软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(8):3604−3636 [doi: 10.13328/j.cnki.jos.007353] [CSTR: 32375.14.jos.007353]

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

面向 Rust 语言的形式化验证方法研究综述*

张卓若, 常 瑞, 杨申毅, 陈 芳

(浙江大学 计算机科学与技术学院, 浙江 杭州 310027)

通信作者: 常瑞, E-mail: crix1021@zju.edu.cn

摘 要: Rust 作为一种新兴的安全系统级编程语言, 以其创新的所有权模型和借用检查机制提供了内存安全和并发安全保证. 尽管 Rust 的设计宗旨在于安全性, 但现有研究揭示了其仍面临诸多安全挑战. 形式化验证作为一种基于严格数学基础的方法, 为 Rust 安全性提升提供了强有力保障. 通过构建精准清晰的语义模型, 可以证明遵循 Rust 检查规则的程序满足安全性要求; 借助 Rust 自动化验证工具能够帮助用户确保其 Rust 程序的安全性和正确性. 对 Rust 形式化验证工作进行全面系统性分析. 首先介绍 Rust 核心语义和复杂特性, 并探讨 Rust 形式化语义的研究与验证工作, 强调 Rust 类型系统在形式化验证中的潜力. 其次, 阐述面向 Rust 程序的自动化验证方法, 并对比分析不同验证工具的功能、支持的语言特性、采用的验证技术和适用场景, 这对于在 Rust 项目实际开发流程中指导工具的选择和集成有重要意义. 此外, 还总结 Rust 程序验证的典型实例, 展示形式化验证在确保程序正确性方面的显著成效, 并结合验证实例总结工具使用建议供用户参考. 最后讨论当前领域的技术挑战, 并指出未来可能的研究方向, 涵盖了 unsafe Rust 代码的验证、并发代码的验证、可信编译, 以及大模型驱动的形式化验证等. 旨在为 Rust 社区提供坚实的安全基础, 并推动形式化验证在 Rust 开发中的应用.

关键词: 形式化方法; Rust 语言; 程序验证; 形式语义; 内存安全

中图法分类号: TP311

中文引用格式: 张卓若, 常瑞, 杨申毅, 陈芳. 面向Rust语言的形式化验证方法研究综述. 软件学报, 2025, 36(8): 3604–3636. http://www.jos.org.cn/1000-9825/7353.htm

英文引用格式: Zhang ZR, Chang R, Yang SY, Chen F. Survey on Formal Verification Research for Rust Language. Ruan Jian Xue Bao/Journal of Software, 2025, 36(8): 3604–3636 (in Chinese). http://www.jos.org.cn/1000-9825/7353.htm

Survey on Formal Verification Research for Rust Language

ZHANG Zhuo-Ruo, CHANG Rui, YANG Shen-Yi, CHEN Fang

(College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China)

Abstract: As an emerging system programming language with a focus on safety, Rust ensures memory and concurrency safety through its innovative ownership model and borrowing mechanism. Despite its design for safety, existing research has identified several safety challenges that Rust still faces. Formal verification, grounded in rigorous mathematical principles, provides robust assurances for enhancing Rust's security. By constructing precise and well-defined semantic models, it becomes possible to formally prove that programs adhering to Rust's type system meet safety requirements. Automated verification tools for Rust further enable users to validate the safety and correctness of their programs. This study presents a comprehensive and systematic analysis of formal verification approaches for Rust. It begins by introducing Rust's core semantics and complex features, followed by an exploration of research and verification efforts in Rust's formal semantics, emphasizing the role and potential of Rust's type system in formal verification. Next, the study systematically compares the capabilities, supported language features, underlying techniques, and application scenarios of various automated Rust verification tools. These comparisons are instrumental in guiding tool selection and integration within real-world Rust development workflows. In addition,

* 基金项目: 国家重点研发计划 (2022YFB4501903)
本文由"形式化方法与应用"专题特约编辑陈明帅研究员、田聪教授、熊英飞副教授推荐.
收稿时间: 2024-08-26; 修改时间: 2024-10-14; 采用时间: 2024-11-26; jos 在线出版时间: 2024-12-10
CNKI 网络首发时间: 2025-04-03

this study summarizes exemplary cases of verified Rust programs, demonstrating the significant impact of formal verification in ensuring program correctness and providing practical tool usage recommendations for developers. Finally, it discusses the key challenges in the field and outlines promising directions for future research, including the verification of unsafe Rust code, concurrent code verification, trustworthy compilation, and large model-driven formal verification. This study aims to establish a strong security foundation for the Rust community and foster the broader adoption of formal verification methods in Rust development.

Key words: formal method; Rust language; program verification; formal semantics; memory safety

1 引 言

Rust 语言作为新一代安全高性能系统级编程语言, 旨在解决 C/C++等传统底层编程语言在内存安全和并发性方面的挑战. 它通过引入所有权模型, 实现了一种静态的自动内存管理机制, 同时结合函数式编程范式与强多态类型系统等先进语言特性, 不仅确保了内存和并发安全, 同时也达到了高效的执行性能, 满足了构建底层基础软件的需求. Rust 通过编译时的检查自动管理内存安全, 有效预防了空指针解引用、悬挂指针和内存泄漏等常见问题. 在 Rust 中, 每个值都拥有一个明确的唯一所有者, 当该所有者超出作用域时, 值会被自动清理. 此外, 借用规则保证了在任何给定时间, 要么只有一个可变引用存在, 要么可以有多个不可变引用, 但两者不会同时存在. 上述的设计哲学避免了数据竞争, 为并发编程提供了天然保障. 与 C/C++的手动内存管理相比, Rust 的所有权系统实现了自动化的内存管理, 同时避免了 Java 垃圾回收机制可能带来的性能开销, 实现了接近系统级语言的性能表现. 因此, Rust 已被广泛应用于基础软件的构建, 包括操作系统内核[1]、文件系统[2]、云服务[3]、数据库[4]、Web 浏览器[5]、网络协议栈[6]以及区块链技术[7]等多个领域, 证明了其在现代系统级编程中的实用性和可靠性.

然而, Rust 并不是绝对安全的. Rust 允许开发者通过 unsafe 代码块来执行解引用裸指针等底层操作以及跨语言调用等. Unsafe 代码的使用引入了潜在的风险, 它意味着绕过了 Rust 编译器强制性的内存安全检查, 将这部分代码的正确性交由开发者自行保证. 在复杂的系统编程任务中, 人为错误几乎不可避免, 很可能导致安全漏洞的产生. Xu 等人[8]收集了 186 个 Rust 相关的 CVE, 总结出 Rust 中 3 类典型内存安全问题, 指出都与 unsafe 特性的使用有关. Mergendahl 等人[9]揭示了 Rust 通过 unsafe FFI 与 C 代码的交互会引入新的攻击变体. 此外, 尽管 Rust 的安全性得到了社区和学术界的广泛认可, 但一些研究也揭示了即使是在 Rust 中也存在安全漏洞. 例如, 根据 system-pclub[10]的研究, Rust 库中仍然发现了一些内存安全和线程安全缺陷. 这些缺陷的存在表明, 尽管 Rust 的安全机制显著降低了编程错误的发生概率, 但并不能完全消除所有错误. 为应对这些安全问题, Rust 提供了运行时检测工具, 能够检测缓冲区溢出、除零等问题. 此外, Rust 在其调试构建模式中提供了更多的漏洞检测功能, 包括对双锁 (double lock) 和整数溢出的检测. 然而, 这些动态检测机制只能捕获有限范围的问题. 虽然 Rust 使用 LLVM 作为其后端, 许多为 C/C++设计的静态和动态漏洞检测技术也可应用于 Rust, 但 Rust 的新语言特性和库可能会引发新类型的漏洞. 近年来, 针对 Rust 的静态分析器得到了发展, 但这些工具在实际应用中存在较高的误报率[11]. 例如, SafeDrop[12]在分析大规模程序时的误报率可能高达 94%–97%.

因此, 在 Rust 语言的安全性增强方面, 基于形式化验证的研究工作已有显著成效. 形式化验证方法可基于数理逻辑基础对程序和系统进行精确建模和严格证明, 从而保证 Rust 程序的正确性和安全性. 一方面是针对 Rust 形式化语义的研究工作, 构建精准清晰的语义模型, 基于该语义模型可以证明遵循 Rust 所有权、借用检查和生命周期规则的程序确实满足内存安全性和线程安全性. 例如, RustBelt[13]项目利用其语义模型和证明, 在 Rust 编译器和标准库的实现中发现并修复了安全问题. 另一方面是 Rust 程序的自动化验证工作, 借助 Rust 自动化验证工具能够自动构建证明和检查过程, 帮助用户确保其 Rust 实现的程序是安全且正确的. 例如, AWS 通过采用轻量级形式化验证方法, 对其 Amazon S3 云对象存储服务中 Rust 实现的键值存储节点进行了功能正确性验证[14], 成功拦截 16 个潜在问题, 防止其进入生产环境. 此外, 形式化验证方法在增强 unsafe Rust 代码的安全性上也有显著成果[13,15,16].

虽然已有研究在特定领域有一定进展, 但目前尚缺乏从系统性的角度对 Rust 形式化验证工作的全面分析. Rust 作为一种持续演进的新兴编程语言, 其多种语言特性都在不断完善. 比如借用规则这一核心机制近年来的发展

变化, 从最初的词法生命周期 (lexical lifetime) 到非词法生命周期 (non-lexical lifetime, NLL), 再到最新的 Polonius 借用检查器, 使得形式化语义工作也随着 Rust 语言的发展而不断变化. 但目前缺乏对不同语义研究工作的系统性对比分析. 此外, 现有的 Rust 验证工具多由不同团队独自开发, 缺少统一框架来比较它们的功能、支持的语言特性和适用场景. 这影响了这些工具在实际开发流程中的集成和应用, 也限制了它们在技术创新上的相互促进. 因此, 本文的工作旨在填补这一空白. 我们将首先从理论上系统性分析 Rust 语言特性, 然后从 Rust 语义研究的角度探讨形式化验证如何增强 Rust 的安全性, 接着分类讨论了 Rust 程序的自动化验证方法及工具, 最后总结了当前 Rust 形式化验证工作存在的挑战和局限性, 并提出未来可能的改进方向. 本文的工作对于指导工具选择、促进技术交流与创新, 以及推动形式化验证技术的发展具有重要意义. 特别说明的是, 本文系统性分析 Rust 语言的基于所有权和借用检查的强大类型系统, 为形式化验证提供了新的视角. Rust 类型系统扩展了编程语言理论中线性类型、仿射类型的概念, 形式化领域可以通过证明类型系统可靠性的方式来证明 Rust 类型安全和借用安全. Rust 在语言层面上对指针别名实行了严格的规范, 内置了类似分离逻辑的思想. 多种 Rust 验证工具已经巧妙地利用其类型系统提供的安全保证来简化程序证明的过程. 通过本文的工作, 我们希望为 Rust 社区提供一个更加坚实的安全基础, 并为未来的研究指明方向, 推动形式化验证方法在 Rust 开发中的更广泛应用.

