* 和 *sum* 分别表示自然数的加法、对列表添加元素、列表的反转以及将列表的元素求和. 令 $\mathcal{A}$ 是这些函数上的公理, 包括:

$$\begin{cases} sum(nil) = O \\ \forall x,y.\ sum(cons(x,y)) = plus(x, sum(y)) \end{cases},$$

要证明断言 $\psi := \forall x.\ sum(rev(x)) = sum(x)$.

那么在例 5 中, 对 $\neg\psi$ 进行斯科伦化得到 $\neg sum(rev(k)) = k$, k 为斯科伦常数. 要证明例 5 实际上需要生成合适的子目标公式, 这里我们暂时不介绍整个具体的求解过程, 而是以本例介绍过滤技术的相关概念. 假设在求解的某个阶段, 得到当前上下文子句集合为 $M = \{sum(k) = O, sum(rev(k)) = s(O), rev(k) = nil\}$. 此时在 M 中没有形如 $k = x$ 的公式, k 是激活的项, 那么 $sum(k)$ 是基础相关项, 由于 $sum(x)$ 可以泛化为 $sum(k)$, 则 $sum(x)$ 是相关项. 在候选子目标中包括这一项的公式将会保留. 而由于 M 中有 $rev(k) = nil$ 这一公式, 那么 $sum(rev(k))$ 不是基础相关项, 于是 $sum(rev(x))$ 不是相关项. 那么形如 $\forall x.\ sum(rev(x)) = t$, 其中 t 表示任意项, 这样的候选子目标将会被过滤. 直观上我们可以理解为这些形如被过滤的子目标的公式可以轻易地在判定过程中由当前上下文推出, 而无需作为专门的子目标公式引入.

2) 基于规范性 (canonicity) 过滤. 这一技术的直观是将 SMT 求解器中对等式理论和代数数据类型理论基于同余闭包对公式中的项划分等价类进行推导的方法进行扩展. 通过构造等价类的方式来过滤同一等价类中多余的项. 假设当前上下文 M 中的一个由非基础 (non-ground) 的 Σ 项组成的等式集合 U. 维护一个 U 上的同余闭包 U^*, 其中的每一个等价类 $\{t_1,\ldots,t_n\}$, 对任意 $i,j \in \{1,\ldots,n\}$ 有 $\forall[FV(t_i) \cup FV(t_j)].\ t_i = t_j$. 在每个等价类中选择一个具有最小 *size* 的项, 令其为代表项 (representative term). 称 U^* 中的一个项是规范的 (canonical) 当且仅当它是其中一个等价类的代表项, 称其为不规范 (non-canonical) 当且仅当它在 U^* 中且不是代表项. 那么在枚举候选子目标时, 对于那些包含至少一个不规范 (non-canonical) 子项的子目标, 将会被过滤掉.

例 6: 假设上下文 $M = \{\forall x.\ app(x,nil) = x\}$. 可以构造等式集合 $U = \{app(x,nil) = x\}$, 于是同余闭包 U^* 包含两个等价类 $\{x, app(x,nil)\}$ 和 $\{nil\}$. 假设对某一个候选子目标 $\varphi := \forall x.\ rev(app(rev(x),nil)) = x$. 引入 φ 中的所有子项, 同

余闭包 U^* 会增加等价类 $\{rev(x)\}$, $\{app(rev(x),nil)\}$ 和 $\{rev(app(rev(x),nil))\}$. 由于 $app(rev(x),nil)$ 可由 $app(x,nil)$ 进行替换 $\sigma := \{x \mapsto rev(x)\}$ 得到, 且 $app(x,nil) = x$. 可将 $\{rev(x)\}$ 与 $\{app(rev(x),nil)\}$ 进行合并, 合并后代表项是 $rev(x)$. 于是 $app(rev(x),nil)$ 成为不规范 (non-canonical) 项, 从而子目标 φ 在 M 中可被过滤. 事实上由于等价类关系, φ 等价于 $\psi = \forall x.\ rev(rev(x)) = x$, 因此 φ 被认为是冗余的候选子公式, 在生成中更倾向于保留 ψ 来替代 φ.

3) 基于基础事实 (ground fact) 过滤. 这一技术的直观思路是通过找到在当前上下文中可以推出的反例实例来确定对应的候选子公式是否成立. 即对于 M 中候选子公式 $\forall \bar{x}.\ f(\bar{x}) = g(\bar{x})$ 是否成立, 可通过考虑它的实例是否为假来判定, 如果 M 推出 $\neg(f(\bar{x}) = g(\bar{x}))\sigma$ 对某个 $\bar{x}$ 上将自由变量替换为基项的替换 σ (称其为基础替换 grounding-substitution, 替换后的实例称为基础实例 ground-instance) 成立, 那么显然有 $\forall \bar{x}.\ f(\bar{x}) = g(\bar{x})$ 在 M 中不成立. 又由于对公式 φ, 即使它在当前上下文中有反例, 但当上下文公式更新后, 可能并不包含这一反例, 因此对于某个候选子公式 φ, 当它的任意基础替换得到的实例 $(f(\bar{x}) = g(\bar{x}))\sigma$ 都不能在 M 中推出, 或只有少于某个设定的常数数量的实例可被推出, 也会将 φ 进行过滤.

例 7: 假设当前上下文 $M = \{sum(cons(O,k)) = plus(O,sum(k)), plus(O,sum(k)) = sum(k)\}$. 对于候选子公式 $\varphi := \forall x.\ sum(x) = s(O)$. 由于 φ 的任意基础替换得到的实例都不成立. 即使 φ 没有在 M 中为 false 的基础实例, 但由于 φ 的任意基础实例都不会被推出, 于是 φ 将被过滤.

上述 3 种过滤技术被实现在求解器 cvc5 中, 用于辅助归纳推理增强技术, 通过 cvc5 的选项“--quant-ind”调用, 可以较为有效地证明形式相对简单的递归函数问题. 但由于用于归纳增强的归纳模式较为简单, 这一方法无法较好地处理形式更复杂的递归函数问题, 例如归纳基础需要考虑变量而不是常数时以及归纳定义不是简单的直接子项时, 均无法有效完成证明. 近年来有许多工作基于 superposition 推理系统, 在自动定理证明器的推导中引入更复杂的归纳模式, 使其可以处理递归定义更复杂的问题; 另外也有许多工作专注于提出各种子目标生成方法来更好地在求解中生成有用的子公式从而提高命题公式求解能力. 这些技术将在后面的章节进行介绍.

2.4.2 其他子目标生成技术

在第 2.3 和 2.4.1 节中我们所介绍的技术是目前实现在主流 SMT 工具 cvc5 中的归纳推理和引理生成技术. 实际上还有许多方法/工具关注递归定义函数问题的求解, 包括 Scala 的程序验证工具 Leon/Stainless[32–34]、交互式定理证明器 Coq, Isabelle/HOL[35]、半自动定理证明工具 ACL2[36]、自动定理证明工具 Zeno 等. 这些工具所支持的输入形式一般是基于归纳类型构建的纯函数式语言, 如 haskell 等, 对这种语言的程序进行推理天然依赖归纳推理技术. 其中有些工具依赖用户手动提供归纳推理过程, 如 Leon/Stainless 和 Coq 等, 而有些工具如 Zeno[37]则实现了自动的归纳推理. 由于这些技术主要属于程序分析/验证的层次, 且有些基于重写的证明系统与能进行 SMT 求解的求解/推理系统具有较大区别, 为了更明确本文关注的主要问题, 不将主题范围过于扩大, 本文将不对上述技术做详细介绍. 但这些工作中有些为求解递归定义函数的问题提出了比较通用的自动子目标生成技术, 可以在原理上为带递归定义的 SMT 自动化证明提供一些启发思路, 因此下面我们简要对相关技术进行介绍和梳理.

基于泛化 (generalization) 生成辅助引理: 这一方法通常在基于推理系统的定理证明器中使用. 对于一个子目标, 当无法直接证明时, 考虑引入一个假设, 或尝试将目标进行泛化. 对一个公式进行泛化, 即将原公式中的某些固定子项用变量替换, 实际使用中需要通过一些启发式技术, 例如选取最小公共子项、反例检查等, 来选取合适的用于泛化的子项, 并尽量避免过泛化 (over-generalization) 的情况. 如下例是一个使用泛化生成辅助引理的例子.

例 8: 考虑一个插入排序函数 *insertsort*. 它输入一个 *list* 类型:

$$\begin{cases} sorted(nil) = \top \\ \forall x.\ sorted(x) = \top \\ \forall x,y,ys.\ sorted(x :: y :: ys) = x \leqslant y \wedge sorted(y :: ys) \\ \forall x.\ insert(x,nil) = x :: nil \\ \forall x,y,ys.\ insert(x,y :: ys) = ite((x \leqslant y),(x :: y :: ys),(y :: insert(x,ys))) \\ insertsort(nil) = nil \\ \forall x,xs.\ insertsort(x :: xs) = insert(x,insertsort(xs)) \end{cases},$$

其中, $ite(a,b_1,b_2)$ 表示若 a 为真, 则 b_1, 否则为 b_2. 假设我们要证明 $sorted(insertsort(xs))$ 成立, 即插入排序后的 *list* 被正确排序, 通过对 xs 进行结构归纳可以得到基础步骤和归纳步骤:

$$\begin{cases} sorted(insertsort(nil)) = \top \\ sorted(insertsort(xs)) \rightarrow sorted(insertsort(x :: xs)) \end{cases}.$$

对上述的归纳步骤应用归纳定义重写可得到:

$$sorted(insertsort(xs)) \rightarrow sorted(insert(x, insertsort(xs))),$$

注意到这个公式两边有相同的子项 $insertsort(xs)$. 于是可以应用泛化技术生成辅助子公式:

$$sorted(ys) \rightarrow sorted(insert(x, ys)),$$

这一公式具有更简单的形式, 将会更易于定理证明工具进行求解.

这一技术在定理证明工具 ACL2[36]、Zeno[37]中均有所实现. 本文第 3 节中提到在 superposition 推理系统中引入泛化归纳技术也可以视作是这一类子公式生成技术. 这一技术的优势在于当合适的泛化方法被使用时, 可以快速高效地找到正确的子公式. 但这一方法容易产生过度泛化的问题, 而且在证明系统中何时应用泛化没有一个统一的原则, 只能根据具体的问题和推理系统选择启发式方法.

基于失败证明生成辅助引理: 主要思路是当证明失败时, 通过分析造成失败的原因然后尝试生成对应的辅助引理. 这一方法在早期基于重写规则的定理证明系统受到关注, 发展出了 rippling[38]方法, 在自动定理证明系统 RRL[39]中也实现了类似思路的方法用于自动生成归纳引理. 另外 Murali 等人[40]对于带有最小不动点定义的一阶逻辑问题, 提出了一个利用一阶逻辑推理过程中计算出来的反例模型来引导生成归纳引理的方法. 这一类型技术的优势在于应用时机相比基于泛化的方法更加明确, 且可以找到一些泛化技术无法找到的引理项. 缺点在于该方法的使用可能依赖启发式方法, 且可能生成过于复杂的公式导致搜索空间增大, 反而造成求解效率下降的问题[41,42].

基于理论探索 (theory exploration) 生成引理: 该方法的思路是从可用的符号, 包括函数和代数数据类型构造子等, 来构建一个候选引理集合, 在构造时主要通过枚举或公式模板等方法, 然后通过反例检查, 或者启发式的过滤方法来从候选集合中生成相关的引理用于证明原命题, 然后基于这个引理集合来证明待验证公式[43,44]. 这一方法的应用和发展比较广泛, 我们在第 2.4.1 节中提到在 SMT 求解器中进行子目标生成的方法则可以归于这一类, 其中通过公式项的 *size* 来枚举候选引理. 而 Yang 等人[45]则是通过语法定义可能的公式模板来生成引理. Sivaraman 等人[46]提出将引理生成问题转换为一个数据驱动 (data-driven) 的程序综合问题. 并提出了一些用于过滤子引理和评估候选引理的技术. 这一方法的效率决定于对候选引理集合的评估方式, 即如何找到与证明目标最相关的引理, 缺点在于难以生成比较复杂或规模较大的引理, 因而能处理的问题规模受到限制, 可扩展性 (scalability) 可能受限.

归纳友好的引理生成: 针对现有方法在生成辅助引理时过于依赖基于启发式方法的枚举等技术, 推理过程中生成大量无用引理而造成求解时间浪费的问题, Sun 等人[47]提出了一种“定向引理生成 (directed lemma synthesis)”方法. 该方法首先定义两种称为“归纳友好 (induction-friendly)”的公式形式, 在这种形式的公式上可以高效地应用归纳假设进行归纳推理. 然后问题则转变为如何将待验证目标转换为归纳友好形式的公式. 这项工作中提出了两种技术通过生成和应用辅助引理来进行转换, 其中生成合适引理的主要思路是将引理生成问题转为一个程序合成问题, 其中待合成的程序是一个给定的递归结构的函数模板. 由于待生成函数具有较清晰的函数结构, 使用现有的程序合成工具可以较好地完成这一合成任务. 这一方法相比非定向基于枚举生成引理的方法, 可以较为有效地减少无用引理所浪费的时间, 其局限性是目前只能处理等式的问题, 另外求解效率比较依赖后端的程序合成工具.

2.5 求解递归函数可满足性赋值技术

前文主要介绍使用 SMT 求解器证明递归函数性质成立 (即求解器返回 UNSAT) 的技术. 与之相对, 本节我们简单介绍和讨论 SMT 求解器寻找递归函数可满足性赋值 (返回 SAT 并给出反例) 的工作.

该方向面临的主要挑战和代表性工作: 递归函数可满足性求解需要考虑参数取值可能为无穷域 (如整数或 ADT 类型) 的问题. 对包含全称量词和未解释函数公式的否定形式寻找可满足模型, 传统的量词实例化方法[48,49]表现较差. Reynolds 等人[50]在 2016 年提出一种转换技术, 该技术通过定义抽象类型, 并给抽象类型施加约束, 将

原无穷域上的递归函数公式转换为一个可以保持可满足行的有限域上的公式, 进而使得传统的实例化方法能够得以应用. 该技术已经实现在 cvc5 求解器中, 实验结果显示其可以有效提升递归函数问题寻找可满足性赋值的求解能力.