本文的核心议题聚焦于 Rust 语言的形式化验证技术. 该技术依赖于严格的数学逻辑和形式系统, 基于形式语义和形式规约来描述系统的行为和属性, 将系统的分析和验证问题转化为逻辑推理问题或形式模型的判定问题, 并利用定理证明器、模型检查器等形式化工具来进行严格的证明, 确保系统的行为与预期一致. 相比之下, 尽管污点分析、数据流分析、动态分析等常用程序分析方法在 Rust 的漏洞检测研究中占有重要地位, 但这些方法不强制执行严格的形式验证, 可能存在误报或漏报问题. 因此, 这些程序分析技术超出了本文的讨论范围.

本文在第 2 节系统性分析 Rust 语言的特性, 包括以所有权和借用检查机制为核心的特性以及一些复杂特性, 并阐述 Rust 编译的过程. 第 3 节深入探讨针对 Rust 形式语义的研究与验证工作, 包括基于类型系统的语义研究以及基于纯操作语义的语义研究. 第 4 节全面对比分析各类 Rust 自动化验证工具, 这些工具采用了多种验证方法, 包括半自动化程序验证方法、定理证明方法和模型检测方法. 第 5 节对已验证的 Rust 系统进行总结, 展示形式化验证在确保 Rust 程序正确性方面的实际应用及效果, 并综合分析验证案例和实际使用经验, 提出工具选择建议供用户参考. 第 6 节分析当前领域面临的多维度挑战, 这不仅需要验证人员的努力, 还需要与 Rust 社区和开发者的共同协作. 第 7 节探讨值得进一步探索研究的发展方向, 包括对 unsafe Rust 代码的验证, 对并发代码的验证, Rust 可信编译以及大模型驱动的 Rust 形式化验证等.

2 Rust 语言特性概述

2.1 Rust 核心特性

2.1.1 所有权机制

所有权是 Rust 语言为安全高效使用内存而设计的语法机制. 大多数的编程语言都有管理内存的功能, C/C++ 主要通过手动方式管理内存, 开发者需要手动的申请和释放内存资源, 但许多开发者没有及时释放内存的习惯, 所以手动管理内存的方式常常造成资源浪费. 传统的自动内存管理语言通过垃圾回收机制管理内存对象的生命周期, 典型代表有 Java、Python、Go 等. Java 语言编写的程序在虚拟机 (JVM) 中运行, JVM 具备自动回收内存资源的功能. 但垃圾回收期间通常会造成程序短暂停顿而降低运行时效率, 所以实际使用中一般会通过 JVM 参数减小垃圾回收的频率, 这样也会使程序占用较大的内存资源. 而 Rust 最具标志性的所有权模型通过编译时检查实现了自动内存管理, 从而避免了运行时的内存管理开销和潜在的内存安全问题. 所有权模型的设计其根本而言是为了管理堆上的数据, 所有权机制能跟踪代码的哪些部分正在使用堆上的哪些数据, 清理堆上未使用的数据以避免空间不足.

Rust 的所有权模型有以下 3 条规则: (1) Rust 中的每一个值都有一个所有者 (owner); (2) 值在任一时刻有且只有一个所有者; (3) 当所有者 (变量) 离开作用域 (scope), 这个值将被丢弃. 变量范围是变量的一个属性, 其代表变

量的可行域和作用域, 默认从声明变量开始有效直到变量所在域结束.

基于上述所有权规则, Rust 针对不同数据类型实现了相应的内存管理策略. 具体来说, 栈基础数据类型 (整型、浮点型等) 赋值时自动进行值拷贝 (copy), 因此原变量保持有效. 而堆分配类型 (如 String、Vec<T>) 默认使用移动 (move) 语义. 如图 1 所示, 赋值操作会转移所有权并使原变量失效. 如需确保堆分配变量在赋值操作后保持双重有效性, 需显式执行堆数据的深拷贝 (clone). 智能指针 Box<T> 针对其指向的分配在堆上的 T 类型数据实现内存的独占式管理, 当 Box<T> 的所有权被移动或其作用域结束时, 会自动调用析构函数释放堆上分配的内存, 有效防止内存泄漏. 此外, 当变量作为参数传入函数时, 它遵循与移动操作相同的原则, 即所有权转移到函数中. 相应地, 如果一个变量作为函数的返回值, 其所有权会随着返回语句从函数中移动出来, 转移到调用者的作用域.

```
1  let x = String::from("hello"); // x 是 String 类型所有者
2  let y = x; // x 的所有权被转移给了 y
3  // println!("{}", x); // 编译错误: x 不再有效
4  println!("{}", y); // 正常: y 是 String 的新所有者
```

图 1 Rust 中的所有权机制

2.1.2 借用检查

所有权系统为 Rust 提供了内存管理的基础, 但有时我们需要引用一个值而不转移所有权. Rust 通过借用 (borrowing) 机制实现了这一点. 在 Rust 中, 借用有两种类型: 不可变借用 (immutable borrow) 和可变借用 (mutable borrow). 借用机制允许一个值在同一时间被多个地方使用, 但同时只能有一个可变借用或者多个不可变借用, 且所有借用必须在其创建的作用域结束之前有效, 以此保证了数据的一致性, 防止了数据竞争和悬挂指针. 该程序属性由 Rust 的借用检查器 (borrow checker) 保证, 不满足该属性的程序会被拒绝通过编译. 如图 2, r1 和 r2 是 String 对象的不可变借用者, r3 是 String 对象的可变借用者, 当第 6 行试图同时使用 r1, r2 和 r3 时, 编译器会报错, 因为在同一时间不允许既有不可变借用又有可变借用.

```
1  let mut s = String::from("Hello");
2  let r1 = &s; //不可变借用
3  let r2 = &s; // 另一个不可变借用
4  println!("{} and {}", r1, r2); // 同时使用两个不可变借用
5  let r3 = &mut s; //词法生命周期下报错: 同时存在不可变借用和可变借用; 非词法不报错
6  //println!("{},{} and {}", r1, r2, r3); //(词法和非词法下均)报错: 同时存在不可变借用和可变借用
7   println!("{}", r3);
```

图 2 Rust 中的借用规则例 1

Rust 借用检查器在以下条件同时成立时, 会判定程序中的某个语句违反了借用检查规则: (1) 该语句访问了某个内存对象. Rust 中将内存对象表示为路径 (path) P 的形式, 如局部变量 a, 访问字段 a.b, 解引用操作*a.b; (2) 访问路径 P 违反了由某个借用表达式产生的借用 L (loan); (3) 借用 L 此时是活跃 (live) 的, 意味着 L 未来将被用以访问其借用的内存对象.

例如, 在图 3 中所示的悬垂指针问题中, 涉及的事件序列如下: 在第 5 行, 字符串变量 s 退出其作用域, 编译器自动插入的 drop 函数随之释放了与之关联的内存对象, 这一过程相当于对路径 s 执行了一次写操作. 在第 4 行, 一个借用表达式创建了一个借用 L0, 该借用是对 s 的不可变借用. 根据 Rust 的借用规则, 写入 s 与借用 L0 是冲突的, 因为在不可变借用的有效期内, 不允许对借用的内存对象进行修改. 到了第 7 行, 由于 L0 被使用, 这表明在第 5 行执行 drop(s) 时, L0 仍然是活跃的. 因此, 借用检查器发现了这一借用违规, 并报告了错误.

给定一个程序, 能够确定哪些语句将访问内存对象, 以及这些访问操作与借用规则之间的冲突. 关键问题在于准确推断各个借用的活跃区域, 即 Rust 语言中所谓的生命周期. Rust 通常能够通过借用检查器自动推断生命周

期, 但在一些复杂的场景中, 开发者需要显式地标注生命周期. 生命周期标注通常以撇号 ('a, 'b) 的形式表示, 并且出现在引用的类型声明中. Rust 的借用检查器通过建立约束规则来确定借用之间的活跃区域关系, 并求解这些约束以找到最小活跃区域. 这种静态分析方法得到的活跃区域是对实际情况的过度近似, 目的是确保只有遵守借用规则的程序才能被编译通过, 这同时也导致了一些实际上正确的程序可能被错误地拒绝.

```
1  let r: & String;
2  {
3      let mut s = String::from("hello");
4      r = & s; // r 借用 s
5      /* drop(s) 由编译器自动插入*/  // s 离开其作用域, 值被回收, 悬垂指针编译报错
6  }
7  println!("{}", r); // 引用 r 在这里继续使用
```

图 3 Rust 中的借用规则例 2

Rust 借用检查器的发展经历了 3 个阶段: 从早期的词法生命周期, 到目前采用的非词法生命周期, 以及即将引入的 Polonius. 每一个阶段的进步都旨在提高活跃区域推断的精确度, 从而减少对合法程序的误报, 使得更多符合规范的程序得以成功编译.

词法生命周期基于源代码的词法作用域 (lexical scope) 来推断生命周期. 在词法生命周期下, Rust 编译器严格按照代码块的结构来决定引用的有效范围. 这种推断方式虽然简单直接, 但过于保守. 如图 2 中, 第 2 行的借用表达式创建了路径 s 的不可变引用 L0, 并且赋给了 r1, 在词法生命周期的推断下, L0 的生命周期被认为直到代码块结束前都是活跃的, 因此在第 5 行试图创建可变引用 r3 时与 L0 相悖, 编译器会报错. 虽然在实际运行中 r1 在第 4 行之后不再被使用.