由于处理复杂量词公式和无穷域参数的问题具有相当的挑战性, 另外递归函数问题的应用背景更多来自函数式程序验证问题, 递归函数的可满足性这一方向研究相对较少. 我们认为进一步的研究工作可以从如下方向着手: 一是发展更强大的量词处理技术, 如针对特定理论的量词消去技术; 二是引入抽象等程序分析技术, 将原问题更好地进行化简, 或转换到现有方法可处理的问题范围上完成求解.

3 自动推理系统的归纳推理技术

本节主要介绍近些年在自动推理系统中引入归纳推理等技术对带有递归定义的 SMT 公式进行推导的技术. 基于自动推理系统进行 SMT 求解, 与 SMT 求解器基于 DPLL(T) 的求解框架具有很大区别, 目前主流的自动定理证明器一般都基于 superposition 推理系统, 通过 saturation-based proof search 框架来对目标公式进行推导. 有一系列工作考虑在 superposition 推理系统中引入归纳推理[8,51–56], 其中 Hajdú等人[53,54,56]和 Hozzová等人[55]的一系列技术实现在自动定理证明器 Vampire 中, Cruanes 等人[8]的工作实现在自动定理证明器 Zipperposition 中. 下面本文先介绍自动定理证明系统的推理框架, 然后分别介绍基于该推理框架的归纳推理技术.

3.1 Saturation-based proof search 推理框架

给定一个子句集合 S, 我们称基于推理系统 $\mathcal{I}$ 从 S 开始经过一系列推理规则生成的所有 S 的逻辑推论集合的称为 S 的闭包. 当闭包中包含 $\bot$ 时, 原子句集合 S 是不可满足的. 这一计算闭包的过程称为 saturation. 基于 saturation 的证明搜索策略是现在自动定理证明器的主流技术, 为了提高实际使用中的算法效率, 实际工具中会引入许多启发式方法用于选取合适的推理子句、推理规则和简化搜索空间, 这里本文对基于 saturation 的证明过程进行简单介绍.

假设待验证问题由给定前提 (可称为公理) 和待验证问题组成, 假设前提子句的集合为 A, 待验证问题子句集合为 B, 如例 9 中, $A = \{\forall x.\ x = a \vee x = b,\ p(a),\ p(b)\}$, $B = \{\forall x.\ p(x)\}$. 算法的直观思路与基于 SMT 求解器进行证明相同: 通过给定前提, 证明待验证公式 $\neg B$ 的不可满足性, 这一过程称为从 A 到 $\neg B$ 的“反驳证明 (refutation)”. 具体步骤如下.

1) 初始化 S 集合为 $A \cup \neg B$.

2) 选取当前待验证的性质所对应的子句集合 G, 基于推理系统 $\mathcal{I}$ 生成一系列推论 $C_1, \ldots, C_n$.

3) 将推论加入 S 的集合中, 若此时 S 的集合包括空集合 $\bot$, 那么表示对原待验证命题取反作为前提时, 最终会推导出 false 的结论, 于是我们证明了待验证命题成立. 否则重复上述过程.

注意到上面的算法步骤中我们没有给出如何证明待验证命题为反例或算法何时终止的判断, 在实际定理证明系统中需要基于推理过程中推理规则的选取、未处理子句和重复子句的处理等过程来证明反例, 另外通常在给定时间或空间资源耗尽还没有返回结果时返回 unknown. 具体算法过程和优化技术详见文献 [11,57].

这里我们通过一个简单的例子来初步理解 superposition 演算中各种规则的应用.

例 9: 假设我们的符号表中常数集合为 $\{a, b\}$, 即任意变量 x, 要么 $x = a$, 要么 $x = b$, 且对于命题 p、$p(a)$、$p(b)$ 均成立. 要证明公式 $\varphi = \forall x.\ p(x)$ 成立.

使用反证法的原理来证明 φ 成立, 推理过程如下: 首先假设 $\neg\forall x.\ p(x)$, 即 $\exists x.\ \neg p(x)$ 成立, 将存在量词消去 (即斯科伦化), 引入斯科伦常数 k, 有 $\neg p(k)$ 成立. 由问题叙述, $\forall x.\ x = a \vee x = b$, $p(a)$ 和 $p(b)$ 均为前提条件公式. 于是令 $\theta : x \mapsto k$, 它是 x 和 k 的 mgu, 即 $x\theta = k\theta$. 由 Sup3 规则有第 1 步推理 (其中变量 x 的公式表示该公式对任意 x 成立, 全称量词被隐去):

$$\frac{x = a \vee x = b \quad \neg p(k)}{(\neg p(a) \vee x = b)\theta}\ (\text{Sup3}).$$

第 1 步结论 $(\neg p(a) \vee x = b)\theta$ 化简为 $\neg p(a) \vee k = b$. 由第 1 步推理的结论和条件 $p(a)$, 由 Bin 规则有第 2 步推理:

$$\frac{\neg p(a) \vee k = b \quad p(a)}{k = b}\ (\text{Bin}).$$

由第 2 步结论 $k = b$ 和条件 $\neg p(k)$, 此时项中没有变量, θ 满足 $k = k\theta = b\theta = b$, 可应用 Dem 规则进行第 3 步推理:

$$\frac{k = b \quad \neg p(k)}{\neg p(b\theta)}\ (\text{Dem}).$$

结论 $\neg p(b\theta)$ 化简为 $\neg p(b)$. 由第 3 步结论和条件 $p(b)$ 可由 Bin 规则进行最后一步推理:

$$\frac{\neg p(b) \quad p(b)}{\bot}\ (\text{Bin}),$$

即假设 $\neg\forall x.\ p(x)$, 最终导出矛盾, 因此证明 $\forall x.\ p(x)$ 成立.

3.2 在推理框架中引入自动归纳推理

在第 3.1 节介绍基于 saturation-based proof search 推理框架基础上, 我们介绍 Vampire 定理证明器近年来在这一证明框架中引入自动归纳推理的一系列工作[51,53–56]. 这一系列技术具有统一的思路和框架, 我们首先介绍这些工作在自动定理证明推理系统中引入归纳推理技术的基本方法. 然后分别介绍每项工作在此基础上所提出的改进或扩展方法. 对于待验证公式, 当尝试通过归纳推理进行推导证明时, 通常会产生两个待验证的相关子目标公式, 分别表示归纳法中的基本步骤 (base case) 和归纳步骤 (induction step case). 然后基于一种归纳模式 (induction schema), 令推理系统可以通过归纳法的基本步骤与归纳步骤成立推导出原待验证目标成立.

例如待验证公式为 $\forall x \in nat.\ F[x]$. 那么会产生两个待验证子目标 $F[O]$ 和 $\forall z \in nat.\ (F[z] \to F[s(z)])$. 基于结构归纳模式 (structural induction schema):

$$(F[O] \wedge \forall z \in nat.\ (F[z] \to F[s(z)])) \to \forall x \in nat.\ F[x] \tag{3}$$

可证明原公式 F. 这一过程实际上和在数学中使用结构归纳法完成数学命题的证明过程相同, 为了在推理系统中完成证明, 2019 年 Regar 等人[51]提出在基于 saturation 证明搜索算法中引入一种表示归纳模式的推理规则来进行归纳推导的方法: 在例 9 推导步骤第 2 步中, 选择表示归纳性质的公式集合 G, 然后生成一系列新的归纳公理 $C_1,\ldots,C_n$, 例如上文所提到的分别表示归纳基础步骤和归纳步骤的公式 $F[O]$ 与 $F[s(z)]$ 所对应的子句. 这一归纳推理方法的实际使用主要依赖于两点: 1) 找到合适的归纳模式; 2) 开发高效的归纳推理规则来生成归纳公理或辅助证明的公式. Regar 等人[51]引入如下归纳规则:

$$\frac{\neg L[t] \vee C}{cnf(F \to \forall x.\ L[x])}\ (\text{Ind}),$$

其中, t 是一个基项, L 是一个基文字 (ground literal), C 是一个子句, $F \to \forall x.\ L[x]$ 是一个有效的归纳模式. 这一规则的直观思路是, 在推理证明过程中将待验证公式取反, 消去存在量词, 将得到包含基项的子句. 这一推理规则的引入希望能使推理系统从这一待反驳子句自动生成相应的归纳模式, 即这里的推论中的公式 $F \to \forall x.\ L[x]$ 即对应公式 3. F 表示基本步骤和归纳步骤的公式, $\forall x.\ L[x]$ 表示待验证目标. 注意到推论公式等价于 $\neg F \vee \forall x.\ L[x]$, 实际实现中会基于 binary resolution 规则, 只将 $cnf(\neg F \vee C)$ 加入当前子句的搜索空间中. 这一实现可以更好地引导推理系统尽早选取所生成的用于归纳推理的子句进行下一步推理.

下面我们结合实际例子介绍 Vampire 的系列工作基于上述归纳推理框架分别在代数数据类型和整数类型上的研究工作.

例 10: 假设对于代数数据类型 nat, 递归定义函数 add、$even$ 和 $half$ 分别对应如下公理:

$$\begin{cases} add : \forall y \in nat.\ add(O,y) = y, & \forall z,y \in nat.\ add(s(z),y) = s(add(z,y)) \\ even : even(O) = \top, & \forall z \in nat.\ (even(s(z))) \leftrightarrow \neg even(z) \\ half : half(O) = O,\ half(s(O)) = O, & \forall z \in nat.\ (half(s(s(z)))) = s(half(z)) \end{cases}.$$

待验证目标为 $\forall x \in nat.\ even(x) \to (x = add(half(x), half(x)))$.

在例 10 中将待验证目标取反, 再进行斯科伦化, 得到下面两个子句:

$$even(\sigma_0) \tag{4}$$

$$\sigma_0 \neq add(half(\sigma_0), half(\sigma_0)) \tag{5}$$

其中, σ_0 为常数 (类似于 SMT 中的斯科伦常数).

对于推理规则 Ind, Regar 等人[51]引入了结构归纳 (structural induction) 和良基归纳 (well-founded induction) 两种归纳模式, 上述归纳模式均定义在代数数据类型理论上, 而 Hozzová等人[55]则考虑对整数理论的归纳推理进行增强, 引入了整数归纳 (integer induction) 归纳模式. Hajdú等人[54]则区别于引入固定归纳模式的思路, 而是考虑基于公理前提中递归函数的定义 (induction with recursive function definitions) 来生成规模模式, 使推理系统在选择归纳模式时具有更高的灵活性和与公理前提的相关性. 下面分别对这些归纳模式进行介绍.

代数数据类型理论上的结构归纳: 基于结构归纳法的归纳模式分别选择代数数据类型的基础构造子和其他构造子作为基础步骤与归纳步骤的假设, 例如在 *nat* 理论上即实例化为公式 (3). 假设我们对公式 (5) 应用 Ind 规则, 那么将得到对应 $F \to \forall x.\, L(x)$ 的公式为:

$$(O = add(half(O), half(O))) \wedge (\forall z \in nat.\, z = add(half(z), half(z)) \to s(z) = add(half(s(z)), half(s(z))))$$
$$\to \forall x \in nat.\, x = add(half(x), half(x)),$$

然后这一公式中对应 F 的部分将如我们前文所述取反, 转换为相应子句加入推理系统当前的搜索空间之中.

代数数据类型理论上的良基归纳: 良基归纳模式和第 2.3 节中对 SMT 求解器引入归纳增强的原理类似. 实际上结构归纳模式也可以视作良基归纳模式的一种特例. 定义项上的二元良基关系 R. 良基归纳的原理对应的形式化公式如下:

$$\forall x.\, (\neg L[x] \to \exists y.\, (R(y,x) \wedge \neg L[y])) \to \forall x.\, L[x] \tag{6}$$

注意到这里的公式形式与第 2.3 节中公式 (2) 的形式略有区别, 但所表示意义是相同的 (实际上对公式 (2) 稍作变形即可以得到公式 (6). 将 $\forall x.\, (\neg L[x] \to \exists y.\, (R(y,x) \wedge \neg L[y]))$ 转换为等价公式 $\forall x.\, (L[x] \vee \exists y.\, (R(y,x) \wedge \neg L[y]))$, 我们可以理解为要么对于任意项 x, 命题 $L[x]$ 成立, 要么存在一个 R 关系下最小的项 y, 使得 $L[y]$ 不成立.

良基归纳方法的实例同样类似于第 2.3 节, 实际推理系统中考虑 R 为代数数据类型中基于构造子和析构子的直接子项关系. 例如对于自然数的归纳定义 *nat*, 公式 (6) 中的 $R(y,x)$ 表示为 y 是 x 的直接子项 $p(x)$, 于是 $\exists y.\, (R(y,x) \wedge \neg L[y])$ 实例化为 $\exists y.\, (y = p(x) \wedge \neg L[p(x)])$.

整数归纳: 对于递归函数相关公式进行证明时, 往往需要考虑对整数理论或代数数据结构与整数的混合问题. 而现有工作往往专注于研究在代数数据结构理论的公式项上引入归纳推理, 而忽视整数理论的问题. 即使支持整数理论的求解, 也只有最简单的实现, 例如 SMT 求解器中引入的归纳推理增强方法, 对于整数问题, 假设待验证函数只考虑自然数区间, 并且取良基关系 $R(s,t)$ 为 $0 \leqslant s = t-1$. 这一处理方式可以理解为 SMT 求解器对于整数理论的问题只支持使用弱数学归纳法进行求解. Hozzová等人[55]则考虑整数理论问题中当变量定义在整数 $\mathbb{Z}$ 上时, 其中的大于或小于关系不再是良基关系. 那么需要引入新的归纳推理模式来处理这一情况. 将原简单的自然数区间扩展为考虑任意 $\mathbb{Z}$ 的具有上限 (upper bound) 或下限 (lower bound) 的子集. 那么此时这一个子集中的大于或小于号具有良基性质, 可以进行归纳推理. 具体来讲, 引入如下 4 种归纳模式:

$$F[b] \wedge \forall y \in \mathbb{Z}.\, (y \leqslant b \wedge F[y] \to F[y-1]) \to \forall x \in \mathbb{Z}.\, (x \leqslant b \to F[x]) \tag{7}$$

$$F[b] \wedge \forall y \in \mathbb{Z}.\, (y \geqslant b \wedge F[y] \to F[y+1]) \to \forall x \in \mathbb{Z}.\, (x \geqslant b \to F[x]) \tag{8}$$

$$F[b_2] \wedge \forall y \in \mathbb{Z}.\, (b_1 < y \leqslant b_2 \wedge F[y] \to F[y-1]) \to \forall x \in \mathbb{Z}.\, (b_1 \leqslant x \leqslant b_2 \to F[x]) \tag{9}$$

$$F[b_1] \wedge \forall y \in \mathbb{Z}.\, (b_1 \leqslant y < b_2 \wedge F[y] \to F[y+1]) \to \forall x \in \mathbb{Z}.\, (b_1 \leqslant x \leqslant b_2 \to F[x]) \tag{10}$$

公式 (7)–(10) 分别描述了对于整数变量在指定符号化的上界、下界、区间约束下的归纳推理模式. 这些归纳模式的引入可以扩展自动推理系统对整数理论问题的归纳推理能力, 处理形式更丰富的问题类型, 而不局限于将变量定义域固定为自然数, 且递归函数的基础构造子只能从常数开始的问题.

例 11: 假设我们定义一个在 SMT 中整数理论下的函数 *sum*, 它用于计算从整数 n 到整数 m 的求和, 即对任意

$n,m\in\mathbb{Z}$, $n\leqslant m$, 定义 $sum(n,m)=n+(n+1)+\ldots+m$, 这里我们与前面的例子一样直接用公理形式表示函数定义:

$$\begin{cases}\forall n\in\mathbb{Z}.\ sum(n,n)=n\\ \forall n,m\in\mathbb{Z}.\ n\leqslant m\rightarrow sum(n,m)=n+sum(n+1,m)\end{cases}.$$

待验证断言为 $\varphi=\forall n,m\in\mathbb{Z}.\ n\leqslant m\rightarrow 2\times sum(n,m)=m\times(m+1)-n\times(n-1)$.

从该问题中递归函数的定义可以看出, 对于 n 的定义区间不是常数, 且 $sum(n,m)$ 的结果依赖 $sum(n+1,m)$, 这与简单的递归函数问题定义方向相反, 于是如 SMT 求解器中简单的数学归纳法实现无法处理这种类型的问题, 而基于整数归纳模式公式 (7), 则例 11 的问题可以得到证明.

基于递归定义函数的归纳: 前面我们介绍了通过考虑固定的归纳推理函数模式, 如结构归纳和良基归纳等, 来应用推理规则 Ind 的方法, Hajdú等人[54]则从另一种思路: 通过利用递归定义函数的方式来生成推理规则 Ind 中的归纳公式. 对于一个 n 元函数 f, 假设 f 的函数定义对应的公理子句为 $f(\bar{s})=t\vee C$, 这里 $\bar{s}$ 表示一个参数向量. 称 $f(\bar{s})$ 为函数头 (function header), 称任意 $f(\bar{s}')\trianglelefteq t$ (这里 $\trianglelefteq$ 表示 $f(\bar{s}')$ 是 t 的子项) 为这个函数头 f 的递归调用 (recursive call). 对于函数 f 的第 i 个参数对应的递归调用 $f(s_i)$, 这里 $1\leqslant i\leqslant n$, s_i 是一个只包含构造子和变量的代数数据类型项, 那么称这个 s_i 为归纳参数. 于是可以通过函数定义的归纳参数生成归纳模式.

例 12: 以例 10 中的 $half$ 函数为例, $half$ 的递归定义公理分别为 $half(O)=O$, $half(s(O))=O$ 以及 $\forall z\in nat.\ half(s(s(z)))=s(half(z))$. 那么归纳模式的基础步骤可以由前两个公理中函数头的第 1 个参数 O 和 $s(O)$ 生成. 归纳步骤也对应于第 3 个公理. 其中函数头 $half(s(s(z)))$ 有唯一的参数 $s(s(z))$. 这是一个代数数据类型项, 且对于递归调用 $half(z)$ 的第 1 个参数 z, 有 $z\trianglelefteq s(s(z))$. 于是这一公理可生成归纳模式的归纳步骤. 最终生成如下归纳模式的公式:

$$F(O)\wedge F[s(O)]\wedge\forall z.\ (F[z]\rightarrow F[s(s(z))])\rightarrow\forall x.\ F[x].$$

从本例中可看出基于归纳函数定义的归纳模式生成方法, 可以更灵活地在推理系统中引入归纳模式, 而不局限于固定的模式模板.

前面我们主要介绍了如何生成合适的归纳模式的一系列工作, 下面将介绍在帮助提升推理系统中进行归纳推理效率的几种主要技术.

泛化归纳技术: 这一技术由 Hajdú等人[53]引入 superposition 推理系统中, 思路类似于第 2.3 节中引入子目标来证明原有命题: 当公式 A 难以直接证明时, 改为证明 B, 且 $B\rightarrow A$. 这一过程称为证明 A 的一个泛化 (generalization) B. 在 SMT 求解以及定理证明工具中通常会基于一些启发式方法来猜测这样的一个子目标 (或者称为引理). 但在基于 saturation 的推理证明过程中只能通过推理规则的形式来修改搜索的子句空间, 而不能直接用生成的子目标公式来替换原目标公式, 因此为了在推理系统中使用子目标生成技术, Hajdú等人[53]提出推理系统中的泛化归纳技术: 引入一个新的推理规则 IndGen, 使得推理中可以添加所生成的辅助公式 B 的实例, 然后可以使用在这些实例上进行归纳推理. 这一规则公式如下:

$$\frac{\neg L[t]\vee C}{cnf(F\rightarrow\forall x.\ L'[x])}\ (\text{IndGen}),$$

类似 Ind 规则, 这里 t 是基项, L 是基文字, C 是子句, $F\rightarrow\forall x.\ L'[x]$ 是一个有效的归纳模式, 主要不同点在于这里 $L'[x]$ 是通过从 $L[t]$ 中将某些 t 的出现替换为 x 获得.

例 13: 假设对于任意 $x\in nat$, 我们希望验证如下结合律性质成立:

$$\forall x\in nat.\ (x+(x+x))=((x+x)+x)\tag{11}$$

对公式 (11) 使用 Ind 规则, 选择基于公式 (3) 中定义的归纳模式, 得到如下公式:

$$\begin{aligned}((0+(0+0))=(0+0)+0\wedge\forall x.\ (x+(x+x)=(x+x)+x\rightarrow s(x)+(s(x)+s(x))=(s(x)+s(x))+s(x)))\rightarrow\\ \forall y.\ (y+(y+y)=(y+y)+y)\end{aligned}\tag{12}$$

然后取反再进行斯科伦化, 得到下面两个子句:

$$0+(0+0)\neq(0+0)+0\vee\sigma+(\sigma+\sigma)=(\sigma+\sigma)+\sigma\tag{13}$$

$$0+(0+0)\neq(0+0)+0\vee\sigma+(s(\sigma)+s(\sigma))=(s(\sigma)+s(\sigma))+s(\sigma)\tag{14}$$

这一步之后推理系统无法进行进一步有效的推理. 但引入 IndGen 规则后, 令 t 是 σ_1, $\neg L[t]$ 是 $\sigma_1+(\sigma_1+\sigma_1)\neq(\sigma_1+$

$\sigma_1)+\sigma_1$, 我们可以对公式 (11) 应用 IndGen 规则得到如下公式:

$$\begin{aligned}&((0+(\sigma_1+\sigma_1))=(0+\sigma_1)+\sigma_1\wedge\\&\forall x.\,(x+(\sigma_1+\sigma_1)=(x+\sigma_1)+\sigma_1\rightarrow s(x)+(\sigma_1+\sigma_1=(s(x)+\sigma_1)+\sigma_1)\\&\rightarrow\forall y.\,(y+(\sigma_1+\sigma_1)=(y+\sigma_1)+\sigma_1)\end{aligned}\tag{15}$$

结合公式 (15) 和公式 (12), 则可以证明原命题, 一个完全的推理过程详见文献 [53].

归纳假设重写技术: 为了提升基于 saturation 的搜索证明过程效率, 通常我们需要确保在推理中规模大的项或者文字总能被规模小的所重写 (这里的大小一般基于我们前面所定义的某种化简序关系 $\succ$). 但在归纳推理中往往经常需要使用归纳假设来重写结论, 而归纳假设的规模一般会更大, 这会与通常的规定产生冲突. Hajdú等人[54]提出归纳假设重写技术来克服这一冲突. 引入一个新的推理规则:

$$\frac{l=r\vee D\quad s[l]\neq t\vee C}{cnf(F\rightarrow\forall x.\,(s[r]=t)[x])}\ \text{(IndHRW)},$$

其中, $s[l]\neq t$ 是一个归纳结论的文字, 它对应的归纳假设文字是 $l=r$, $l\prec r$, $F\rightarrow\forall x.\,(s[r]=t)[x]$ 是一个有效的归纳公式. 这一规则的引入: 1) 使推理系统可以对归纳结论文字的一边用它的归纳假设进行重写 (可以违反序关系限制). 2) 可以在重写的归纳结论文字上进行归纳推理且不增加搜索空间.

上面我们介绍了在基于 superposition 的推理系统中引入归纳推理的主流归纳推理技术, 除此之外还有一些工作有类似的尝试.

Cruanes 等人[8]将递归定义函数转换为推理系统中的重写规则, 然后基于 splitting 规则和一些启发式方法来进行归纳推理中的子句选取和子目标引理生成. Cruanes 等人[8]的方法实现在 Zipperposition 这一工具中, 其主要局限是只支持对代数数据类型的问题进行结构归纳, 另外他们的算法作为比较早期的尝试, 依赖集成在 AVATAR 的定理证明框架中进行逐例分析和引入特定的启发式优化.

Echenheim 等人[52]通过定义一系列推理规则增量式地生成归纳不变式的方式在自动定理证明系统中进行归纳推理, 但他们的方法同样只适用于基于子项顺序定义的代数数据类型问题, 然后他们的算法需要在子句层面引入一些额外约束条件, 然后在约束子句上将原来标准的重复子句消去算法进行扩展. 另外 Echenheim 等人[52]只提供了理论分析, 而没有提供将算法实现在真实工具中进行实现和对比实验的结果.

相比之下, Vampire 的系列工作基于 saturation 证明搜索框架这一层次进行归纳方法引入, 具有更高的抽象层次, 适用性更广泛, 更易于扩展到其他定理证明系统和实现. 但他们依然在一些方面具有局限性, 例如对于公式规模较大、递归结构较复杂、存在多个可归纳项的问题, 基于启发式归纳模式选择和子公式生成方法可能无法覆盖到更一般的情况, 从而造成求解失败. 另外 Vampire 的系列方法所能处理的递归定义函数有限, 对于函数中包含分支的附带条件或分支条件无法折叠到函数头中的更复杂的递归定义函数, 由于它们的良基性质不明显, 因此现有算法还无法进行处理. 此外推理证明系统基于 saturation 的搜索证明框架, 目前还缺少对更丰富的理论进行有效求解的支持, 从而可能使得自动定理证明器在求解混合理论问题上效率远低于基于 DPLL(T) 框架的 SMT 求解器.

4 基于 CHC 的求解技术

在第 2 节和第 3 节中, 主要介绍了基于 SMT 求解器和定理证明推理系统求解带递归定义 SMT 公式的技术. 在本节中, 我们不直接处理通用形式的 SMT 公式, 而是考虑先将待求解的 SMT 公式转换为 CHC 形式, 然后利用 CHC 求解技术来进行求解. 由于 CHC 求解技术的特点, 这一种方法可以较好地作为直接求解 SMT 公式方法的一种补充. 这一点在第 5 节实验部分也将有所体现.

CHC 是一种特殊的一阶逻辑公式表现形式. 其最早引入是为了求解程序验证问题. 对于待验证程序, 将其编码为一阶逻辑公式的验证条件后, 往往其中的变量是带有全称量词的, 而 SMT 求解器对某些带量词的问题求解能力有限. 为了对传统的 SMT 求解器进行补充, CHC 方法被提出. CHC 问题可以视作一类特殊的 SMT 问题, 它将程序待验证条件表示成特殊的形式, 即 CHC 公式, 然后通过求解 CHC 公式的可满足性来对原程序待验证性质进行验证. 基于 CHC 的验证方法往往有利于原程序中带有循环的问题, 因为 CHC 求解技术通常会尝试生成循环不

变式. 在本文中, 我们考虑的主要问题是带有递归结构的 SMT 公式求解, 由于递归函数定义一般具有基础步骤和递归步骤, 这一形式类似于迁移系统中的起始状态和状态迁移, 可以分别对应将递归函数编码为 CHC 形式. 于是除了上述直接在 SMT 层面进行求解的技术外, 还可以将复杂的 SMT 问题编码为 CHC, 然后进行求解. 这一方法往往可以利用 CHC 求解技术计算原问题中递归函数定义的归纳不变式的优势, 来求解原 SMT 方法无法求解的问题, 作为对直接面向 SMT 求解方法的一种补充. 下面首先介绍求解 CHC 问题的主流方法框架, 然后介绍针对带有递归结构 CHC 问题进行求解的技术.

4.1 约束霍恩子句

约束霍恩子句 (CHC) 是一阶逻辑的一部分, 通常用于程序验证和程序合成问题中. 一个 CHC 通常表示成如下形式的公式:

$$\forall V.\ (\varphi \wedge p_1(\mathbf{X}_1) \wedge \ldots \wedge p_n(\mathbf{X}_n)) \rightarrow h(\mathbf{X}).$$

令 T 表示一阶逻辑的背景理论, 如线性实数算术理论、Bool 理论或理论的组合, φ 是在背景理论下的约束 (constraint), V 是变量, $\mathbf{X}_i$ 是 V 上的项, $p_1,\ldots,p_n,h$ 是谓词, $p_i[\mathbf{X}_i]$ 是谓词在一阶逻辑项上的应用. 通常将 $p_i(\mathbf{X}_i)$ 形式的公式称为一个原子公式 (atom), $h(\mathbf{X})$ 被称为 CHC 公式的 head 部分, 它是一个原子公式或者 false. $\forall V.\ (\varphi \wedge p_1(\mathbf{X_1}) \wedge \ldots \wedge p_n(\mathbf{X}_n))$ 通常称为 CHC 公式的 body 部分. 如果 head 是一个形如 $p(t_1,\ldots,t_n)$ 的原子公式, 那么称谓词 p 是一个 head predicate. 对于 head 是一个原子公式的子句, 称它为 definite 子句, head 是 false 的子句称为 goal 子句.

4.2 CHC 可满足性

对于一个 CHC 公式的集合 Π, 称它的 T-model 是它的模型 M 在背景理论 T 下的扩展, 该扩展带有对每个谓词 p_i 的一阶逻辑解释, 该解释使 M 中 Π 的所有子句为 true. 从谓词 p_i 到背景理论 T 中公式的替换 σ, 当 $\Pi\sigma$ 在 T 理论中有效 (valid), 即公式在变量的任意赋值均为真时, 称该替换为 Π 的 T-solution.