为了改善词法生命周期的保守性, Rust 团队引入了非词法生命周期推断 (NLL). 在 NLL 模式下, 不再依靠词法作用域来推断借用的生命周期, 而是基于程序的控制流图. 即, 借用的生命周期被表示为控制流图上程序点 (program point) 的集合. 如果一个生命周期包含了点 P, 这意味着具有该生命周期的引用在进入 P 时是有效的. 如图 4 所示, 右边是一个示例程序, 左边是对应的简化了的该程序控制流图. 首先通过活跃变量分析可以得到引用变量 p 的生命周期'p 为'foo 和'bar. 右图第 4 行程序语句生成约束'foo: 'p @ A1, 这表示在程序点 A1 处, 'foo 会包含 'p 从 A1 点可达的程序点. 于是, 'foo={A1, B0, C0}, 同理, 由第 7 行生成约束'bar: 'p @ B2 得到'bar={B2, C0}. 根据非词法生命周期, 图的示例可以不需增加词法块直接通过借用检查. 在 NLL 的推断下, 面对同样图 2 的代码编译器会发现 r1 和 r2 在第 4 行之后不再被使用, 因此可以在该点结束 r1 和 r2 的生命周期, 从而允许创建 r3 的可变借用, 这种改进极大地提升了 Rust 代码的灵活性和易用性.

Polonius[17]是 Rust 未来的借用检查引擎, 它不再使用生命周期, 即为每一个借用计算出可能覆盖的程序点, 而是去维护一个借用来源 (Origin) 的集合 (后文用“区域 (Region)”指代), 直接反映数据流向关系. 如果一个区域包含了某个借用并且该区域根据活跃变量分析在程序点 P 处活跃, 就判定该借用在该程序点是活跃的. 对于图 4 的程序, 其右图第 4 行和第 7 行程序语句分别创建了两个借用 L0 和 L1, 区域为'foo={L0}和'bar={L1}, 根据第 4 行生成约束'foo: 'p @ A1, 这表示'foo 是'p 的子集, 于是'p={L0}. 第 7 行处的约束使'p='p∪{L1}−{L0}={L1}, 并最终在 C0 处合并为'p={L0, L1}, 所以 L0 在{A1, B0, C0}活跃, L1 在{B2, C0} 活跃, 这和非词法生命周期是一致的. Polonius 不仅可以接受所有 NLL 可以接受的程序, 对于其保守拒绝的一些情况也能有效处理, 如图 5 所示的返回借用, 在非词法生命周期下, 由于行 3 的 v 会被返回给调用者函数, 其生命周期会包含 get_or_insert 函数所有的程序点, 导致行 2 的不可变借用 L0 的生命周期也覆盖了 None 分支里面的所有程序点, 于是行 5 创建可变借用的时候就会冲突报错; 而在 Polonius 的视角, v 的区域在 None 分支并不活跃, 所以 L0 在 None 分支里也不是活跃的, 不会冲突. Polonius 获取到程序相关事实 (facts) 后, 会转交给 Datalog 来做推理, 以提升效率. Polonius 已集成到 nightly rustc 中, 可以使用该-Zpolonius 标志运行试用.

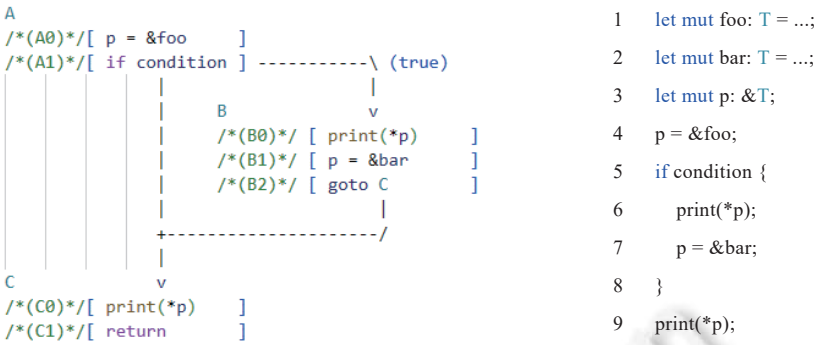

图 4 Rust 代码例子和程序流图

```
1  fn get_or_insert(map: & mut HashMap<u32, String>) -> &String {
2    match HashMap::get(&*map, &22){
3      Some(v) => {v}
4      None => {
5        HashMap::insert(&mut *map, 22, String::from("hi"));
6        &map[&22]
7      }}}
```

图 5 Polonius 分析复杂借用

实际上, Rust 提供了多种借用形式, 每种形式都有其特定的用途和规则, 适用于不同的具体情况. 例如, 返回借用 (return borrows) 允许函数返回对传入引用参数的借用, 但要求借用的生命周期至少与返回值一样长; 嵌套借用 (nested borrows) 即一个借用内部又包含了对另一个值的借用的情况, 这要求内部借用的生命周期不超过外部借用; 两阶段借用 (two-phase borrows) 允许在不同时间点进行可变借用和不可变借用, 但不允许同时存在, 以避免数据竞争等问题. 这些借用形式共同构成了 Rust 灵活而强大的借用检查系统.

2.2 Rust 复杂特性

2.2.1 unsafe Rust 代码

Rust 作为一种现代的系统编程语言, 在设计上除了考虑内存安全保证外, 同时还需要保持高性能. 尽管 Rust 的大部分代码都是用所谓安全子集 (safe subset) 编写的, 并通过 Rust 所有权系统和借用检查器提供强大的内存安全保证, 但在某些情况下, 开发者需要绕过这些安全检查以直接访问底层系统资源或优化性能. 为此, Rust 允许开发者在明确了解潜在风险的情况下编写 unsafe 代码块, 与底层操作系统交互或提供低级别的抽象, 并获得一些超越常规 Rust 安全保证的特殊能力. Unsafe 代码块主要允许以下几种操作: 解引用原始指针 (raw pointer)、调用 unsafe 函数或方法 (包括外部 C/C++函数)、访问或修改可变静态变量、实现非安全特质 (unsafe traits)、读写 union 联合体中的字段. Unsafe 代码通常用于性能优化、低级别操作和与其他语言的接口等场景.

Unsafe 代码是 Rust 设计中不可或缺的一部分, 它使得安全抽象能够在库中高效实现. 例如双向链表类型的实现就需要利用 unsafe 代码来达到高效的内存管理和操作. 然而, 对 unsafe 机制的不当使用容易导致安全漏洞, 形成对 Rust 程序的新攻击面. 如图 6 的例子是使用 unsafe Rust 的典型示例. 原始指针提供了类似于 C 语言中指针的操作能力, 例如进行指针算术. 与 Rust 中的引用不同, 原始指针不会被借用检查器所跟踪. 在 Rust 中, 解引用原始指针是一种不安全的操作, 因为对于原始指针, 类型系统无法确保它们是否指向了有效的、可以安全访问的内存位置. 如果原始指针指向的内存不是有效的, 或者该内存没有被正确初始化, 那么尝试解引用它将可能导致程序崩溃, 例如触发段错误 (segmentation fault). 图 6 的第 5 行会因越界访问导致未定义行为 (undefined behaviour), 但由于 unsafe 代码可以绕过 Rust 编译器的检查, 因此不会报错.

```
let data = 0u32; // 声明一个 u32 整数
let data_ref = &data as *const u32; // 将数据的地址转换为不可变原始指针
unsafe {// 通过原始指针访问字节
    let first_byte = *data_ref.offset(200) as u8; // 越界的字节访问
    println!("First byte: {}", first_byte); // 输出第 1 个字节的值
}
```

图 6 Unsafe 代码的安全隐患

除了原始指针之外, 另一个容易出现安全问题的原因是 unsafe 块中使用 FFI 调用外部函数, 尤其是在涉及指针、内存管理或外部库函数行为不确定的情况下可能会引入不安全的操作, 如果不小心处理, 可能会导致内存泄漏、悬空指针、数据竞争或其他未定义行为. 假设从 C 库中调用一个函数返回一个指向动态分配的内存的指针, 该函数期望调用者负责释放这块内存, 如果 Rust 代码未能正确释放内存, 或者在释放后继续使用该指针, 可能会引发未定义行为.

为了避免上述问题, 程序员有责任确保 unsafe 代码不会表现出未定义行为. 形式化验证可以帮助开发者理解代码的行为, 并通过验证 unsafe 代码满足特定的规范和属性以提供额外的保证, 确保其不会引入漏洞或安全风险. 此外, 尽管 unsafe 代码在总代码库中所占比例不大, 但它们往往是 Rust 程序中最复杂、最容易出错的部分. 因此, unsafe 代码也是最迫切需要进行形式化验证的.

2.2.2 Rust 多态性

多态性是面向对象编程语言中的一个核心概念, 它允许编写的代码更加通用和灵活. Rust 语言的多态性主要通过特质 (traits) 和泛型 (generics) 实现. Rust 中的泛型允许函数、结构体、枚举和 traits 定义可以处理不同类型的数据, 而无需为每种类型编写单独的代码, 使代码更加通用和可复用. 泛型通过参数化类型来实现多态性, 如图 7 中 largest 函数使用了泛型 T, 使得该函数可以接受任何实现了 PartialOrd 特征的类型的切片. 这样无论是整数列表还是字符列表, 函数都可以由同一个 largest 函数处理.

```
fn largest<T: PartialOrd>(list: &[T]) -> &T {
    let mut largest = &list[0];
    for item in list {
        if item > largest {largest = item;}
    }
    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];
    let result = largest(&number_list);
    let char_list = vec!['y', 'm', 'a', 'q'];
    let result = largest(&char_list);
}
```

图 7 Rust 中的泛型实例

Rust 的 trait 类似于其他面向对象语言 (如 Java、C#) 中的接口 (interface), 它定义了一组方法签名, 而具体的实现则由结构体或枚举来完成. Rust 的 trait 可以在静态和动态两种方式下使用. 静态调度 (static dispatch) 是在编译时确定实际调用的方法, 编译器在编译时会为每个具体的类型生成相应的代码, 能获得更好的性能, Rust 通过泛型和 trait bounds 实现静态调度. 在 Java 和 C#中接口的实现通常都是通过动态调度 (dynamic dispatch) 完成的, 方法调用是在运行时根据对象的实际类型来决定的, Rust 通过特征对象 (trait objects) 使用 dyn Trait 关键字支持动态调度, 在运行时选择具体的实现, 从而实现接口的封装 (interface encapsulation). Rust 中的关联类型 (associated types) 是一种在 trait 中定义类型占位符的机制, 这种占位符可以在实现 trait 时被具体类型所替换, 能简化 trait 的使用, 使代码更加清晰和易于理解. 与泛型相比, 关联类型让 trait 实现更为简洁, 并且在某些情况下, 关联类型比泛型更适合表达类型间的关系.

接口在 Java、C#等语言中可以支持多继承, 一个类可以实现多个接口, 这样该类就可以被当作多种类型使用, trait 在 Rust 中也支持多重 trait 实现, Rust 允许一个类型实现多个 trait, 这类似于 Java 或 C#中实现多个接口, 但

Rust 进一步通过组合和默认实现来增强了 trait 的灵活性, Rust 高级 trait 绑定 (advanced trait bounds) 支持更复杂的 trait 约束组合如多个 trait 的联合约束、特定生命周期的 trait 约束, 以及关联类型的 trait 约束. Rust 的 trait 与接口相比其显著的优势在于其与类型系统的深度集成, 尤其是在所有权、借用和生命周期等 Rust 特有的安全机制中. Rust 编译器会在编译时对 trait 的实现进行严格的类型检查, 确保所有权和借用规则不被违反, 这使得 Rust 的 trait 不仅是一种多态性实现的手段, 更是保证内存安全和并发安全的重要工具.

2.2.3 Rust 并发安全

Rust 在设计时充分考虑了并发编程的安全性和性能. 所有权系统是 Rust 并发编程的基础, 通过所有权、借用、生命周期的设计 Rust 能够在编译时确保数据的一致性与安全性. Rust 的借用检查器在编译期确保只有一个可变借用或多个不可变借用, 从而避免数据竞争. Rust 编译器通过自动实现 Send 和 Sync 这两个 traits 来帮助开发者确保并发安全, Send 表示类型可以在线程间安全地传递, Sync 表示类型可以安全地在多个线程中同时访问. Rust 的标准库中还提供了多种并发原语如线程 (std::thread)、消息传递 (std::sync::mpsc)、互斥锁 (std::sync::Mutex) 与读写锁 (std::sync::RwLock).

Rust 最初的并发模型依赖于操作系统线程和标准库中基本的并发原语, 如 Mutex 和 mpsc 通道. 这种模型非常强大, 但在高并发场景下, 尤其是需要大量线程或需要避免线程切换开销的场景下, 可能会遇到性能瓶颈. 为了应对现代系统中广泛存在的 IO 绑定 (I/O-bound) 任务, Rust 引入了异步编程模型 (asynchronous programming), Rust 的异步编程基于 Future trait, 它表示一个可能未完成的计算结果, 通过 poll 方法, Future 可以逐步推进, 直到完成. 在 Rust 1.39 版本中, async/await 语法被稳定引入, 这使得异步代码的编写更加直观和易于理解. 在异步编程中, Pin 和 Unpin 类型用于确保异步任务在内存中不会被移动, 从而防止数据竞争和内存安全问题. Rust 标准库本身不提供异步运行时或执行器 (executor), 开发者通常使用第三方库 (如 tokio 或 async-std) 来执行异步任务, 这些库提供了基于事件驱动的执行模型, 能够高效处理大量并发任务.

Rust 在并发编程中的安全性主要来自于其所有权系统和编译时的静态分析. 通过这些机制, Rust 在编译时防止了许多潜在的并发错误, 如数据竞争和悬空指针, 提供了强大的静态安全性保证. 然而, 并发编程在涉及复杂的生命周期、互斥锁的死锁或异步任务的协作等复杂的并发场景中仍然充满挑战, 需要开发者小心设计和实现以避免逻辑错误. 形式化验证可以通过模型检查等技术在编译器的静态检查之外进一步确保并发程序的安全性和正确性, 这对于构建高可靠性、高性能的并发系统尤为重要.

2.3 Rust 编译过程

Rustc 作为 Rust 编程语言的官方编译器, 承担着将 Rust 源代码转换为可执行程序或库文件的关键角色. 虽然编译过程可能会因编译器的具体版本和所选优化等级而有所变化, 但其基本步骤可概括为如图 8 所示的编译过程. Rust 源代码首先被解析成抽象语法树 (AST). 在这个阶段, 编译器首先进行词法分析, 将源代码文本分解成一系列的词素 (tokens), 例如关键字、标识符、操作符等. 然后, 编译器进行语法分析, 根据 Rust 语言的语法规则, 将这些 tokens 组合成 AST. AST 是源代码的树状结构表示, 反映了代码的语法结构. 该阶段中也会进行宏展开, 名称解析等步骤. 接下来, AST 会被转换成 HIR (high-level intermediate representation), 这是一种对编译器更友好的 AST 表示法. 这个过程被称为降级 (lowering), 其中涉及大量的去糖 (desugar), 如 for 循环可能会被转换为迭代器的调用, async 函数可能会被转换为状态机. 然后, 使用 HIR 进行类型推断 (自动检测表达式类型的过程)、trait 求解 (将 impl 与 trait 的每个引用配对的过程) 和类型检查. 类型检查是将 HIR 中的类型 (代表用户编写的内容) 转换为编译器使用的内部表示的过程. 之所以称为类型检查, 是因为这些信息用于验证程序中使用的类型的安全性、正确性和一致性, 如检查变量的类型是否与预期一致, 以及确保类型转换不会引入错误. 在做类型检查后, 将 HIR 进一步降级为 MIR (mid-level intermediate representation). 该步骤会将函数中的语句及表达逻辑转换成控制流图 CFG, 精确捕捉程序的控制流和数据流, 以便对 Rust 的所有权和借用检查规则进行更深层次的优化. 在该过程中, 代码也会被单态化 (monomorphized), 即将泛型的类型参数替换为不同的具体类型, 复制相关泛型代码生成不同的函数. 同时, 还在 MIR 层做一些优化以提高后面代码生成和编译速度. 在 MIR 上进行 Rust 的静态分析借用检查.

此后, 据 MIR 内容及 LLVM IR 规范进行相应的逻辑转换, 生成 LLVM IR 代码, 再使用 LLVM 生成汇编代码, 最后进行编译链接生成二进制代码.

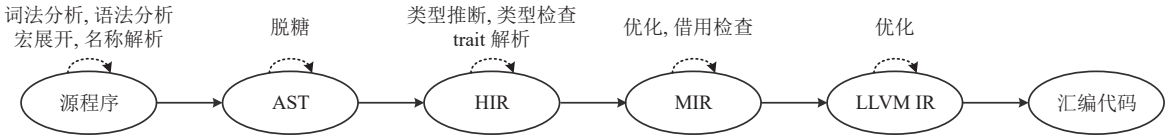

图 8 Rust 编译过程

值得一提的是, Rust 语言的表面 (surface) 语法富含大量的语法糖, 而对于这些语法, Rust 语言社区尚未给出一套正式的操作语义定义. 相比之下, MIR 显著简化了语言的复杂性. MIR 拥有一组精简的操作集, 通过去糖处理简化掉了 Rust 表面层的许多复杂特性, 例如循环和模式匹配语句, 且函数体以控制流图的形式清晰呈现. 同时, MIR 中仍然保留了 Rust 所有权和借用概念的核心, 这些概念在类型系统中占据着重要位置. 此外, MIR 还保留了泛型定义, 使得语言的通用性得以延续. 因此, 与复杂的表面 Rust 语法相比, MIR 提供了一个更为简洁和易于操作的环境, 同时保留了语言的关键特性. 同时, MIR 提供了丰富的应用程序接口 (API), 这些 API 允许开发者访问、提取和操作 Rust 程序的 MIR 表示. MIR 的这些特性使其不仅在编译过程中扮演着关键角色, 而且还是进行深入程序分析和逻辑推理的理想选择. 如今, MIR 已经成为形式化语义深入研究和精准高效验证工具开发的对象. 很多验证工具与 Rust 编译器的紧密集成, 在 Rust 编译器完成类型检查之后提取 MIR 及类型和借用检查信息, 再进行进一步的分析推理.

3 针对 Rust 形式语义的研究工作

类型系统作为 Rust 语言的基石, 对程序的正确性、安全性和效率起着至关重要的作用. 大多数对 Rust 的形式化语义研究工作捕捉了 Rust 类型系统的核心, 描述 Rust 类型推断和借用检查的过程, 并给出类型系统的可靠性证明, 表明遵循 Rust 所有权、借用和生命周期规则的程序是内存和线程安全的. 还有一些工作没有借助类型系统, 而是主要建模了 Rust 以内存管理为核心的操作语义来描述 Rust 的形式规范, 包括所有权系统、借用检查、生命周期规则等. 对 Rust 的形式化语义的研究工作提供了一种精准清晰的语义框架以便开发人员更深入地学习和探索这一新兴编程范式的机制和特性, 同时也为后续开发 Rust 程序验证工具提供了坚实的理论基础. 本节首先对类型系统的概念进行精炼的概述, 并特别聚焦于 Rust 语言的类型系统, 此后分别介绍基于 Rust 类型系统的语义研究工作和基于纯操作语义的 Rust 语义研究工作. 表 1 总结了不同形式语义模型研究工作的差别. 其中, “正确性保证”表示该工作是否使用了机械化的证明 (如借助 Coq 证明助手[18]证明) 确保语义模型正确性, “内存模型”表示该工作是否精准地刻画了 Rust 的内存模型.

表 1 Rust 形式语义模型研究工作总结对比

研究工作	正确保证	类型系统	数据结构	内存模型	生命周期	验证性质	支持语言特性			
							Unsafe	Closure	并发	Trait
RustBelt[13]	●	●	MIR	○	生命周期逻辑	并发、内存安全	●	○	●	○
RustHornBelt[19]	●	●	MIR	○	生命周期逻辑	并发、内存安全	●	○	●	○
RefinedRust[15]	●	●	MIR	●	生命周期逻辑	功能正确性	●	○	○	○
Oxide[20]	○	●	Rust	○	Polonius	功能正确性	○	●	○	○
Flux[21]	○	●	MIR	●	NLL	类型安全	○	●	○	○
FR[22]	○	●	Rust	○	词法	类型安全, 功能正确性	○	○	○	○
Patina[23]	○	●	Rust	○	词法	类型安全	○	○	○	○
StackedBorrows[24]	○	○	Rust	●	栈表示	编译优化, UB	●	○	○	○
KRust[25]	○	○	Rust	○	词法	功能正确性	○	○	○	○
RustSEM[26]	○	○	Rust	○	NLL	并发、内存安全	●	●	●	●

RustBelt 提出生命周期逻辑 (lifetime logic), 其支持很多 NLL 的示例, 但和 NLL 借用检查规则有所不同, 例如, 其对共享引用的处理更加严格.