例 14: 我们用一个程序验证中的问题来解释上述可满足性相关定义, 如下.

1 assume (x <= 0)	
2 while (x < 5) {	
3 $x = x + 1$	$\forall x.\ x \leqslant 0 \rightarrow Inv(x)$
4 }	$\forall x,y.\ Inv(x) \wedge x < 5 \wedge y = x+1 \rightarrow Inv(y)$
5 assert (x < 10)	$\forall x.\ Inv(x) \wedge \neg(x < 5) \wedge \neg(x < 10) \rightarrow$ false

给定例 14 代码左边的程序, 它假设前置条件为 x <= 0, 经过一个循环, 要验证跳出循环后满足性质 x < 10. 将程序编码为 CHC 问题, 用未解释谓词 Inv 表示程序中的循环不变式, 那么程序待验证断言的正确性可以通过 CHC 的可满足性得到. 求解右边的 CHC 公式, 本例的 T-model 可以视作对算术理论带有对谓词 Inv 的如下解释的扩展:

$$Inv^M = \{z \mid z \leqslant 5\}.$$

且例 14 代码右边的 CHC 公式有一个 T-solution, 将 Inv 进行如下定义:

$$Inv = \lambda z.\ z \leqslant 5,$$

将这一定义代回 CHC 公式, 可以轻易地验证原公式是 valid, 即它和我们的定义相符合.

CHC 的提出主要用于解决程序验证问题, 将程序编码为 CHC 公式后, CHC 公式和原程序验证问题有如下的对应关系: 1) 一个程序满足待验证性质当且仅当其对应的 CHC 公式是可满足的. 2) CHC 公式的模型对应原程序中的验证证明 (verification certificate), 如循环不变式. 3) CHC 公式不可满足对应一个原程序中违反性质的反例. 在本文中我们主要将关注利用 CHC 来求解带有递归函数和归纳数据结构的问题, 为了和其他方法/工具进行统一比较, 我们将对 SMT 公式进行编码和转换, 表示为 CHC 的形式然后求解, 下面介绍具体的求解和编码技术.

4.3 CHC 求解方法

主流的 CHC 求解方法有两大类: 一类是基于模型检测算法 IC3/PDR[58,59]的求解方法, 另一类是基于谓词抽象和 Craig 插值进行抽象精化的方法. 这两类方法都可以认为是将程序视作迁移系统, 然后尝试生成程序的归纳不变式来完成性质验证.

基于 IC3/PDR 的求解算法: CHC 求解工具 Spacer[60,61]中实现了基于 IC3/PDR 的求解算法. 该算法与模型检测中经典的 IC3/PDR 算法思路相同, 可以认为是将 IC3/PDR 算法进行迁移并适配到 CHC 的框架下. 主要原理是将原问题视作一个迁移系统安全性验证问题. 从初始状态开始迭代构造表示可达状态上近似的归纳公式序列. 每次检查公式序列中是否存在可达坏状态的反例路径, 根据检查结果精化 (refine) 公式序列, 直到找到真实反例证明性质违反或者归纳不变式证明性质成立. 此外 Spacer 还加入了模型映射技术 (model based projection)、全局引导 (global guidance) 等优化方法.

基于谓词抽象和 Craig 插值的求解方法: CHC 求解工具 Eldarica[62] 中实现了基于谓词抽象的 CHC 求解算法, 其主要原理是尝试对原霍恩子句构建一个抽象可达图 (abstract reachability graph, ARG). 然后基于 ARG 来获得抽象的反例路径, 该反例将会被 SMT 求解器进行检查以排除假反例. SMT 检查的结果最终返回一个具体的反例路径或者一个基于 Craig 插值计算得到的新的谓词. 这一计算过程中, 不同的文献考虑使用和优化不同的插值方法, 如树插值 (tree interpolation), 析取插值 (disjunctive interpolation) 等.

CHC 求解方法所计算的归纳不变式可以视作一种用于辅助证明的引理, 对于某些无法由归纳推理求解的问题, 通过该方法可能会比较容易生成所需要的辅助引理, 但这一技术依赖比较复杂的判定求解过程, 可能无法高效生成规模较大或形式较复杂的引理公式.

例 15: 这里我们介绍一个通过 CHC 求解生成引理帮助原命题求解的简单例子[63]. 考虑如下的函数定义.

```
1 def posTwice(x: BigInt): BigInt = {
2   if x <= 0 then BigInt (0)
3   else 2 + posTwice(x – 1)
4 } ensuring (res => res != –1)
```

这个函数在输入 x 小于 0 时返回 0, 在 x 大于等于 0 时计算 $2\times x$. 需要证明的性质是 $\forall x.\ posTwice(x)\neq -1$. 注意这一个问题实际上无法通过归纳推理求解, 因为原命题太弱, 直接应用归纳假设对证明没有帮助 (例如我们假设 $posTwice(x-1)\neq -1$, 那么我们只能得到 $posTwice(x)\neq 1$). 而若将该问题转换为如下的 CHC 公式:

$$\begin{cases}\forall x.\ x\leqslant 0\rightarrow posTwiceInv(x,0)\\ \forall x,y.\ posTwiceInv(x-1,y)\wedge x>0\rightarrow posTwiceInv(x,y+2)\\ \forall x,y.\ posTwiceInv(x,y)\wedge y=-1\rightarrow \text{false}\end{cases}.$$

注意这里 *posTwiceInv* 与原来的 *posTwice* 不同, 它是一个谓词, 且输入两个参数, 分别可以对应原来 *posTwice* 函数的输入和输出, 实际上它表示一个待求解的归纳不变式. 使用 CHC 求解器则可以求解这一问题, 它将返回“可满足”, 且生成一个归纳不变式.

```
(define-fun posTwiceInv((A Int)(B Int)) Bool (>= B 0))
```

表示原 *posTwice* 函数的输出总大于等于 0, 该不变式可以视作一个辅助引理, 基于这一个引理, 则原命题可以被容易地证明.

4.4 使用 CHC 形式表示递归函数

为了与直接输入 SMT 公式的方法进行对比, 我们将按照 CHC 社区常用的方法将 SMT 中的递归定义转换为 CHC 形式, 以定义 *list* 类型的长度函数为例.

```
1 ; 定义原递归函数为谓词
2 (define-fun lenCHC(list Int) Bool)
3 ; 编码函数定义
4 (assert (forall ((x list))
5         (=> (= x nil)
6             (lenCHC x 0))))
7 (assert ((forall ((h Int) (t list) (x list) (y0 Int) (y1 Int))
8         (=> (and (lenCHC t y0) (= y1 (+ y0 1)) (= x (cons h t)))
9             (lenCHC x y1)))))
10 ; 编码待验证断言
11 (assert ((forall ((x list) (y Int))
12         (=> (and (lenCHC x y) (< y 0))
13             false))))
```

如上, 将原参数为 $list$ 的 len 函数 $len(x:list)$ 定义为二元谓词 $lenCHC(x:list,y:Int)$, 令 $len(x)=y$ 等价于 $lenCHC(x,y)$ = true. 用 definite 子句表示递归定义. 本例中两条断言分别编码原递归函数 len 在基本情况和递归情况下的构造. 对于待验证性质则表示为 goal 子句. 注意此时与 SMT 求解的方法不同, 在 SMT 求解中返回不可满足对应待验证性质被证明成立. 而在 CHC 公式的表示中, 我们已经用反证的方式定义了待验证性质. 因此当 CHC 求解器返回可满足时, 表示它找到一个归纳不变式, 不变式对应于一个 CHC 公式的模型, 使得我们的 goal 子句成立, 这表明我们通过反证的方式证明了待验证性质成立. 而如果 CHC 求解器返回不可满足, 将表示它找到一个令待验证公式取反成立的反例, 即待验证公式不成立. 对于带有递归函数的 SMT 问题, 通过如上方式转换成 CHC 形式后, 就可以通过本节所介绍的 CHC 求解方法进行求解.

4.5 基于展开和抽象的递归函数求解技术

尽管在第 4.3 和 4.4 节中, 我们介绍了可以直接通过通用的 CHC 求解算法来求解递归函数问题, 但为了提高求解效率, 2022 年 Govind 等人[15]提出在 CHC 求解器中设计专用于处理递归函数结构的算法. 这一算法思路最早来源于 2010 年 Suter 等人[32,64]的相关工作. 本节中我们首先介绍早期基于展开的递归函数求解算法, 最后介绍在 CHC 求解框架中引入基于展开和未解释函数抽象的 CHC 求解方法.

早期基于展开的递归函数求解算法, Suter 等人[32,64]从 2010 年的工作开始研究带有递归定义函数的一阶逻辑公式判定算法, 由于当时 SMT 标准还处于早期阶段, 因此他们的工作没有直接支持 SMT 公式或集成到 SMT 求解器中, 但他们提出了一种针对包含递归定义函数的无量词代数数据结构理论一阶逻辑公式的判定算法, 该算法主要在后续被一些程序验证的工作如 Scala 验证工具 Leon/Stainless、CHC 求解器[15]所继承, 可能对于 SMT 类似形式的公式求解具有一些思路上的启发意义.

下面我们对该方法进行介绍, Suter 等人[32,64]考虑无量词代数数据类型理论的 SMT 公式中包含 catamorphism 的问题. 其方法所能处理的公式主要为无量词代数数据类型理论的公式, 一般不涉及理论组合 (如整数理论等). catamorphism 类似函数式编程语言中的 fold 函数, 可以认为是一类特定的递归函数, 但其范围足够覆盖较为常见的递归函数, 如计算 $tree$ 高度的递归定义函数、计算 $list$ 元素集合的递归定义和计算 list 的长度等. Suter 等人[32,64]提供了一种基础方法来推导包含 catamorphism 的 ADT 公式, 主要思路即对 catamorphism 基于递归函数的定义进行有限步展开, 在展开过程中, 还未展开的递归部分被视作未解释函数, 为了处理引入未解释函数带来的“假反例”情况, 再在展开过程中引入一个 Bool 类型的“控制变量”的集合, 每个控制变量对应一个逻辑公式中的项, 但这个项不包含未解释函数时, 则控制变量为 true, 代表结果可信任, 否则为 false. 将控制变量集合合取原公式. 当求解器返回 SAT 结果时, 此时控制变量必然全为 true, 我们可以相信 SAT 这一结果; 当求解器返回 UNSAT 结果时, 可能有

两种情况, 即原公式为 UNSAT, 或原公式为 SAT 但存在某个控制变量为 false, 而控制变量为 false 代表其对应的逻辑公式中的项可能存在未解释函数 (即此时对该项的求解结果可能不可信), 于是需要对原公式重新检查, 对原公式调用求解器, 当返回 UNSAT 时则没有问题, 最终结果为 UNSAT, 当返回 SAT 时, 即有可能出现我们所说未解释函数造成“假反例”情况, 于是无法得到最终结果, 需要继续对递归函数进行展开, 然后重复上述过程.

该方法适用性有限且不完备, Pham 等人[65,66]对 Suter 等人[32,64]进行了扩展, 提出了一个对满足“广义充分满射条件 (generalized sufficient subjectivity condition)”的 catamorphism 完备的基于展开的判定算法, 并将算法实现在开源工具 RADA 中[67]. 该方法的一个局限在于实际应用中, 证明所给的递归函数 (catamorphism) 满足广义充分满射条件不是一个容易的事情, 另外这些工作中没有考虑组合理论问题, 因此提及的逻辑公式中的断言只有等式 (equality), 这些情况限制了该工作的实际应用范围.

CHC 递归函数求解技术: 2022 年 Govind 等人[15]将上述基于展开的方法发展到组合理论下带有递归定义的 CHC 问题求解中. 他们将包含递归函数的 CHC 公式进行预处理, 显式地定位出其中的递归函数定义, 然后对递归函数使用基于递归定义展开的判定算法来进行求解, 并且结合 k-实例化 (k-instantiation) 和未解释函数抽象 (uninterpreted function abstraction) 技术来帮助提升递归定义求解效率. 这两种方法的主要思路是将递归函数进行有限 k 步展开 (即进行 k 步实例化), 然后对剩下未展开的部分视作一个未解释函数, 这可以视作一种抽象方法, 减少求解中的公式规模. 该算法实现在工具 Racer 中, 该工具的实现基于 CHC 求解器 Spacer 的求解框架. 这一方法对于递归函数的处理主要是基于函数定义展开技术, 该技术的缺点在于无法进行自动的归纳推理, 从而可能无法对许多必须通过归纳推理求解的命题完成自动验证. 那么在实际问题中, 没有归纳推理, 基于归纳不变式和递归函数展开的 CHC 求解器与基于归纳推理的 SMT 求解器/自动定理证明器相比, 其求解表现究竟能达到什么程度? 在第 5 节, 我们将收集和构造一个用于递归函数 SMT 求解问题的数据集, 对主流 CHC 求解器, SMT 求解器和自动定理证明器进行统一实验对比.

5 实　验

我们对目前主流的开源求解器进行统一的实验比较, 以分析这些求解器在应对各种带有递归定义的一阶逻辑公式时的求解能力. 用于实验的工具分为如下 3 类: 1) SMT 求解器, 包括 Z3 求解器和 cvc5 求解器. 2) 自动定理证明器 Vampire. 3) CHC 求解器, 包括 Spacer、Eldarica 和 Racer. 用于实验比较的数据集主要分为两类, 一类是背景理论为整数理论的递归函数问题, 这一类数据集的来源是 Vampire 相关文献[55]和我们手工构造. 另一类是背景理论为 ADT 理论和整数理论混合的递归函数问题, 这一类数据集最早来源于 Reynolds 等人[31]为 CVC4 求解器引入归纳推理的工作, 后续 ADT 和递归函数求解的相关文献[12,15,47]都选取了该数据集的部分样例进行实验.

下面我们先对用于实验比较的工具和相关参数进行介绍, 然后介绍用于实验比较的数据集和这些工具在数据集上的运行结果.