3.1 Rust 类型系统

类型系统[27]是编程语言设计中的一个基础性概念, 它为程序中的数据提供了一种结构化的规范, 确保了数据操作的合理性和安全性. 在类型理论的框架下, 类型系统不仅定义了数据的静态结构, 还规定了数据在程序执行过程中的动态行为. 一个成熟且严谨的类型系统能够有效地预防诸如类型错误、内存泄漏、数组越界等常见的编程错误, 从而提高软件的可靠性和稳定性. 线性类型 (linear types) 和仿射类型 (affine types) 是编程语言理论中的概念, 它们在类型系统中引入了严格的约束. 线性类型系统强制要求每个资源只能被使用一次, 而仿射类型则进一步放宽了这一限制, 允许资源可以被使用一次或不被使用, 即允许资源在没有被使用的情况下可以安全地存在. 它们都规定一个值只能拥有一个所有者, 并且在所有者超出作用域时, 该值会被自动释放. 这样的设计宗旨在于确保资源在编译时得到安全和高效的管理, 预防内存双重释放、使用未初始化变量等常见错误. 这些概念在函数式编程语言中得到了广泛应用, 它们通过限制变量的使用次数来增强程序的表达能力, 同时保持了程序的安全性.

类型检查是类型系统的核心机制之一, 它在编译阶段对程序代码进行分析, 负责确保程序的类型安全. 类型安全是指程序在运行时不会出现违反类型规则的错误, 例如尝试将一个整数与字符串相加, 或者访问一个空指针. 类型检查通过静态分析源代码来预防这类错误, 从而避免了潜在的运行时异常或安全漏洞. 在类型检查过程中, 编译器会遍历程序的每个表达式, 验证其中的变量和操作是否符合它们的声明类型. 这包括检查变量的声明和使用是否一致, 确保函数调用时提供的参数类型与函数签名中定义的参数类型相匹配, 以及确保返回值的类型与预期的返回类型相符. 类型检查还涉及更复杂的场景, 如类型转换、类型推断 (type inference) 和泛型编程中的类型约束. 类型推断即通过类型系统中定义的类型规则 (typing rules) 推测出某表达式的类型. 在许多编程语言中, 编译器能够根据上下文自动推断出变量的类型, 减少了程序员显式声明类型的需要. 这种类型推断不仅简化了代码的编写, 而且有助于提高代码的可读性和可维护性, 同时也减少了因类型声明错误而导致的编译错误. 此外, 类型检查还与程序的优化紧密相关. 编译器利用类型信息来优化程序的执行效率, 例如通过消除冗余的类型转换, 或者应用特定类型的高效算法实现. 在某些情况下, 类型检查还能够揭示程序中的潜在性能瓶颈, 指导程序员进行代码优化. 总之, 类型检查为程序的正确性、安全性和性能提供了坚实的保障. 通过类型检查, 程序员可以在编写代码时获得即时的反馈, 提高了开发效率和代码质量.

Rust 类型系统是该语言的核心特性之一, 以其独特的设计和对安全性的强调而著称. Rust 语言并未直接采用线性类型或仿射类型的概念, 而是通过所有权和借用规则隐式达成了相似的编程目标, 为程序员提供了一种在编译时确保内存安全的方法. Rust 的所有权模型规定, 一个值在任何时候只能有一个所有者, 这个所有者负责管理该值的生命周期. 借用机制则允许在不转移所有权的情况下临时访问数据, 并通过借用规则保证该访问不会引发数据竞争和悬挂指针等问题. 在 Rust 中, 类型检查过程可以划分为两个阶段: 首先是传统的类型检查, 确保代码符合基本的类型安全规则; 其次是借用检查器的借用检查, 这一独特的机制确保程序符合所有权模型的规定和借用检查规则, 为数据的访问和修改提供了精细的控制. 这两个阶段的结合构成了 Rust 类型系统的基础, 有效预防了诸如使用后释放、双重释放以及数据竞争等普遍存在的编程错误, 保障了 Rust 代码的内存安全性, 同时避免了对垃圾收集器的依赖. Rust 类型系统的另一个显著特点是其对泛型 (generics) 和特质 (traits) 的支持. 泛型允许程序员编写与特定数据类型无关的通用代码, 而 traits 则定义了一组行为的契约, 确保了类型的多态性和可替换性. 这些特性使得 Rust 的类型系统不仅在安全性上表现出色, 而且在表达力上也具有显著优势.

3.2 基于类型系统的 Rust 语义研究

随着 Rust 语言的蓬勃发展和其在实际应用中的深入实践, 其核心的类型系统也在持续地演进与完善. 在传统的形式化研究中, 类型系统的可靠性往往通过证明其进展 (progress) 和保持 (preservation) 性质来确立, 即语法类型良好 (well-typed) 的程序不会陷入无法继续执行的僵局. 该方法称之为语法类型可靠性 (syntactic type soundness). 然而, 这种类型系统证明的标准方法不容易扩展到混合安全和 unsafe 代码的情况, 因为不安全的代码

不符合 Rust 基于所有权和借用检查的强大类型系统中所定义的语法良好性. 在现实世界的应用中, 许多广泛使用的 Rust 标准库 API 内部都包含 unsafe 代码来扩展表达能力, 如何确保该 API 在观察上是安全的 (observationally safe) 是一个重要问题. 观察上安全意味着即使在存在 unsafe 代码的情况下, API 对外的表现仍然是符合 Rust 安全原则, 不会对调用者造成不可预测的影响. 针对此问题, 现有研究工作提出了语义类型可靠性 (semantic type soundness) 的概念, 即证明语义类型良好的程序是内存安全和线程安全的, 可以安全执行. 本节将依次探讨这两种证明方法, 并展开相关研究工作.

3.2.1 针对纯 safe Rust 的语义研究

比较早期的形式化语义建模工作包括 Patina[23], 旨在为 Rust 早期版本 (1.0 之前) 提供内存安全证明. Patina 描述了 Rust 早期借用检查器的操作语义, 并基于该操作语义, 给出了该语言子集的可靠性证明. 然而, 早期 Rust 语言设计不够稳定, 因此 Patina 的模型中缺少很多 Rust 关键特性, 如借用检查过程中缺少生命周期推理, 因此该模型在现阶段的可参考性有限.

Payet 等人[22]参考 Featherweight Java[28]提出 Featherweight Rust 轻量演算 (lightweight calculus) FR, 描述了 Rust 复制、移动、借用和生命周期特性. FR 仅限于小的 Rust 子集, 如涉及语句表达式仅包括 let 绑定、赋值、移动、部分移动和借用, 没有考虑控制流、类型转换、函数可变性、通用函数调用语法等. 该演算采用了流敏感的类型系统来编码类型和借用检查的规则, 并证明类型系统的语法可靠性, 即类型和借用安全的程序不会陷入僵局 (get stuck), 并保持借用不变式, 确保类型和借用安全的程序不会尝试解引用悬挂指针. 文中探索了对 FR 演算的几个扩展, 如对控制流、元组、方法调用等更复杂特性的支持, 但没有证明扩展后类型系统的可靠性. 该研究基于 FR 形式化系统定义的规则实现了一个工具, 对程序进行类型检查和借用检查. 该工具解析 Rust 源代码为抽象语法树 (AST), 遍历树形结构为每个节点分配类型并确保程序中的所有类型操作都是合法的, 最后进行借用检查, 确保程序遵循 Rust 的借用规则. 实验中还对 Rust 编译器进行了模糊测试, 基于 FR 定义的规则生成符合或不符合类型和借用安全的程序, 比对编译器的判断结果, 发现了一个确认的编译器错误.

同样参考 Featherweight Java, Weiss 等人[20]提出了 Oxide, 在类型系统层面对 Rust 的借用检查规则进行了精确的描述. 相比于 Featherweight Rust, Oxide 覆盖大量 Rust 子集. 此外, 不同于 FR 使用的结构化生命周期, Oxide 提出了一种类似于原型借用检查器 Polonius 中的生命周期视角, 即, 将生命周期视为一系列称为区域 (region) 的位置集合, 这些 region 能够近似地表示引用的起源 (origin), Oxide 借用检查器利用这些关于引用的起源信息来决定在程序的特定位置创建或使用引用是否安全, 实现对引用的精确跟踪和管理. Oxide 中对每个借用表达式都自动注释了与其绑定的 region, 并定义相应的类型规则基于该注释信息进行推导, 从而实现一种基于区域的别名管理系统. 具体来说, region (表示为'a、'b 等) 可以被视为一系列借用 (loan) 的集合, 这些借用集合共同构成了对引用起源 (origins) 的静态近似. 这样, 我们可以将 region 理解为内存中不同对象的静态分组. 当我们将引用与特定的 region 关联时, 我们实际上是在指明该引用可能指向的内存对象必须来自于与该 region 相关联的 loan 集合. Oxide 语言的设计接近于 Rust 的源代码级别, 但增加了类型注释. 如图 9 的简单 Oxide 代码为例. 该例中定义了一个包含 u32 类型字段的结构体 Obj, x 是该类型的可变实例, y 是对 x 的引用, z 是重借用. 注释内容展现了类型检查过程中内部元数据的更新. 在类型检查时, 第 2 行通过特定语法 letrgn 引入了两个新的 regions $'y$ 和 $'z$, 各自关联一个空的 loan 集合. 第 4 行创建对 x 的一个唯一引用 (unique reference) y 后, 这次借用操作产生元数据 $'y \rightarrow \{^{\text{uniq}}x\}$, 它将 region $'y$ 与包含 $^{\text{uniq}}x$ 的 loan 集合相关联. 这表明, 任何带有 region $'y$ 标识的引用都必须指向 x. 其中, uniq 表示这是唯一引用 (即可变引用), 和共享引用 shrd 区分开. 第 5 行从 y 重新借用得到 z 后, 这次操作将区域 $'z$ 与包含 $^{\text{uniq}}x$ 和 $^{\text{uniq}}{*y}$ 的 loan 集合相关联. $^{\text{uniq}}{*y}$ 表示从 y 的重新借用. 这包含两个关键信息: 首先, 带有 region $'z$ 标识的引用指向 x; 其次, 带有 region $'z$ 标识的引用是通过从 y 重新借用而创建的. 后者意味着, 只要引用 z 存在, y 就应该被视为不可用, 以确保引用的合法使用. 该研究利用了基于语法的方式证明了 Oxide 的类型安全性. 此外, 该研究实现了一个从 Rust 到 Oxide 语言的编译器以及一个原型类型检查器, 实验评估了 Oxide 类型检查器结果与 rustc 借用检查器在 Rust 官方非词法生命周期测试套件上的一致性.

```
1  struct Obj(u32);
2  letrgn<'y, 'z> { // 'y -> {}, 'z -> {}
3      let mut x = Obj(5); // x : Obj
4      let y = &'y uniq x /* 'y -> { uniq_x } */;
5      let z = &'z uniq *y /* 'z -> { uniq_x , uniq_*y } */;
```

图 9 Oxide 代码例子

Lehmann 等人[21]基于“液态类型 (liquid type)”方法提出 Flux, 通过引入用于函数式验证的精化类型扩展 Rust 的类型系统. 液态类型的方法将类型构造器与简单的无量词逻辑谓词组合来表达复杂不变式, 以简化程序验证. 精化类型中可以加入额外的逻辑约束, 表达关于值的更多信息. 例如, 一个整型变量除了 int 类型, 还可能被进一步细化为 int≥0 来表示它总是非负的. 然而, 命令式语言 (imperative language) 中别名的问题使得很难追踪变量类型的改变, 因此已有的精化类型研究主要针对函数式语言 (functional language). Flux 将逻辑精化与 Rust 的所有权机制相结合, 实现命令式 (安全) Rust 的验证. 为了处理对可变引用的写操作, Flux 引入了“强引用”的概念, 允许引用的类型发生变化 (即允许类型强更新), 并跟踪确切的引用位置. Flux 基于 Stacked Borrows 提出的别名规则 (aliasing discipline) 描述其操作语义, 并证明了其类型系统的可靠性, 确保“良好借用的良类型程序 (合法程序) 的评估不会陷入僵局”. 基于 Flux 的类型系统规则构建了一个轻量级且高度自动化的验证工具, 实现为 Rust 编译器插件, 在编译器做检查后分析程序, 自动合成精化类型, 循环不变式和霍尔逻辑约束, 最终基于不动点的方式求解约束, 定位出错位置. 为展示精化类型相对于程序逻辑的优势, 实验对比了该工具和基于程序逻辑的验证器 Prusti 的验证效率, 表明在轻量且普遍存在的验证用例上, 如在使用向量的程序中检测索引溢出错误时, Flux 的液态类型将规范行数减少一半从而将验证时间减少一个数量级, 且能够自动合成循环不变式注释, 减少了人工标注的负担. 然而, Flux 通过将类型构造器与简单的无量词逻辑谓词组合来表达不变式, 相比之下, Prusti 等基于程序逻辑的方法的有更丰富的表现力, 可以验证深层功能正确性规范. 此外, Flux 同样针对的是安全代码的验证, 不能推理关于低级指针操作等问题.

3.2.2 针对包含 unsafe Rust 的语义研究

为了证明包含 unsafe Rust 的安全 Rust 标准库 API 是否提供了安全抽象, Jung 等人[13]采用了米尔纳 (Milner) 风格的语义方法描述 Rust 类型系统, 允许在使用不安全特性的情况下也可以做类型推导. 即用丰富的程序逻辑将某个类型解释为分离逻辑中的一个命题, 所有符合这个命题的值都可以有这个类型, 将类型判断 (typing judgment) 解释为这些谓词之间的逻辑蕴涵. 这种方法不同于语法上基于类型规则的类型定义. 基于这种方法证明语义类型的可靠性, 即语义类型良好的程序是内存安全和线程安全的, 所有执行的指针访问都是有效的, 并且不存在数据竞争, 可以安全执行. 语义类型良好限定为需要遵守的验证条件, 即证明只要某个库中的 unsafe 代码符合某些限制条件, 就是做了安全的封装. 该研究采用高阶并发分离框架 Iris[29,30]机械化建模了形式语义模型 RustBelt, 并用它以机器检查的方式验证了大量使用 unsafe 代码的 Rust 标准库抽象的外在安全性. 即, 证明某内部包含 unsafe 代码的库满足模型中定义的该接口的验证条件. 该研究验证了几个使用 unsafe 代码的常用 Rust 标准库, 包括: Arc、Rc、Cell、RefCell、Mutex、RwLock 等. RustBelt 语义模型中描述了 Rust 的核心子集 λRust, 包含所有权和生命周期等基本语法特性. 其中一大亮点是基于 Iris 提出了生命周期逻辑 (lifetime logic), 模仿了 Rust 借用检查机制. RustBelt 借助给出或收回生命周期令牌 (lifetime token) 的方式表示赋予或收回对某数据结构的暂时性访问权限. 在每次借用语句时该模型通过生命周期逻辑的推理规则检查借用的有效性, 即确保在使用某引用时, 它拥有表示其生命周期是活跃的 lifetime token. 此外, 在 RustBelt 中详细描述了线程安全的概念. 在 RustBelt 模型中, 所有权谓词 (ownership predicate) 表示拥有一个 T 类型的值, 共享谓词 (sharing predicate) 表示拥有一个&T 类型的值. 为了捕捉线程安全的语义, 这两个谓词都引入了一个额外的参数, 该参数与线程标识符 (thread identifier) 相关联. 通过该线程标识符对应的参数, 类型系统能够建立类型与声明所有权线程之间的依赖关系. 在 Rust 中, Send 表示类型 T 是线程安全的, Sync 表示类型&T 是线程安全的. 而上述方法提供了一种直观的方式来描述 Send 和 Sync 的特性: 在语义层面, 如果类型 T 的 ownership predicate 不依赖于线程标识符, 那么类型 T 就是 Send 的; 如果类型 T

的 sharing predicate 不依赖于线程标识符, 那么类型 T 就是 Sync 的. 这确实符合直觉, 因为如果类型 T 的值 v 的所有权与线程无关, 则在线程之间转移 v 的所有权是完全安全的. 基于这种方法, RustBelt 提供了对并发语义的支持. 然而, 该工作还存在很多局限. λRust 是 Rust 的子集, 没有包括 traits 等高级语言特性. 并且, λRust 建模的是顺序一致性内存模型, 避免处理松散内存 (relaxed memory) 所带来的复杂问题, 而实际上某些库 (如 Arc) 采用 related-memory 操作. 针对此问题, Dang 等人[31]对 RustBelt 进行了扩展, 加入了对 Rust 库广泛使用的弱内存模型的支持. 该研究针对弱内存模型下的内存回收问题, 提出了同步影子状态 (synchronized ghost state) 的结构, 并在实验中发现了 Rust 的 Arc 库中的一个未知的数据竞争安全漏洞. 其次, λRust 是一个具有简化内存模型的小型 Lambda 演算. 例如, 它有意避免了处理真实的 Rust 堆的挑战, 且操作语义与部分编译器优化不兼容. 此外, 虽然 RustBelt 具有可扩展性, 用户可以验证自己的 unsafe API, 但是该框架不能直接自动化用于验证 Rust 源程序. 用户需要手工将 Rust 源代码描述成 RustBelt 模型中的 λRust 形式的代码再使用该工作基于的 Iris 框架进行机械式证明, 学习门槛高且过程耗时又容易出错. 后文将展开介绍缓解这两个问题的 RefinedRust 工作.

Matsushita 等人[19]基于 RustBelt 提出的语义类型 (semantic typing) 方式, 加入 RustHorn 风格的一阶逻辑 (FOL) 形式的规范, 定义了 RustHornBelt 类型系统, 并提供了经过机器检查的可靠性证明. 该研究通过一种叫做参数化预言 (parametric prophecies) 的新分离逻辑机制开发了 RustHorn 风格的语义模型. 参数化预言的思路由 RustHorn 工具提出, 然而, RustHorn 的可靠性仅针对 Rust 的安全子集建立, RustHornBelt 将其扩展以支持封装 unsafe 代码的安全 API, 为其提供一阶逻辑 (FOL) 形式的规范. 此外, RustHornBelt 中不仅像 RustBelt 一样验证了类型安全, 还考虑了功能正确性.

Gäher 等人[15]扩展了 RustBelt 语义模型并加入了精化所有权类型, 定义了 RefinedRust 类型系统, 并提供了经过机器检查的可靠性证明. 该研究针对 RustBelt 在内存模型精确度和自动化验证方法方面的不足进行了改进. 一方面, 该工作扩展了 λRust 的内存模型以描述更多 Rust 特性, 以支持验证包含 unsafe 代码的真实 Rust 程序. λRust 使用简化内存模型以便于形式化验证以及专注于 Rust 所有权和借用规则的核心特性. 然而这种简化可能会忽略一些 Rust 语言的底层细节和优化. 例如, 在 λRust 中, 抽象地将所有整数看作无界整数, 整数只有一种类型, 并只占用一个内存位置 (memory cell). 而实际上 Rust 提供 12 种不同的原始机器整数类型, 这些类型在内存中占用 1–16 个字节. 此外, Rust 编译器会做一些优化, 如利基优化 (Niche optimization) 在不影响表达力的情况下改变某结构体中不同字段的布局顺序, 以减小结构体的总体大小. 而在 λRust 的内存模型中, 结构体具有固定的布局, 并且忽略了对齐, 因此结构体的字段之间不需要做填充. 针对此类问题, RefinedRust 参考 RefinedC[32]中对 C 操作语义的描述, 并真实而细粒度地建模了 RustMIR 的操作语义 Radium. Radium 中定义了数据类型在内存中的排列方式, 包括其大小和对齐要求, 并详细定义了加载 (loads) 和存储 (stores) 操作受到内存访问布局的影响, 以支持底层原始指针操作等. 如此, 相比 λRust, Radium 中定义的整数是有界的, i32 类型的整数占用 4 个字节. 然而, Radium 中也有很多 Rust 语义没有包含, 如递归类型, trait, closure 等. 另一方面, 该研究基于 RefinedRust 类型系统中定义的规则实现了一套工具链, 能够半自动化地对包含安全和 unsafe 代码的 Rust 程序进行功能正确性验证. 具体来说, 在验证过程中, 带有注释的 Rust 程序被转换成 Coq 中浅嵌入 (shadow embedding) 的 Radium 语言描述的程序, 并基于分离逻辑证明该程序符合 RefinedRust 类型系统定义的规则. 该工作是首个在有正确性保证下验证包含 safe 和 unsafe Rust 程序的工作. 实验中通过验证 Rust 标准库 Vec 实现的一个变体, 表明 RefinedRust 可以完成复杂的推理, 验证包含不安全指针操作的 Rust 程序.

3.3 基于纯操作语义的 Rust 语义研究

除了依赖于类型系统的方法, 还有一些研究专注于基于 Rust 的操作语义来构建形式规范, 这些基于操作语义的方法不涉及类型推导的规则定义, 而是将所有权和借用检查作为内存访问控制的核心部分, 以前提条件的形式嵌入在了内存访问操作语义的语义规则中. 操作语义方法的特点是直接对程序的执行行为和内存等状态的变化进行建模. 其中, 状态为程序执行期间任一时刻观察到的变量取值, 迁移关系规定如何从一个状态迁移到下一个状态, 一般定义为一组迁移规则, 每条迁移规则对应一条程序语句. 研究者通过证明模型满足一系列不变式 (invariants)

来确保其安全性和正确性, 而不是依赖类型系统的可靠性证明. 例如, 可以证明在程序的任何执行时刻, 对特定内存位置的有效可变访问不会与任何其他形式的访问冲突. 这种证明过程需要对操作语义进行深入分析, 并严格检查程序状态的变化, 以确保所有安全性条件得到满足.