5.1 工具介绍

Z3 求解器是被应用最广泛的 SMT 求解器之一, 通常作为各种程序验证、模型检测和符号执行工具的默认求解引擎, 最早由微软研究院在 2008 年开发[23]. 它支持多种背景理论的判定, 包括线性算术、位向量、数组、未解释函数、ADT 等. 对于递归函数问题的求解, 当递归函数使用 define-fun-rec 定义时, Z3 将对递归函数定义进行展开, 首先尝试在固定展开次数寻找可满足模型, 若由于展开次数限制导致不可满足, 则增长展开次数. Z3 中没有实现归纳推理算法, 因此无法应对许多深层的递归函数性质证明问题. 在本文实验中调用 Z3 时直接使用默认设置, 不添加额外参数. 将 Z3 的实验效果作为在递归函数证明问题中的基准线 (baseline), 用于与其他实现了针对递归函数求解算法的求解器进行对比.

cvc5 求解器[24]是 CVC 系列求解器的最新版本. CVC 求解器最初由斯坦福大学和纽约大学的研究团队联合开发, 是学术界被广泛使用和研究的求解器之一. cvc5 求解器在 2021 年发布, 其主要开发团队来自斯坦福大学、爱荷华大学和纽约大学等. 相比 Z3 求解器, 它的代码架构更模块化, 易于添加新理论; 并针对字符串、位向量、

ADT 和量词等理论的推理效率进行了优化和最新算法的集成. 对于递归函数求解, 它实现了用于求解递归函数可满足模型的“有限模型寻找”算法, 以及用于证明递归函数性质的归纳推理与引理生成算法, 是目前唯一能够较好地支持归纳推理证明的主流 SMT 求解器. cvc5 求解带递归函数 SMT 问题的主要参数选项如表 1.

表 1 cvc5 归纳推理参数选项

选项	说明
--fmf-fun	使用finite-model-finding技术求解使用define-fun-rec定义的递归函数可满足性问题
--dt-stc-ind	在代数数据类型的问题中基于结构归纳对存在量词消去进行归纳增强
--int-wf-ind	通过良基归纳对整数理论进行归纳推理增强技术
--quant-ind	使用所有的归纳推理技术. 实际上同时支持了上述两种数据类型理论上的归纳推理技术, 根据公式中项的具体理论自动选择合适的归纳模式
--conjecture-gen	在归纳证明中生成候选子句
--full-saturate-quant	在带量词问题的推理中尽可能多的尝试各种内置实例化技术. 由于在cvc5中进行归纳推理可以视作在量词消去过程中的一种优化技术, 例子中待验证命题都是包含全称量词的断言, 因此在实验中开启这一选项易于求解器尽可能产生结果而不直接返回unknown

Vampire 自动定理证明器最早由 Voronkov 于 20 世纪 90 年代在瑞典乌普萨拉大学主导开发, 之后经过多次代码重构和版本升级. 目前的版本由 Voronkov 和 Kovács 带领英国曼切斯特大学和维也纳技术大学的研究团队从 2014 年开始进行开发, 并进行维护[68]. Vampire 是一阶逻辑定理证明领域的代表性工具, 在自动定理证明领域的 CASC 竞赛中多次获奖. 相比 Z3 和 cvc5 等 SMT 求解器, 它的主要特点是基于 superposition 演算的推理引擎, 并在处理带有量词公式时具有相对更高的求解效率[11]. 但 SMT 求解器 (尤其是 Z3) 在工业级应用上更广泛和成熟, 且支持更广泛的 SMT 理论, 相比之下 Vampire 更多流行于学术界, 且目前还不支持浮点数、位向量和字符串理论求解[69]. 对于递归函数问题, Vampire 中实现了众多用于归纳推理证明的算法 (详见第 3 节), 主要参数选项如表 2. 由于 Vampire 的参数选项较多, 开发者实现了易于用户使用的 portfolio 运行模式, 并提供了专门用于归纳推理的策略组合. 这一模式下可以使 Vampire 通过在多核并行运行多种归纳参数策略, 提高求解成功率. 我们在实验中选择使用这一模式.

表 2 Vampire 归纳推理参数选项

选项	说明
--induction int/struct/both/none	选择对应理论的归纳证明模式
--structural_induction_kind one/two/three/rec_def/all	选择在代数数据类型项的归纳中使用哪一种归纳公理
--induction_on_complex_terms on/off	在复杂项上应用归纳推理
--induction_gen on/off	在归纳推理规则中应用IndGen规则
--int_induction_interval infinite/finite/both	在归纳推理规则中应用IndHRW规则
--induction_hypothesis_rewriting on/off	选择整数归纳技术中的归纳规则
--int_induction_default_bound on/off	在整数归纳规则中选择默认的区间界

Eldarica 求解器最早由 Hojjat 等人[62,70]于 2013 年在瑞典乌普萨拉大学和瑞士洛桑联邦理工学院 (EPFL) 开发. 它接受 SMTLIB、Prolog 格式的 Horn 子句和部分 Scala 与 C 程序作为输入. 通过结合插值、谓词抽象和反例引导的抽象精化技术来求解程序验证问题. Eldarica 是求解能力最强的 CHC 求解器之一, 在 2024 年 CHC 求解竞赛 CHC-COMP 中 (Spacer 求解器未参加), Eldarica 在 LIA、LIA-Arrays、LIA-ADT-Arrays 等多个理论背景赛道中求解数最优[71]. 选取 Eldarica 作为 CHC 求解器的代表之一进行实验, 与其他求解器进行比较, 用以分析在实际样例中 CHC 求解方法在求解递归函数问题时的表现能力与优劣.

Spacer 求解器是另一代表性的 CHC 求解器. 它最早由美国卡耐基梅隆大学 (CMU) 的 Komuravelli 等人[60,61]在 2013 年左右完成原型工具开发. 之后 Gurfinkel 团队与微软研究院的 Bjørner 团队合作将 Spacer 代码进行整合, 合并到 Z3 的官方代码库中, 成为 Z3 的核心 CHC 求解引擎. 当输入逻辑为“Horn”时 (SMTLIB 中写作 (set-logic

HORN)), Z3 将自动调用 Spacer 引擎进行 CHC 公式的求解. Spacer 在学术界和工业界均具有广泛应用. 例如基于 LLVM 的 C++程序验证框架 SeaHorn, 其核心推理引擎依赖 Spacer. 在微软等公司中 Spacer 用于云服务协议和操作系统组件以及区块链等领域的安全性验证[72]. Spacer 求解核心算法类似在待验证问题对应的迁移系统上进行 IC3/PDR 模型检测算法, 且它集成了许多求解效率优化技术, 如 Craig 插值、基于模型映射技术 (model based projection) 以及全局引导技术 (global guidance) 等. Spacer 作为最主流的 CHC 求解器之一, 且它具有和 Eldarica 不同的核心推理算法, 因此我们选取它作为 CHC 求解器的另一代表进行实验. 由于下文即将介绍的 Racer 求解器是基于 Spacer 求解器进行开发, 它在 Spacer 原求解算法基础上实现了专门针对 ADT 和递归函数的求解技术, 因此 Spacer 的求解结果也作为 baseline, 用于分析 Racer 求解技术的效果.

Racer 求解器是加拿大滑铁卢大学的 Govind 等人[15]在 2022 年基于 Spacer 实现的 CHC 求解器, 它在 Spacer 算法框架上实现了专门用于处理递归函数和 ADT 类型 CHC 问题算法. 它可以视作 Spacer 求解器的扩展版本, 我们选择它进行实验比较, 分析其算法在不同类型的实际样例中表现效果.

5.2 实验样例和实验设置

我们在两类样例上进行实验比较, 分别是整数递归函数样例与 ADT 与整数混合理论样例. 其中整数递归函数样例来自 Hozzová等人[55]在 Vampire 中引入整数归纳优化的文献, 以及我们从其他程序验证问题中手工构造的样例, 这一部分样例均不包含 ADT 类型的问题; ADT 与整数混合理论样例来自 Reynolds 等人[31]在 SMT 求解算法中引入归纳推理的文献以及后续 CHC 领域研究者在 CHC 求解算法中引入 ADT 和递归函数求解的一些相关文献[12,15,47].

● 整数递归函数样例

这一类样例公式中递归函数的参数和返回值为 SMT 理论中的整数. 包括文献 [55] 中 120 个样例和我们手工构造的 102 个样例, 如表 3 所示.

表 3 整数递归函数样例

样例来源	关键递归函数	待求解性质举例
文献[55]样例	$pow(x,e)=x^e,\ x,e\in\mathbb{Z},\ e\geqslant 0$	$\forall e\in\mathbb{Z}.\ e\geqslant 0\rightarrow (x\cdot y)^e=x^e\cdot y^e$
	$sumX(x,y)=x+(x+1)+\ldots+y,\ x,y\in\mathbb{Z},\ x\leqslant y$	$\forall x,y\in\mathbb{Z}.\ 2\cdot sumX(x,y)=y\cdot(y+1)-x\cdot(x-1)$
	$f(0)=0,\ f(x)=f(x+1),\ x\in\mathbb{Z}$	$\forall x\in\mathbb{Z}.\ x\geqslant -10\rightarrow f(x)=f(0)$
手动构造	$pow2(x)=2^x,\ x\in\mathbb{N}$	$\forall x,y\in\mathbb{N}.\ 2^{x+y}=2^x\cdot 2^y$
	$bitLen(x)=bitLen(\lfloor x/2\rfloor)+1,\ x\in\mathbb{N},\ x>0;\ bitLen(0)=0$	$bitLen(2^x)=x+1,\ x\in\mathbb{N}$
	$fbi(x)=fbi(x-1)+fbi(x-2),\ x\in\mathbb{N},\ x\geqslant 2;$ $fbi(0)=0,\ fbi(1)=1$	$\forall x\in\mathbb{N}.\ fbi(x)=5\cdot fbi(x-4)+3\cdot fbi(x-5)$

文献 [55] 中 120 个样例可分为 3 类, 分别是幂函数 *pow* 相关样例、求和函数 *sumX* 相关样例和赋值函数 *f* 相关样例.

1) *pow* 函数定义见表 3 第 1 行, 表示计算 x^e, 其中 x, e 均为整数, 且 $e\geqslant 0$. 这类样例待求解目标公式是各种幂函数相关的性质, 例如 $(x+y)^e=x^e\cdot y^e$ 等.

2) 求和函数 *sumX* 定义见表 3 第 2 行, 表示计算 $x+(x+1)+\ldots+y$, 其中 $x\leqslant y$, x 和 y 均为整数. 这类样例待求解目标公式是设置不同的 x 和 y 范围, 求和函数的计算公式是否成立, 如表 3 中所示, $\forall x,y\in\mathbb{Z}.\ x\leqslant y\rightarrow 2\cdot sumX(x,y)=y\cdot(y+1)-x\cdot(x-1)$, 或者固定一个变量 y, 求解 x 不同范围下性质是否成立, 例如 $\forall x\in\mathbb{Z}.\ x\leqslant(-16)\rightarrow(2\cdot sumX(x,0)=-x\cdot(x-1))$.

3) 赋值函数 *f* 定义见表 3 第 3 行, 表示定义函数 *f*, 有 $f(0)=0$, 对任意整数 x, $f(x)=f(x+1)$. 这个函数背景是对程序中的数组进行统一赋值. 这类样例求解目标公式是设置不同的变量范围, 计算 f 需要满足的一些数学性质, 例如任意整数 x, $x\geqslant -10\rightarrow f(x)=f(0)$.

手工构造的 102 个样例分为两类.

1) 第 1 类样例的背景来源于我们 2024 年基于 Scala 验证方法对 Chisel 硬件电路设计进行参数化验证的工作[73]. 这个工作中为了能表示任意位长的位向量类型, 使用整数来模拟位向量的运算. 使用整数模拟位向量运算时涉及大量以 2 为底的幂函数和对数函数计算. 例如取位向量 R 的低 m 位得到位向量 S, 需要用取模计算来表示位向量对应数值的关系: $S.v = R.v \bmod 2^m$, 其中 $S.v$ 和 $R.v$ 表示位向量对应的数值. 该工作将电路中信号需要满足的性质表示为数学公式, 在程序中用程序规约 (specification) 进行表示, 然后从程序规约中生成验证条件 (verification condition), 基于 SMT 求解器进行求解. 验证条件往往包含大量幂函数、对数函数相关计算 (且包括取模、除法等非线性计算) 的求解, 为了使求解器能处理这些问题, 在文献 [73] 中我们使用 Stainless 工具提供的接口, 在程序层次手工书写归纳推理证明过程和进行验证条件公式化简.

我们从上述问题背景下提取 SMT 公式, 注意到这些 SMT 公式所包含的重点函数定义, 如幂函数、对数函数等均为递归函数. 在本文中我们考察在不经过程序层次用户手工书写归纳推理证明对公式进行化简的情况下, 直接使用现有求解工具对这些包含递归函数和复杂计算的 SMT 公式进行自动求解的能力.

第 1 类样例我们选取两个关键整数递归函数 $pow2(x)$ 和 $bitLen(x)$. 从它们对应的数学性质直接生成 SMT 公式. 然后直接使用现有求解工具进行实验. 这里两个函数的参数和返回值均为自然数. $pow2$ 函数用于计算 2 的自然数次幂, 表 3 中 $\forall x,y \in \mathbb{N}.\ 2^{x+y} = 2^x \cdot 2^y$ 是它相关的一个待验证数学性质举例. $bitLen$ 函数表示数值 x 的二进制位向量形式需要的位长. 它实际上可以用于计算参数为正整数、返回值取整的 2 为底的对数函数, 但 $bitLen$ 具有更直观的硬件类型背景, 因此本文选择它作为样例. 令 $x = 0$ 时返回 0; $x > 0$ 时, $x = 1$ 时, 对应二进制 $x = 1_{(2)}$, 需要 1 位表示; $x = 2$ 时, 对应二进制 $x = 10_{(2)}$, 需要 2 位表示; 以此类推有递推关系 $bitLen(x) = bitLen(\lfloor x/2 \rfloor) + 1$, 对 x 为正整数成立. 注意这里 $bitLen$ 函数的递推关系中出现了取整除法运算, 而现有的整数递归函数样例中一般只考虑线性加法, 例如最简单的 $f(x) = af(x-1) + b$ 形式, 递推关系是从 $x-1$ 到 x, 对应弱数学归纳法, 而从 $\lfloor x/2 \rfloor$ 到 x 的递推关系一般需要强数学归纳法来进行求解. 因此这里选取 $bitLen$ 函数可以提升样例中递归函数递推关系类别的多样性.

2) 第 2 类样例来源于被广泛使用的离散数学教材[74]中递推函数相关的例题和课后习题. 其中主要包括斐波拉契数列和其他类似的递推数列函数, 待求解性质为递推数列需要满足的数学性质. 选取这部分样例的原因和 $bitLen$ 函数类似, 由于斐波拉契数列相关性质涉及从 $x-1$, $x-2$ 到 x 的递推关系归纳证明, 一般需要强数学归纳来求解, 用于丰富样例涉及的求解技术类别. 另外这部分例子来源于实际的数学问题, 更具有应用意义.