Wang 等人[25]基于 K 框架 (K-framework) 提出 Rust 子集的形式化可执行操作语义. K 是一个专门用于描述编程语言语义的形式框架, 该研究在 K 中定义了第一个针对 Rust 的形式化可执行语义 KRust, 并借助 K 框架基于 KRust 定义的语义规则自动生成了 Rust 程序的解释器和验证工具. KRust 包括了 Rust 中所有的基本类型、基本运算、复合数据类型、基本的控制流、函数等语法特征. 语义规则涵盖 Rust 的 3 个重要特性, 即所有权、借用和生命周期. 实验中使用 Rust 官方编译器提供的测试集, 将该解释器执行 Rust 程序的结果和 Rust 编译器的结果进行一致性对比测试, 发现了一个 Rust 编译器中的错误. 然而, 虽然 KRust 中有所有权、借用和生命周期的基本概念, 但语义定义不够全面. 例如, 没有描述重借用的过程, 没有描述 unsafe 机制等. 此外, 由于 KRust 也是较早期的工作, 也采用传统的词法生命周期推断.

Kan 等人[26]同样基于 K 框架提出了 Rust 可执行操作语义. 相比于 KRust, RustSEM 涵盖了 Rust 主要语言特性的更大子集, 包括 unsafe 原始指针, 并发等特性, 并描述非词法生命周期. RustSEM 语义的核心是具有 OBS (所有权和借用系统) 操作语义的内存模型. 该文章强调, 基于类型系统的验证工作将 OBS 形式化为语言层面 (language-level) 的类型约束, 只能用于在编译时保守地分析程序是否符合 OBS 不变量, 可能会拒绝动态执行时实际正确的程序. 相比之下, RustSEM 在内存层面 (memory-level) 为 OBS 提供了操作语义, 能够在动态执行 Rust 程序时自动化验证程序的运行时行为. RustSEM 中定义了多个 OBS 不变式, 例如, 生命周期包含不变式即, “如果 X 是 Y 的 (可变/不可变) 引用, 则 X 的生命周期应该始终在 Y 的生命周期之内, 以避免悬挂指针”, 并在程序做内存访问操作的时候检查这些 OBS 不变式是否都被满足. RustSEM 提供动态检查器, 检测 Rust 程序的内存安全, 如所有权和借用错误, 缓冲区溢出等问题. 此外, 该研究利用 K 框架自动生成了 RustSEM 验证器, 能基于 RustSEM 的操作语义对 Rust 程序做符号执行推理, 如基于霍尔逻辑验证 unsafe 代码是否会导致未定义行为. 实验评估了 RustSEM 相对于 Rust 编译器的语义正确性, 并表明 RustSEM 能够检测出更多内存错误, 拒绝编译器允许通过的错误程序. 然而, RustSEM 的内存模型较为抽象, 没有精准描述 Rust 的内存模型. 例如, 其内存模型没有反映值的字节级表示, 并且不支持无界堆片段. 此外, RustSEM 主要检测内存相关的错误, 不能验证较复杂的功能正确性.

类型系统的价值不仅体现在它们为程序提供的安全性保障上, 更在于它们通过简化关键的程序分析过程, 助力编译器生成效率更高的代码. 在 Rust 语言中, 类型系统对指针别名实行了严格的规范, 而 Rust 编译器团队的一个核心目标就是利用这些别名信息来实施程序优化, 以重新排序内存访问操作. 然而, Rust 对 unsafe 代码的支持带来了挑战, 因为程序员可以通过 unsafe 代码绕过编译器的常规检查, 从而违背别名规则. 为了在优化需求与 unsafe 代码的使用之间找到平衡点, 语言的设计必须提供一套清晰的规则. 这些规则要能够让编写 unsafe 代码的开发者确信, 只要他们遵守规则, 即使在编译器执行各种优化的情况下, 他们的代码语义也能得到保留. Jung 等人[24]提出了 Stacked Borrows 操作语义框架, 定义了 Rust 引用类型内存访问的别名规则, 并规定违反这些规则的程序将具有未定义行为. 这样的设计意味着编译器在执行优化时, 可以忽略那些违反规则的程序. 该工作旨在利用这些类型中编码的别名信息以实现过程内优化, 即允许编译器仅凭过程内推理, 就对未知代码和函数调用周围的内存访问进行重新排序优化, 这些优化的有效性已通过 Coq 形式化验证. 该模型的创新之处在于, 它实现了 Rust 原有静态借用检查器的动态版本, 并将原有针对安全 Rust 代码段的借用检查逻辑扩展到整个语言, 即能够将分析范围扩展至 unsafe 代码, 特别是那些操作原始指针的代码. 值得一提的是, 该工作没有像 Rust 静态借用检查器中那样显式描述生命周期, 理由是编译器对生命周期的推理规则在不断演进, 从基于 AST 的结构式借用检查演变到非结构式生命周期 (NLL) 再到 Polonius 项目, 而这不该影响对程序是否是有良好行为的 (well-behaved) 判断. 该工作在不提及生命周期的情况下重新表述这一特性为, “引用的每次使用以及由其派生的所有内容都必须发生在被引用对象 (referent) 的下一次使用之前”, 并将其定义为“栈原则 (stack principle)”. 该研究将此模型实现在 Miri 解释器中, 并通过该解释器运行了 Rust 标准库的大部分测试套件, 以验证模型能够兼容并支持现实世界中广泛存在的 unsafe Rust 代码. Miri 的评估工作中 Stacked Borrows 发现的违规问题基本都是被 Rust 社区作为 bug 并后期

修复的, 表明其有效性. 该研究未来计划将 Stacked Borrows 与 RustBelt 中的 Rust 形式化类型系统相结合, 以验证所有安全的 Rust 程序是否都遵循 Stacked Borrows 规则. Stacked Borrows 模型的长期发展目标是成为 Rust 官方语义的一部分, 这需要与 Rust 社区和开发团队保持紧密合作, 根据实际需求不断调整模型. 这不仅将增强 Rust 语言的安全性和可靠性, 也将为编译器优化提供坚实的理论基础.

4 面向 Rust 程序的自动化验证工作

自动化程序验证研究的主要目标是基于程序验证相关理论 (如霍尔逻辑, 分离逻辑等), 利用自动化验证工具实现自动推理, 证明程序的部分或完全正确性, 即程序在运行时满足特定规范. 自动化验证工具能够自动构建证明和检查过程, 不会暴露底层复杂的形式逻辑, 允许用户在编程语言的抽象级别上工作. 现有的 Rust 自动化验证工具依据它们所依赖的验证技术, 分为半自动化程序验证方法、模型检测方法和定理证明方法. 每种方法有各自的优势及适用的应用场景. 总结来说, 半自动化验证器以模块化的方式高效验证程序功能准确性等复杂性质. 这些验证器利用 Rust 类型系统提供的安全保证来简化推理过程. 然而, 它们在同时支持 unsafe 代码方面面临挑战. 定理证明器采用浅嵌入技术, 在其他语言中以纯函数式代码的形式规范描述 Rust 的一个子集, 并利用该语言的后端工具链进行后续证明. 该方面避免了显式描述程序内存布局, 但目前仅限于验证 Rust 安全代码, 且在语言转换过程中缺乏正确性保证. 模型检查器通过将 Rust 程序转换为类似 C 的表示形式, 并借助已有 C 内存模型支持验证低级内存模型的 unsafe 代码和并发代码. 但这种方法缺乏灵活性, 仅限于验证基本属性, 且不能精确描述 Rust 生命周期等自身语言特性. 表 2 中更细粒度地列举了不同 Rust 自动化程序验证工具的特点. 值得注意的是, 虽然一些专注于 Rust 语义的研究也开发了相应的自动化程序分析工具, 但这些工具的主要目的并不在于自动程序验证, 本文已将这些工作列入了第 3 节, 在此不再赘述. 在介绍这些工具前, 本节先主要围绕分离逻辑介绍一些基础概念. Rust 语言独特的类型系统和所有权规则提供了类似分离逻辑的表达能力, 为程序验证提供了新的解决方案. 理解这些概念对于掌握 Rust 的自动化验证工具至关重要, 同时也为如何有效利用这些工具提供了理论基础.

表 2 Rust 自动化程序验证研究工作总结对比

验证技术	研究工作	中间语言	数据结构	验证性质	支持语言特性							
					Closure	Unsafe	并发	Trait	借用			
									(1)	(2)	(3)	(4)
半自动化	Prusti[33]	Viper	MIR	功能正确性	●	○	○	●	●	●	—	○
	Verus[34]	无	HIR	功能正确性	●	◐	◐	●	●	○	—	—
	Creusot[35]	WhyML	MIR	功能正确性	●	○	○	●	●	●	●	—
	RustHorn[36]	CHC	MIR	功能正确性	○	○	○	○	●	●	—	—
	VeriFast[37]	C AST	MIR	类型安全	○	●	○	○	●	—	—	—
	Gillian-Rust[16]	GIL	—	功能正确性, 类型安全	○	●	○	●	◐	◐	—	—
模型检测	Crust[38]	C-like	AST	内存安全	○	●	○	○	●	—	—	—
	SMACK[39]	Boogie	LLVM IR	内存安全, 功能正确性	●	●	○	●	●	—	—	—
	Kani[40]	C-like	MIR	内存安全, 功能正确性	●	●	○	●	●	●	—	—
	Loom[41]	C	AST	并发安全	—	○	●	○	●	—	—	—
	TrustPN[42]	Petri-Net	MIR	并发安全	—	○	●	○	●	—	—	—
定理证明	Electrolysis[43]	Lean	MIR	功能正确性	●	○	○	●	◐	○	○	○
	Aeneas[44]	LLBC	MIR	功能正确性	○	○	○	○	●	●	◐	●

注: ●: 支持; ○: 不支持; ◐: 部分支持, 例如Gillian-Rust只支持可变引用而不支持不可变引用. —: 没有显式说明. 各类借用形式: (1) 常见的借用, (2) 返回借用(return borrows), (3) 嵌套借用 (nested borrows), (4) 两阶段借用 (two-phase borrows)