● ADT 和整数混合理论样例

这一部分样例中包含了 251 个例子, 最早由 Reynolds 等人在文献 [31] 中提供, 在后续 ADT 和递归函数求解相关文献[13,15,47]中也被部分沿用. 这部分样例由代数数据类型和整数类型混合组成, 涉及的逻辑主要是 SMTLIB 标准格式中的 UFDTLIA 理论, 即未解释函数 (UF)、代数数据类型 (DT) 和线性整数 (LIA) 混合理论. 这些例子从带有 *list*、*tree*、*queue* 和 *heap* 等递归数据结构的程序验证问题中产生.

以 *list* 为例, 样例中的 *list* 表示每个元素为整数的列表 (类型写作 *Int*, 为 SMTLIB 标准中的整数类型, 表示数学意义上的整数, 注意它与 C++等程序语言中的整型有区别). 样例中对不同的 ADT 结构定义用于表示其计算或者操作的递归函数, 这里我们以在 *list* 上定义 *len* 和 *append* 函数为例. 它们分别表示 *list* 长度的计算和在 *list* 尾端添加一个 *list* 的操作. 待求解性质为定义的递归函数线性计算, 表示性质分别为: 对任意 *list* x 和 y, x 尾端添加 y 后得到的新列表长度等于 y 尾端添加 x 的新列表长度; 对任意 *list* x 和 y, x 尾端添加 y 后得到的新列表长度等于 x 的长度加上 y 的长度.

251 个例子中包括 168 个待证明的问题和 83 个性质不成立的问题. 待证明问题需要证明性质成立, 性质将在 SMT 文件中被取反, 当 SMT 求解器返回 UNSAT 时表示反例不存在, 证明性质成立. 性质不成立的问题一般是将原问题中递归函数定义或待证明性质进行简单改动所得到, 目标是让求解器求解出性质不成立的反例, 当 SMT 求解器返回 SAT 时表示找到一个不满足性质的反例. 以表 4 中待求解性质为例: 对于 $\forall x : list, y : list.\ len((append(x,y))) = len(x) + len(y)$, 在 SMTLIB 中取反, 表示为如下断言.

```
1 (assert (not (forall ((x list) (y list))
2       (= (len(append x y)) (+ (len x) (len y)))
3 ))) ; prove
```

此时作为待证明问题, 目标是令 SMT 求解器返回 UNSAT, 表示该断言不成立, 由反证法知原性质成立. 将断言进行简单修改得到:

```
1 (assert (not (forall ((x list) (y list))
2       (= (len(append x y)) (+ (+ (len x) (len y)) 1))
3 ))) ; find cex
```

表 4　ADT 理论和整数理论混合样例

类别	样例
递归数据结构	$list := nil \mid cons(x : Int, y : list)$
递归函数	$len(nil, 0) = 0$ $\forall x : Int, y : list.\ len(cons(x, y)) = 1 + len(y)$ $\forall x : list.\ append(nil, x) = x$ $\forall x : Int, y : list, z : list.\ append(cons(x, y), z) = cons(x, (append(y, z)))$
待求解性质	$\forall x : list, y : list.\ len((append(x, y))) = len(append(y, x))$ $\forall x : list, y : list.\ len((append(x, y))) = len(x) + len(y)$

此时表示的性质为 $\forall x : list, y : list.\ len((append(x, y))) = len(x) + len(y) + 1$ 作为性质不成立问题, 目标是令 SMT 求解器返回 SAT, 表示它能找到一个模型, 使得该断言成立, 从而对应的性质不成立.

实验设置如下: 我们在 20 核内存 64 GB 的 xeon gold 5115@2.4 GHz 处理器上进行实验. 设置每个例子的时间限制为 300 s. 所有样例统一表示为通用的 SMTLIB 格式, 且其中递归函数是使用公理表示 (即不使用 define-fun-rec, 而是通过多条 assert 断言来表示). 通用的 SMTLIB 格式作为 Z3 求解器, cvc5 求解器和 Vampire 自动定理证明器的输入. 然后样例将由通用的 SMTLIB 格式转换为 CHC 形式, 作为 Eldarica、Spacer 和 Racer 这 3 种 CHC 求解器的输入.

对于整数递归函数样例, 我们运行和对比第 5.1 节中除 Racer 求解器以外的所有工具, 不对比 Racer 的原因是目前 Racer 的使用需要先通过相关文献 [15] 提供的脚本, 对原 CHC 公式进行预处理, 将其中用公理表示定义的递归函数体进行自动转换, 显式地表示成 Racer 可以处理的特定形式. 然后 Racer 才能对这些递归函数体进行识别和调用专用算法求解. 否则 Racer 无法调用特定算法, 将和 Spacer 表现一样. 而目前预处理脚本和 Racer 工具不支持本文整数样例中的递归函数定义, 只能支持 ADT 和整数混合理论样例的问题, 因此我们只在混合理论样例中才考虑 Racer 工具. 对于其他工具, 除 cvc5 求解器和 Vampire 自动定理证明器以外, 均直接使用默认命令选项. cvc5 求解器使用归纳推理增强和子引理生成的命令选项: --quant-ind --conjecture-gen --full-saturate-quant. Vampire 自动定理证明器使用 portfolio 模式的归纳推理策略 (其中包含了专用于整数归纳推理算法的命令), 命令选项为: --mode portfolio --schedule induction.

对于 ADT 和整数混合理论样例, 我们运行第 5.1 节中所有工具, 其中: 1) 在运行 Racer 工具求解前, 先通过 Racer 文献[15]提供的预处理脚本将原 CHC 公式中的递归函数定义转换为 Racer 需要的形式. 2) 对于 168 个待证明问题, 运行所有工具, 其中 cvc5 和 Vampire 使用如整数样例一样的命令选项, 其他工具均使用默认命令. 3) 对于 83 个性质不成立的问题, 由于 cvc5 和 Z3 求解器实现了用于递归函数问题的有限模型寻找 (finite-model-finding) 技术, 根据文献 [50] 报告, 该技术有利于可满足性问题的求解, 但需要输入显式定义的递归函数, 因此我们先将这些问题中通过公理断言表示的递归函数定义转换成由 define-fun-rec 表示的递归函数定义, 然后再调用 cvc5 和 Z3 进行求解. cvc5 在命令选项中加入--fmf-fun 参数, 表示调用有限模型寻找技术, 其他参数选项均和之前

实验一样.

5.3 实验结果

整数递归函数样例结果如表 5 所示. 对于总的 222 个样例, 如表 5 中加粗数字所示, 基于 SMT 求解和归纳推理算法的求解器 (Z3/cvc5/Vampire) 中求解数最多的工具是 Vampire, 可求解数为 84/222=37.84%, 基于 CHC 归纳不变式生成的求解器 (Spacer/Eldarica) 中求解数最多的工具是 Eldarica, 求解数为 105/222=47.30%. 下面分别介绍在两类样例来源上的表现.

表 5 整数递归函数样例不同求解器实验结果

样例来源	总数	Z3	cvc5	Vampire	Spacer	Eldarica
文献[55]样例	120	0	29	76	62	84
手工构造样例	102	4	29	8	15	21
总解出数	222	4	58	**84 (37.84%)**	77	**105 (47.30%)**

在 120 个文献 [55] 样例中, Eldarica 和 Vampire 解出数最多, Vampire 在这部分样例上求解数达到 76/120=63.33%, 远多于 SMT 求解器 Z3 和 cvc5. 这一表现符合预期. 因为文献 [55] 样例正是 Vampire 团队提供, 他们某种意义上针对这部分样例的形式进行了专门的整数归纳推理优化, 并实现在 Vampire 中. 相比之下, Z3 没有实现归纳推理, 解出数为 0, 而 cvc5 虽然实现了归纳推理增强, 但它在整数归纳推理上实现较为简单, 无法应对整数变量小于 0, 以及具有非常数边界的情况, 因此 cvc5 可以求解较少数量的例子.

CHC 求解器方面的求解能力稍微有些超出预期. 其中 Eldarica 解出了 84/120=70% 的样例, 甚至超过了 Vampire. 以往文献中并缺少整数递归函数样例上 CHC 求解器与其他 SMT 求解器或自动定理证明器的对比实验结果, 本文的实验可以作为补充. 结果表明 CHC 求解器即使没有专门实现归纳推理技术, 仅靠基于归纳不变式生成算法在求解整数递归函数的问题上就具有较好的求解能力. 其原因可能是 CHC 求解器对整数理论的支持较好, 且整数实验样例中的递归函数形式相对简单, 且性质公式中线性计算为主, 易于 CHC 求解器计算出归纳不变式, 完成待验证性质的证明.

对于 102 个手工构造的样例, 用于实验的工具均表现不佳, 其中求解数最多的是 cvc5 和 Eldarica, 求解的数目分别为样例总数的 29/102=28.43% 和 21/102=20.59%. 手动构造样例的主要求解难点来源于两部分, 一部分是递归函数定义和相关性质公式, 如取模计算等带来的非线性. 非线性理论的求解一直都是求解理论判定算法中非常具有挑战性的难点问题. 另一部分是根据 *bitLen* 和递推数列问题中包含的递归函数定义, 求解带有这些递归函数的性质公式依赖强数学归纳法求解. 但目前的求解器归纳推理算法中均没有实现强归纳法. 其主要原因可能是: 1) 强归纳推理的归纳假设需要在公式中额外引入全称量词; 2) 在 ADT 理论的问题中进行强归纳推理, 需要能够判定任意两个项之间的子项关系 (subterm relation). 这些都将增加求解过程中的计算开销, 带来一定的挑战性. 且现有开源样例中需要通过强归纳法进行自动推理求解的问题比较少, 使得研究者缺少研究动力 (motivation).

总的来看, CHC 求解器基于归纳不变式生成的验证方法和 Vampire 整数归纳技术的引入, 在整数递归函数样例上都具有一定的效果. 主流 SMT 求解器在整数理论的递归函数问题上, 尽管如 cvc5 实现了基于弱数学归纳法的归纳推理增强技术, 但实现较为简单, 无法处理复杂的问题类型. 主流求解器可以完成相对简单的归纳推理或归纳不变式生成的求解任务, 但难以应对复杂计算公式和函数定义的问题. 根据调研和实验结果, 现有工作对于整数递归函数 SMT 公式自动求解的研究非常欠缺. 其原因可能是: 整数归纳推理问题更多来源于数学背景 (如本文中提供的斐波拉契递推数列问题) 而不是程序验证背景. 对于数学背景问题的求解目前主流思路还是基于交互式定理证明工具 (如 Lean、Coq 等) 进行人工证明而不是机器自动证明. 因此学术界对自动归纳推理证明的研究更偏向从程序验证背景中获得的 ADT 相关问题, 从而缺少整数理论 SMT 公式自动归纳推理的开源样例和研究工作. 但随着当前人工智能的流行, 许多人开始对人工智能辅助进行数学问题的自动求解感兴趣, 作为自动求解的底层引擎, 对带有整数递归函数的 SMT 公式进行归纳推理等自动求解的技术或许将成为未来的研究热点.

ADT 和整数混合理论样例的结果如表 6 所示.

待证明的 168 个样例结果显示, 现有求解器则表现均一般. 基于 SMT 求解和归纳推理算法的求解器 (Z3/cvc5/Vampire) 中求解数最多的工具是 cvc5, 可求解数为 74/168=44.05%. 而相比整数样例, CHC 求解器在 ADT 样例上的表现较差. 其中求解数最多的工具是 Racer, 它实现了专门针对 ADT 和递归函数的求解算法, 但只能求解出 39/168=23.21% 的样例. 其原因可能是待证明样例较为依赖归纳推理进行证明, 而当前 CHC 求解器算法中无法进行自动归纳推理. 另外 CHC 求解器的理论判定算法也无法较好地处理包含 ADT 的问题, 对于复杂的 ADT 理论样例无法求解出用于证明性质的归纳不变式.

表 6　ADT 和整数混合理论样例不同求解器实验结果

样例类别	总数	Z3	cvc5	Vampire	Spacer	Racer	Eldarica
待证明样例	168	12	**74 (44.05%)**	51	4	**39 (23.21%)**	12
性质不成立样例	83	47	32	0	83	81	83
总解出数	251	59	106	51	87	120	95

结果显示, 在性质不成立样例上 CHC 求解器几乎均可以求解全部例子. 这可能得益于 CHC 求解算法中求解归纳不变式的计算框架. 如 IC3/PDR 框架本身就是模型检测问题中求解反例最优算法之一. 而相比之下, 尽管 SMT 求解器中实现了对递归函数进行有限模型寻找的技术[50], 能够求解一部分性质不成立样例, 但它搜索可满足模型的算法没有更进一步进行专门优化, 从而求解效率远低于 CHC 求解器. 实验中 SMT 求解器 Z3 和 cvc5 需要对输入文件中定义递归函数的形式进行预处理, 然后调用有限模型寻找方法 (--fmf-fun 参数), 但只能求解 47/83=56.63% 和 32/83=38.55% 的样例. 自动定理证明器 Vampire 则无法求解出任何样例. 这是由于自动定理证明器没有实现任何专用于递归函数问题可满足性求解的推理技术, 且相比之下推理系统更适合也更关注进行“证明”任务. 需要注意的是实验发现若 SMT 求解器不开启模型寻找算法, 会和 Vampire 一样无法求解任何问题. 从这一角度来看, Reynolds 等人在文献 [50] 中提出的递归函数模型寻找算法在帮助 SMT 求解器应对带有递归函数的性质不成立问题时具有重要作用但有待进一步优化和提升. 另外, 在部分关注 ADT 和递归函数求解的相关文献中[13,15], 作者同样运行 SMT 求解器 Z3 和 cvc5 与 CHC 求解器进行了实验比较, 但文中实验结果显示 SMT 求解器无法解出任何样例. 实际上这是由于它们没有开启 SMT 求解器的递归函数模型寻找算法.

从 ADT 和整数混合理论样例的实验结果中, 我们可以得出如下结论: 1) 归纳推理技术的使用具有较好的效果, cvc5 和 Vampire 在开启归纳增强选项后能求解的例子数相比不支持归纳推理的 SMT 求解器 Z3 具有较大优势, 但依然有超过一半的例子无法被求解, 因此现有求解工具都具有很大提升空间. 求解能力不足的主要原因可能是由于现有求解器中实现的引理生成技术难以处理长度较大、结构复杂的 ADT 公式, 需要研究更高效的引理生成算法. 2) SMT 求解器中实现归纳推理增强的 cvc5 求解器, 相比在推理系统中引入归纳推理增强的 Vampire 具有更好的效果, 其原因可能是 SMT 求解器对 ADT 理论以及混合理论问题的支持更好, 集成了更丰富的优化和求解算法. 3) CHC 求解算法在 ADT 理论问题的“证明”任务上求解能力有限, 但在“找错”任务上具有很好的效果. 可能是由于 CHC 求解器对 ADT 理论判定方法的支持还不够成熟. 且 CHC 求解器中仅靠归纳不变式生成和基于展开的递归函数求解算法来进行验证, 无法替代基于自动归纳推理的验证算法.

两类求解算法对比如下.

值得一提的是, 对于性质证明问题, 尽管单一求解器的求解能力有限, 且 CHC 求解器相比 Z3/cvc5/Vampire 求解效果稍弱. 但如果考虑每一个样例被任意工具求解的情况, 我们发现 CHC 工具的求解能力可以与其他工具互为补充. SMT 求解器 (Z3/cvc5) 和自动定理证明器 (Vampire) 主要基于从前提公理推导出性质取反不成立的反证法思想, 来进行待证明 SMT 问题的求解, 然后通过归纳推理技术处理递归函数; CHC 求解器则主要通过生成可推导出安全性质成立的归纳不变式等模型检测思想, 来进行待证明 CHC 问题的求解, 然后通过递归函数展开和抽象方法来处理递归函数. 在表 7 中我们分别统计了这两类求解器在待证明样例上的综合求解效果 (即只考虑前文所述所有实验样例中排除掉 83 个性质不成立样例的问题). 对每一个样例, 若它能被 Z3/cvc5/Vampire 中任意求解器求

解, 则统计到表 7 样例来源 Z3/cvc5/Vampire 下; 若能被 Spacer/Racer/Eldarica 中任意求解器求解, 则统计到样例来源 Spacer/Racer/Eldarica 下; 若能被这里所有求解器中任意求解器求解, 则统计到样例来源任意工具下. 括号中的百分比表示求解数占这一类样例总数的比例.

将表 7 括号中的百分比分别与表 5、表 6 括号中的百分比进行对比, 可以看出, 类似算法的求解工具可求解样例重合度较大, 百分比提升不明显, 但如果考虑不同算法综合求解能力, 则求解数提升较大: 对于整数样例, Z3/cvc5/Vampire 和 Spacer/Eldarica 这两类工具中, 单一求解器表现最好的分别是 Vampire 和 Eldarica, 解出数比例分别为 37.84% 和 47.30%. 如表 7 样例来源为 Z3/cvc5/Vampire 下, 整数样例总解出数的结果所示, 如果分别考虑每一类工具, 则能被任意 Z3/cvc5/Vampire 求解样例数为 52.25%, 相比 Vampire 提升 52.25%–37.84%=14.41%; 能被任意 CHC 求解器求解比例为 48.65%, 相比 Eldarica 提升了 48.65%–47.30%=1.35%. 这两者提升都较为有限. 但如果综合所有工具来看, 能被任意工具求解的样例数占比为 69.37%. 提升比例达到 69.37%–37.84%=31.53% 和 69.37%–47.30%=22.07%. 相比只考虑单一种类算法的情况提升较大. 类似地对比 ADT 样例中待证明数据结果, Z3/cvc5/Vampire 和 Spacer/Racer/Eldarica 中表现最好的分别为 cvc5 和 Racer, 求解比例如表 6 中数据 44.05% 和 23.21%. 同类算法求解比例分别为 47.62% 和 24.40%, 表明分别看两类求解器, 那么在每一类求解器中最优求解器求解样例基本覆盖其他求解器能求解的样例. 但如果综合所有求解器来看, 求解比例为 60.12%, 相比 cvc5 和 Racer 分别提升了 60.12%–47.62%=12.5% 和 60.12%–24.40%=35.72%, 具有一定的提升效果, 这表明两类工具所能求解样例可以在一定程度上互为补充.

表 7 所有待证明样例不同求解算法对比结果

样例来源	总数	Z3/cvc5/Vampire	Spacer/Racer/Eldarica	任意工具
文献[55]样例	120	86	86	108
手工构造样例	102	30	22	46
整数样例总解出数	222	116 (52.25%)	108 (48.65%)	154 (69.37%)
ADT和整数混合理论待证明样例	168	80 (47.62%)	41 (24.40%)	101 (60.12%)
所有待证明验证总解出数	390	196 (50.26%)	149 (38.21%)	255 (65.38%)

6 总 结

本文主要关注带有递归定义的 SMT 公式求解技术. 我们对求解 SMT 公式的 3 类主流工具: SMT 求解器、自动定理证明器以及 CHC 求解器分别进行介绍. 在每一类工具中详细展示了通用判定算法和针对递归定义的专用求解技术.

对于 SMT 求解器, 我们从通用的 DPLL(T) 判定算法开始, 介绍了无量词代数数据类型的理论判定算法和为求解递归函数问题的归纳增强与子引理生成技术. 对于自动定理证明器, 我们从推理框架开始, 介绍了近年来在推理系统中引入自动归纳推理的一系列工作.

上述两类是目前直接求解 SMT 公式的主流方法和工具, 它们都基于在原求解判定框架中引入归纳推理方法来提升对带有递归函数的 SMT 公式的求解能力. 可将它们归为第 1 类方法, 根据我们对这类方法相关文献的整理, 可以发现这类方法的引入来源于认识到带有递归定义的 SMT 公式尽管表示形式极为复杂, 但具有特定的归纳结构, 于是可以通过在求解环境中引入表示归纳推理的基础步骤公式和归纳步骤公式的两类特定公式来实现“归纳推理增强”. 使得求解工具具有自动进行归纳证明的能力, 不带有归纳推理增强的原算法则几乎无法自动求解任何递归函数问题. 另外在引入归纳推理增强的基础上, 辅助证明引理的生成技术对求解效率起到非常大的作用. 这是由于求解器缺少对用未解释函数和 ADT 结构表达的递归函数问题基本性质的理解, 辅助引理的引入, 可以提供必要的公理, 来提高求解器对递归函数相关计算关系和性质的认识, 帮助对复杂公式进行化简. 因此基于 SMT 求解和自动定理证明求解的方法, 主要的研究重点是在引入必要的归纳推理模式基础上, 进一步提升求解过程中辅助证明引理生成的效率和质量.

除了 SMT 求解器和自动定理证明器外, 我们还关注到可以通过 CHC 求解器来求解带有递归函数的 SMT 问题. CHC 求解器主要用于软件模型检测问题中的程序验证问题, 而 ADT 和递归函数的递归结构, 也可以被表示成 CHC 求解程序验证问题算法需要的迁移系统形式. 在使用 CHC 求解 SMT 公式时, 需要先将 SMT 公式从通用形式转换为 CHC 特定形式. 本文介绍了 CHC 求解的主流算法和通过 CHC 算法求解递归函数 SMT 问题的方法和工具. 基于 CHC 求解的方法可以称为第 2 类方法, 与第 1 类方法最大的不同在于这类方法不通过归纳推理来求解递归函数问题, 而是通过生成归纳不变式和递归函数展开的算法完成求解. 实验结果表明这一类方法能够与第 1 类方法互补.

我们通过从现有文献收集和手工构造的方式获得了两类具有程序验证背景的实验样例集. 分别是纯整数理论样例集和 ADT 理论整数理论混合样例集. 在这两个样例集上进行实验, 然后对比分析了基于不同求解算法的几种主流求解工具的表现. 实验结果表明, 现有求解器在求解带递归函数的 SMT 问题上可提升空间较大. 尤其是在“证明”任务上, 对于这两类样例, 单一求解器能求解的数目均不到一半.

这些主流求解器中, 对于递归函数问题, Z3 没有实现自动归纳推理方法, 因此能求解数目最少, 但它实现了递归函数的模型寻找方法, 可以求解一部分性质不成立的“找错”问题.

cvc5 的主要技术是基于归纳推理增强和引理生成算法, 它的综合表现较好, 在每一类问题上都能够解出一定数量的问题, 但在整数归纳推理, 非线性理论求解等部分还有待加强, 可以考虑在 DPLL(T) 求解框架中引入类似 Vampire 的整数归纳推理模式. 并进一步提升其引理生成算法在复杂样例上的能力. 结合当下大模型在程序生成和数学推理上的热门趋势, 或许可以基于大模型方法来帮助算法进一步理解复杂的递归函数公式, 并更好地生成用于辅助证明的引理.

Vampire 的主要技术是在自动推理系统中引入各种归纳推理模式, 但它的引理生成算法较弱, 这导致它在手工构造的复杂整数样例和 ADT 整数混合样例上求解表现均不如 cvc5. 虽然在推理系统中引入自动归纳证明的文献较多, 但大多数集中在引入不同类型归纳模式, 这可以认为是一种“广度优先”思路, 能够提高推理系统能求解的问题种类. 这一思路可以在其他求解算法未深入涉足的问题类型 (例如整数理论递归函数求解) 上取得优势, 但在更通用问题类别上难以胜过其他工具. 需要进一步提升推理系统求解各种类型问题的“深度”, 如加强自动定理证明工具对非线性理论和 ADT 理论背景公式的求解算法能力, 提升基于推理系统的辅助证明引理生成技术等.

CHC 求解器对于“找错”任务的处理能力好于 SMT 求解器和自动定理证明工具. 在“证明”任务上, 对于整数类型问题的处理能力相对较好, 对于复杂的 ADT 理论问题的求解能力则有待提升. 而且得益于 CHC 求解算法基于归纳不变式生成的原理, CHC 求解器可以求解一部分基于归纳推理方法无法求解的问题, 这提示我们可以尝试将自动归纳推理算法与 CHC 求解的归纳不变式生成算法相结合, 帮助提升现有工具/算法应对复杂多样理论背景问题的求解能力.

References

[1] Bjørner N, Gurfinkel A, McMillan K, Rybalchenko A. Horn clause solvers for program verification. In: Beklemishev LD, Blass A, Dershowitz N, Finkbeiner B, Schulte W, eds. Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday. Cham: Springer, 2015. 24–51. [doi: 10.1007/978-3-319-23534-9_2]

[2] Oppen DC. Reasoning about recursively defined data structures. Journal of the ACM (JACM), 1980, 27(3): 403–411. [doi: 10.1145/322203.322204]

[3] Barrett CW, Shikanian I, Tinelli C. An abstract decision procedure for a theory of inductive data types. Journal on Satisfiability Boolean Modeling and Computation, 2007, 3(1–2): 21–46. [doi: 10.3233/SAT190028]

[4] Reynolds A, Blanchette JC. A decision procedure for (co)datatypes in SMT solvers. Journal of Automated Reasoning, 2017, 58(3): 341–362. [doi: 10.1007/s10817-016-9372-6]

[5] Reynolds A, Viswanathan A, Barbosa H, Tinelli C, Barrett C. Datatypes with shared selectors. In: Proc. of the 9th Int'l Joint Conf. Held as Part of the Federated Logic Conf. on Automated Reasoning. Oxford: Springer, 2018. 591–608. [doi: 10.1007/978-3-319-94205-6_39]

[6] Hojjat H, Rümmer P. Deciding and interpolating algebraic data types by reduction. In: Proc. of the 19th Int'l Symp. on Symbolic and Numeric Algorithms for Scientific Computing. Timisoara: IEEE, 2017. 145–152. [doi: 10.1109/SYNASC.2017.00033]

[7] Shah A, Mora F, Seshia SA. An eager satisfiability modulo theories solver for algebraic datatypes. In: Proc. of the 38th AAAI Conf. on Artificial Intelligence. Vancouver: AAAI Press, 2024. 8099–8107. [doi: 10.1609/aaai.v38i8.28649]

[8] Cruanes S. Superposition with structural induction. In: Proc. of the 11th Int'l Symp. on Frontiers of Combining Systems. Brasília: Springer, 2017. 172–188. [doi: 10.1007/978-3-319-66167-4_10]

[9] Kovács L, Robillard S, Voronkov A. Coming to terms with quantified reasoning. In: Proc. of the 44th ACM SIGPLAN Symp. on Principles of Programming Languages. Paris: ACM, 2017. 260–270. [doi: 10.1145/3009837.3009887]

[10] Cruanes S. Extending superposition with integer arithmetic, structural induction, and beyond [Ph.D. Thesis]. Palaiseau: École Polytechnique, 2015.

[11] Kovács L, Voronkov A. First-order theorem proving and Vampire. In: Proc. of the 25th Int'l Conf. on Computer Aided Verification. Saint Petersburg: Springer, 2013. 1–35. [doi: 10.1007/978-3-642-39799-8_1]

[12] De Angelis E, Fioravanti F, Pettorossi A, Proietti M. Removing algebraic data types from constrained Horn clauses using difference predicates. In: Proc. of the 10th Int'l Joint Conf. on Automated Reasoning. Paris: Springer, 2020. 83–102. [doi: 10.1007/978-3-030-51074-9_6]

[13] De Angelis E, Fioravanti F, Pettorossi A, Proietti M. Satisfiability of constrained Horn clauses on algebraic data types: A transformation-based approach. Journal of Logic and Computation, 2022, 32(2): 402–442. [doi: 10.1093/logcom/exab090]

[14] Kostyukov Y, Mordvinov D, Fedyukovich G. Beyond the elementary representations of program invariants over algebraic data types. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. ACM, 2021. 451–465. [doi: 10.1145/3453483.3454055]

[15] Govind VKH, Shoham S, Gurfinkel A. Solving constrained Horn clauses modulo algebraic data types and recursive functions. Proc. of the ACM on Programming Languages, 2022, 6(POPL): 60. [doi: 10.1145/3498722]

[16] Cook SA. The complexity of theorem-proving procedures. In: Proc. of the 3rd Annual ACM Symp. on Theory of Computing. Shaker Heights: ACM, 1971. 151–158. [doi: 10.1145/800157.805047]

[17] Armando A, Giunchiglia E. Embedding complex decision procedures inside an interactive theorem prover. Annals of Mathematics and Artificial Intelligence, 1993, 8(3-4): 475–502. [doi: 10.1007/BF01530803]

[18] Armando A, Castellini C, Giunchiglia E. SAT-based procedures for temporal reasoning. In: Proc. of the 5th European Conf. on Planning Recent Advances in AI Planning. Durham: Springer, 2000. 97–108. [doi: 10.1007/10720246_8]

[19] Bryant RE, German S, Velev MN. Exploiting positive equality in a logic of equality with uninterpreted functions. In: Proc. of the 11th Int'l Conf. on Computer Aided Verification. Trento: Springer, 1999. 470–482. [doi: 10.1007/3-540-48683-6_40]

[20] Giunchiglia F, Sebastiani R. Building decision procedures for modal logics from propositional decision procedures: The case study of modal $K(m)$. Information and Computation, 2000, 162(1–2): 158–178. [doi: 10.1006/inco.1999.2850]

[21] Ganzinger H, Hagen G, Nieuwenhuis R, Oliveras A, Tinelli C. DPLL(T): Fast decision procedures. In: Proc. of the 16th Int'l Conf. on Computer Aided Verification. Boston: Springer, 2004. 175–188. [doi: 10.1007/978-3-540-27813-9_14]

[22] Dutertre B, de Moura L. A fast linear-arithmetic solver for DPLL(T). In: Proc. of the 18th Int'l Conf. on Computer Aided Verification. Seattle: Springer, 2006. 81–94. [doi: 10.1007/11817963_11]

[23] de Moura L, Bjørner N. Z3: An efficient SMT solver. In: Proc. of the 14th Int'l Conf. on Held as Part of the Joint European Conf. on Theory and Practice of Software Tools and Algorithms for the Construction and Analysis of Systems. Budapest: Springer, 2008. 337–340. [doi: 10.1007/978-3-540-78800-3_24]

[24] Barbosa H, Barrett C, Brain M, Kremer G, Lachnitt H, Mann M, Mohamed A, Mohamed M, Niemetz A, Nötzli A, Ozdemir A, Preiner M, Reynolds A, Sheng Y, Tinelli C, Zohar Y. cvc5: A versatile and industrial-strength SMT solver. In: Proc. of the 28th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Munich: Springer, 2022. 415–442. [doi: 10.1007/978-3-030-99524-9_24]

[25] Brummayer R, Biere A. Boolector: An efficient SMT solver for bit-vectors and arrays. In: Proc. of the 15th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. York: Springer, 2009. 174–177. [doi: 10.1007/978-3-642-00768-2_16]

[26] Niemetz A, Preiner M, Wolf C, Biere A. BTOR2, BtorMC and boolector 3.0. In: Proc. of the 30th Int'l Conf. Computer Aided Verification. Oxford: Springer, 2018. 587–595. [doi: 10.1007/978-3-319-96145-3_32]

[27] Nieuwenhuis R, Rubio A. Paramodulation-based theorem proving. Handbook of Automated Reasoning, 2001, 1: 371–443. [doi: 10.1016/B978-044450813-3/50009-6]

[28] Reger G, Suda M, Voronkov A. Instantiation and pretending to be an SMT solver with Vampire. In: Proc. of the 15th Int'l Workshop on Satisfiability Modulo Theories. Heidelberg: CEUR-WS.org, 2017. 63–75.

[29] Nelson G, Oppen DC. Fast decision procedures based on congruence closure. Journal of the ACM (JACM), 1980, 27(2): 356–364. [doi: 10.1145/322186.322198]

[30] Biere A, Heule M, van Maaren H, Walsh T. Handbook of Satisfiability. 2nd ed., Washington: IOS Press, 2021.

[31] Reynolds A, Kuncak V. Induction for SMT solvers. In: Proc. of the 16th Int'l Conf. on Verification, Model Checking, and Abstract Interpretation. Mumbai: Springer, 2015. 80–98. [doi: 10.1007/978-3-662-46081-8_5]

[32] Suter P, Köksal AS, Kuncak V. Satisfiability modulo recursive programs. In: Proc. of the 18th Int'l Symp. on Static Analysis. Venice: Springer, 2011. 298–315. [doi: 10.1007/978-3-642-23702-7_23]

[33] Blanc R, Kuncak V, Kneuss E, Suter P. An overview of the Leon verification system: Verification by translation to recursive functions. In: Proc. of the 4th Workshop on Scala. Montpellier: ACM, 2013. 1. [doi: 10.1145/2489837.2489838]

[34] Hamza J, Voirol N, Kunčak V. System FR: Formalized foundations for the stainless verifier. Proc. of the ACM on Programming Languages, 2019, 3(OOPSLA): 166. [doi: 10.1145/3360592]

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

[36] Kaufmann M, Manolios P, Moore JS. Computer-aided Reasoning: ACL2 Case Studies. New York: Springer, 2000. [doi: 10.1007/978-1-4757-3188-0]

[37] Sonnex W, Drossopoulou S, Eisenbach S. Zeno: An automated prover for properties of recursive data structures. In: Proc. of the 18th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Tallinn: Springer, 2012. 407–421. [doi: 10.1007/978-3-642-28756-5_28]

[38] Bundy A, Basin D, Hutter D, Ireland A. Rippling: Meta-level Guidance for Mathematical Reasoning. Cambridge: Cambridge University Press, 2005.

[39] Kapur D, Subramaniam M. Using an induction prover for verifying arithmetic circuits. Int'l Journal on Software Tools for Technology Transfer, 2000, 3(1): 32–65. [doi: 10.1007/PL00010808]

[40] Murali A, Peña L, Blanchard E, Löding C, Madhusudan P. Model-guided synthesis of inductive lemmas for FOL with least fixpoints. Proc. of the ACM on Programming Languages, 2022, 6(OOPSLA2): 191. [doi: 10.1145/3563354]

[41] Hesketh JT. Using middle-out reasoning to guide inductive theorem proving [Ph.D. Thesis]. Edinburgh: University of Edinburgh, 1991.

[42] Johansson M, Dixon L, Bundy A. Dynamic rippling, middle-out reasoning and lemma discovery. In: Siegler S, Wasser N, eds. Verification, Induction, Termination Analysis. Berlin, Heidelberg: Springer, 2010. 102–116. [doi: 10.1007/978-3-642-17172-7_6]

[43] Buchberger B. Theory exploration with theorema. Analele Universitatii Din Timisoara, Seria Matematica-Informatica, 2000, 38(2): 9–32.

[44] McCasland R, Bundy A, Autexier S. Automated discovery of inductive theorems. Studies in Logic, Grammar and Rhetoric, 2007, 10(23): 135–149.

[45] Yang WK, Fedyukovich G, Gupta A. Lemma synthesis for automating induction over algebraic data types. In: Proc. of the 25th Int'l Conf. on Principles and Practice of Constraint Programming. Stamford: Springer, 2019. 600–617. [doi: 10.1007/978-3-030-30048-7_35]

[46] Sivaraman A, Sanchez-Stern A, Chen B, Lerner S, Millstein TD. Data-driven lemma synthesis for interactive proofs. Proc. of the ACM on Programming Languages, 2022, 6(OOPSLA2): 143. [doi: 10.1145/3563306]

[47] Sun YC, Ji RY, Fang J, Jiang XL, Chen MS, Xiong YF. Proving functional program equivalence via directed lemma synthesis. In: Proc. of the 26th Int'l Symp. Formal Methods. Milan: Springer, 2025. 538–557. [doi: 10.1007/978-3-031-71162-6_28]

[48] Reynolds A, Tinelli C, Goel A, Krstić S. Finite model finding in SMT. In: Proc. of the 25th Int'l Conf. Computer Aided Verification. Saint Petersburg: Springer, 2013. 640–655. [doi: 10.1007/978-3-642-39799-8_42]

[49] Reynolds A, Tinelli C, Goel A, Krstić S, Deters M, Barrett C. Quantifier instantiation techniques for finite model finding in SMT. In: Proc. of the 24th Int'l Conf. on Automated Deduction. Lake Placid: Springer, 2013. 377–391. [doi: 10.1007/978-3-642-38574-2_26]

[50] Reynolds A, Blanchette JC, Cruanes S, Tinelli C. Model finding for recursive functions in SMT. In: Proc. of the 8th Int'l Joint Conf. on Automated Reasoning. Coimbra: Springer, 2016. 133–151. [doi: 10.1007/978-3-319-40229-1_10]

[51] Reger G, Voronkov A. Induction in saturation-based proof search. In: Proc. of the 27th Int'l Conf. on Automated Deduction. Natal: Springer, 2019. 477–494. [doi: 10.1007/978-3-030-29436-6_28]

[52] Echenim M, Peltier N. Combining induction and saturation-based theorem proving. Journal of Automated Reasoning, 2020, 64(2): 253–294. [doi: 10.1007/s10817-019-09519-x]

[53] Hajdú M, Hozzová P, Kovács L, Schoisswohl J, Voronkov A. Induction with generalization in superposition reasoning. In: Proc. of the 13th Int'l Conf. on Intelligent Computer Mathematics. Bertinoro: Springer, 2020. 123–137. [doi: 10.1007/978-3-030-53518-6_8]

[54] Hajdu M, Hozzová P, Kovács L, Voronkov A. Induction with recursive definitions in superposition. In: Proc. of the 2021 Formal Methods in Computer Aided Design. New Haven: IEEE, 2021. 1–10. [doi: 10.34727/2021/isbn.978-3-85448-046-4_34]

[55] Hozzová P, Kovács L, Voronkov A. Integer induction in saturation. In: Proc. of the 28th Int'l Conf. on Automated Deduction. Springer, 2021. 361–377. [doi: 10.1007/978-3-030-79876-5_21]

[56] Hajdú M, Kovács L, Rawson M, Voronkov A. The Vampire approach to induction (short paper). In: Proc. of the 2022 Workshop on Practical Aspects of Automated Reasoning Co-located with the 11th Int'l Joint Conf. on Automated Reasoning. Haifa: CEUR Workshop Proc., 2022.

[57] Hajdu M. Automating inductive reasoning with recursive functions [Ph.D. Thesis]. Wien: Technische Universität Wien, 2020.

[58] Bradley AR. SAT-based model checking without unrolling. In: Proc. of the 12th Int'l Conf. on Verification, Model Checking, and Abstract Interpretation. Austin: Springer, 2011. 70–87. [doi: 10.1007/978-3-642-18275-4_7]

[59] Een N, Mishchenko A, Brayton R. Efficient implementation of property directed reachability. In: Proc. of the 2011 Int'l Conf. on Formal Methods in Computer-aided Design. Austin: FMCAD Inc, 2011. 125–134.

[60] Komuravelli A, Gurfinkel A, Chaki S, Clarke EM. Automatic abstraction in SMT-based unbounded software model checking. In: Proc. of the 25th Int'l Conf. Computer Aided Verification. Saint Petersburg: Springer, 2013. 846–862. [doi: 10.1007/978-3-642-39799-8_59]

[61] Komuravelli A, Gurfinkel A, Chaki S. SMT-based model checking for recursive programs. In: Proc. of the 26th Int'l Conf. on Computer Aided Verification. Vienna: Springer, 2014. 17–34. [doi: 10.1007/978-3-319-08867-9_2]

[62] Hojjat H, Rümmer P. The Eldarica Horn solver. In: Proc. of the 2018 Formal Methods in Computer Aided Design. Austin: IEEE, 2018. 1–7. [doi: 10.23919/FMCAD.2018.8603013]

[63] Gambhir S, Kunčak V. CHC solvers in stainless: Work in progress. 2024. https://www.sci.unich.it/hcvs24/papers/HCVS2024_paper_9.pdf

[64] Suter P, Dotta M, Kuncak V. Decision procedures for algebraic data types with abstractions. In: Proc. of the 37th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Madrid: ACM, 2010. 199–210. [doi: 10.1145/1706299.1706325]

[65] Pham TH, Whalen MW. An improved unrolling-based decision procedure for algebraic data types. In: Proc. of the 5th Int'l Conf. on Verified Software: Theorie, Tools, Experiments. Menlo Park: Springer, 2014. 129–148. [doi: 10.1007/978-3-642-54108-7_7]

[66] Pham TH, Gacek A, Whalen MW. Reasoning about algebraic data types with abstractions. Journal of Automated Reasoning, 2016, 57(4): 281–318. [doi: 10.1007/s10817-016-9368-2]

[67] Pham TH, Whalen MW. RADA: A tool for reasoning about algebraic data types with abstractions. In: Proc. of the 9th Joint Meeting on Foundations of Software Engineering. Saint Petersburg: ACM, 2013. 611–614. [doi: 10.1145/2491411.2494597]

[68] Voronkov A, Kovács L. Vampire team. 2025. https://vprover.github.io/team.html

[69] Reger G, Suda M, Voronkov A, Kovács L, Bhayat A, Gleiss B, Hajdu M, Hozzova P, Rath J, Rawson M, Schoisswohl J. Vampire 4.8—SMT system description. 2023. https://smt-comp.github.io/2023/system-descriptions/Vampire.pdf

[70] Rümmer P, Hojjat H, Kuncak V. Disjunctive interpolants for Horn-clause verification. In: Proc. of the 25th Int'l Conf. on Computer Aided Verification. Saint Petersburg: Springer, 2013. 347–363. [doi: 10.1007/978-3-642-39799-8_24]

[71] Ernst G, Morales JF. CHC-COMP 2024 report. 2024. https://chc-comp.github.io/2024/CHC-COMP%202024%20Report%20-%20HCSV.pdf

[72] Bjørner N. SPACER and Z3: Accessible, reliable model checking as theorem proving. 2018. https://www.microsoft.com/en-us/research/blog/spacer-and-z3-accessible-reliable-model-checking-as-theorem-proving/

[73] Feng WZ, Liu YC, Liu JX, Jansen DN, Zhang LJ, Wu ZL. Formally verifying arithmetic chisel designs for all bit widths at once. In: Proc. of the 61st ACM/IEEE Design Automation Conf. San Francisco: ACM, 2024. 213. [doi: 10.1145/3649329.3657311]

[74] Rosen KH. Discrete Mathematics and Its Applications. 8th ed., New York: McGraw-Hill, 2019.

作者简介

冯维直, 博士生, CCF 学生会员, 主要研究领域为形式化方法, 计算机软硬件验证.

刘嘉祥, 博士, 副研究员, CCF 专业会员, 主要研究领域为形式化方法, 程序验证, 神经网络验证.

张立军, 博士, 研究员, 主要研究领域为概率模型检测, 协议验证, 学习算法, 自动驾驶系统验证.

吴志林, 博士, 研究员, CCF 高级会员, 主要研究领域为计算逻辑, 自动机理论, 计算机软硬件基础设施的形式化验证.