4.1 分离逻辑与 Rust 所有权系统

分离逻辑 (separation logic)[45]是对霍尔逻辑 (Hoare logic) 的一种扩展. Hoare 逻辑[46]是一套基于公理语义的程

序正确性推理系统, 自提出以来已成为所有程序验证系统的核心基础. 它基于霍尔三元组{P}C{Q}刻画程序行为, C 是所验证的程序, P 和 Q 通常都是一阶逻辑里的公式, 用于描述程序 C 里的程序变量需满足的约束条件, 其中 P 称为前置条件 (precondition), Q 称为后置条件 (postcondition). 分离逻辑 (separation logic) 的目标是在霍尔逻辑的基础上支持验证访问堆内存的程序. 它引入指向运算符 $\mapsto$, $x \mapsto y$ 表示 x 的值对应的地址, 在堆上对应的值等于 y 的值. 而其难点在于同一个内存地址可以被多个指针所指向, 当一个指针指向的数据被修改时, 有关其他指向该地址的指针的程序断言可能不再成立, 从而使证明难以进行. 分离逻辑的解决方法是在程序断言中引入分离合取 (separation conjunction), P1*P2 来表示程序断言 P1 和 P2 各自描述了一块符合对应断言的内存, 且各自的内存地址是没有重叠的. 如 $x \mapsto 1 * y \mapsto 2$ 可以理解成两个指针 x 和 y 和各自指向的值为 1 和 2, 且 x 和 y 指向不同的地址. 这样就避免了不同的断言之间有共享的地址, 在使用一部分断言和其对应的内存进行证明的时候, 程序对内存的影响就不会改变其他断言的成立性. 基于此, 分离逻辑最大的优点之一就是可以很有效地支持局部推理, 得益于该逻辑特有的框 (Frame) 规则.

$$\frac{\Gamma \vdash \{P\}c\{Q\}}{\begin{array}{c}c\text{不使用}R\text{中的变量和内存}\\ \Gamma \vdash \{P*R\}c\{Q*R\}\end{array}}.$$

Frame 规则结论中的前后条件分别都分离合取了同一个断言 R, 如果程序 C 不会对 R 所描述的那部分程序状态与内存进行修改, 那么证明 C 的过程中可以不使用断言 R, 因而其前提是一个前后条件为 P 和 Q 的霍尔三元组. 这样一种将不需要用到的内存的断言从霍尔三元组中暂时移除的方式, 既使得模块化的证明更为简洁, 又保证了可以证明一个程序不会访问禁止其访问的内存. 如此, 在分析验证局部程序时只需考虑该局部程序所访问的内存区域, 而不需要涉及全局状态.

然而, 分离逻辑对用户和工具都有较高的要求, 使得其在实际应用中的普及受到限制. 用户需要自定义丰富的断言来描述程序的行为, 包括指向性谓词、分离合取等, 这要求用户有深入的理解才能编写出准确的规范, 导致这些逻辑的应用主要局限于专家研究者的小圈子内. 此外, 在基于分离逻辑的证明过程中, 为了确保内存安全需要详细追踪资源的使用情况和内存的状态, 这不仅增加了证明的复杂性, 也对自动化工具提出了更高的要求. 现代形式化验证工具, 如 SMT (satisfiability modulo theories) 求解器[47], 已经在处理命题和一阶逻辑问题上取得了巨大进步. 相比之下分离逻辑的自动化验证仍处于起步阶段, 且面临更大的技术挑战, 因为分离逻辑通常涉及比普通逻辑更复杂的计算复杂性类别. 在这一背景下, Rust 语言以其独特的类型系统和所有权规则, 为形式化验证提供了新的视角. Rust 的所有权模型在语言层面上内置了分离逻辑的思想, 对指针别名实行了严格的规范. 例如, 在 Rust 中声明两个类型相同的变量 x 和 y 时, Rust 所有权模型保证它们天然就是分离的. 此外, 指向堆分配内存的智能指针 Box<T> 提供了表达“指向”关系的谓词, 这在分离逻辑中是至关重要的. 从某种意义上说, Rust 的类型检查器在每次编译时都执行一种分离逻辑证明. 因此, Rust 允许在验证时依赖 Rust 的类型系统来实现类似分离逻辑的精确推理, 捕获关于内存位置的关键信息, 包括别名、副作用、Frame 规则和是否存在数据竞争等, 同时避免了繁琐和困难的资源追踪过程. 如此, 设计和使用自动化工具时就可以将注意力集中在要证明的核心功能属性上, 例如功能正确性. 这对于提高验证效率和准确性具有重要意义, 也使得用户体验更友好. 当然, 为了分析 Rust 程序, 还需要做额外的扩展. 例如, 借用机制的引入尽管为 Rust 带来了灵活性, 但也为分离逻辑的 Frame 带来了挑战. 当创建一个可变引用 x=&mut y 时, y 的值似乎可以通过两种方式访问, 这在分离逻辑中需要特别处理, 以避免别名问题对证明过程的影响.

例如, 图 10 中包含可变借用的例子就展示了 Rust 类型系统所提供的安全保证简化了程序验证问题. 假设 p0 和 p1 是两个 Point 实例, 传入 align 函数中分别表示为 segm.0 和 segm.1. 为了证明 align 函数调用中的断言, 必须确保几个关键属性得到满足: (1) 调用 shift_x 会将 p1 的 x 字段的值增加 s 值, 这是实现功能性正确性所需的后置条件. (2) 这个调用不会影响 p0 的 x 字段, 因此调用结束后, p0.x 将等于 p1.x. (3) shift_x 的调用不会更改传入的 segm 元组, 这意味着调用后 p0 和 p1 仍然分别等于 segm.0 和 segm.1, 从而保证了 (*segm.0).x 与 (*segm.1).x 的值相等. (4) 代码是无数据竞争的, 这确保了在整个执行过程中所有内存位置的值保持稳定. 而这些属性中的 (2)–(4)

都得益于 Rust 的类型系统提供的内在保证. Box 类型确保了其值具有唯一的所有权, 这意味着 segm.0 和 segm.1 不可能是指向同一内存位置的别名, 这验证了属性 (2) 和 (3). Rust 的类型系统还要求, 要修改一个内存位置必须拥有对该位置的独占访问权, 这消除了对线程交互执行的担忧, 也免去了显式证明无数据竞争的需要, 这验证了属性 (4). 因此, 程序员在验证时只需要关注属性 (1) 的功能正确性证明即可.

```
1  fn shift_x(p: &mut Point, s: i32) {p.x = p.x + s}
2  fn align (mut segm: (Box<Point>, Box<Point>)) -> (Box<Point>, Box<Point>) {
3    let end_x = (*segm.1).x;
4    shift_x(&mut segm.1, (*segm.0).x - end_x);
5    assert!((*segm.0).x == (*segm.1).x);
6    segm
7    }
```

图 10 Rust 简化程序验证的例子

4.2 半自动化程序验证方法

半自动化 (auto-active) 程序验证的方法要求开发人员在实现代码上编写证明注释 (annotation), 例如前置条件、后置条件和循环不变式. 验证器将注释代码转换为验证条件, 并调用约束求解器检查其有效性. Rust 的半自动验证工具利用 Rust 语言类型系统提供的保证大大简化编写 Rust 验证工具所需要的规范和验证过程, 能够支持验证超出内存安全的正确性属性, 如功能正确性. 这些工具多实现为了 rustc 编译的一个插件, 在编译器对 Rust 程序做完类型检查和借用检查后, 对程序进行额外性质, 如功能正确性的验证. 近年来 Rust 半自动程序验证器的开发取得了显著进展. 早期的 Rust 半自动验证器多是基于已有工具扩展了前端, 要将 Rust 转到该工具所支持的中间语言继而复用后面的工具链进行验证, 这种方法在表达能力上存在一定的局限性, 后期开发者们设计了专门针对 Rust 语言特性的验证器, 不需要适配到中间语言. 得益于灵活丰富的表达能力和易用性, 这些新兴工具在 Rust 程序验证实践中得到了广泛的使用. 图 11 概述了利用半自动化程序验证方法进行 Rust 代码验证的典型步骤. 然而, 实际执行过程中可能会根据具体情况有所差异. 例如, Verus 验证器[34]没有转换中间表示语言这一步骤.

图 11 半自动化程序验证方法流程

早期的半自动化验证工具包括 Astrauskas 等人[33]基于 Viper 分离逻辑引擎[48]提出的 Prusti. Prusti 将注释的 Rust 程序自动翻译成 Viper 中间验证语言并以模块化的方式自动验证 Rust 程序的功能正确性. 验证结果会从 Viper 翻译回 Rust, 并利用 Rust 编译器自身的错误报告机制向用户报告. Prusti 利用 Rust 编译器的类型检查和借用检查信息, 在 Viper 框架中自动合成 Rust 程序内存安全的核心证明. 该证明是为自动化量身定制的分离逻辑的一个变体. Prusti 基于该核心证明重新验证了 Rust 的借用检查器强制执行的所有权属性. 用户可以通过在 Rust 语言的抽象层次上提供规范以扩展核心证明, 从而验证丰富的功能属性. Prusti 遵循传统半自动化验证器模块式的验证方式, 即关注每个函数调用前后的旧值和当前值, 分别约束函数执行前/后需要满足的条件. 此外, Rust 类型系统的一个复杂之处在于可变借用的概念, 这为表达函数的后置条件带来了挑战. 如图 12 所示, 针对包含可变借用的程序, Prusti 提出了一套保证 (pledges) 规范构造, 在处理返回可变引用的函数时, 还考虑引用生命周期结束时该引用的值并定义该程序点需要满足的条件. Prusti 在几个广泛使用的 Rust 库中数千个函数上评估了其有效性.

RustHorn[36]通过利用 Rust 类型系统提供的所有权保证, 用一阶逻辑 (FOL) 公式表达有状态的 Rust 代码的行为, 不需要显示表示指针和内存状态, 使得这些公式的验证可以利用现成的自动化技术. RustHorn 采用预言变量 (prophecy variables) 巧妙地解决了借用状态的追踪问题, 即将可变引用表示为当前目标对象值和借用结束时的一

对值. 在创建一个新的可变借用&mut a 时, 我们对其最终状态 a_o 进行了预言, 这就像是对程序未来状态的一次预见. 在创建借用时, 我们并不立即确定 a_o 的值. 我们用当前状态 a 和预言的借用结束后值 a_o 来表示可变引用 ma, 形成一对值⟨a, a_o⟩. 当这个可变引用⟨a, a_o⟩ 的生命周期结束, 即所有权被释放时, 我们确立了 a_o=a 的关系, 意味着最终状态 a_o 被设定为那个特定时刻的当前状态 a——这一过程称为预言决议 (prophecy resolution). 随后, 这一决议可以被有效地“传递”回原始的所有者, 从而明确告知借用结束后对象的确切状态. RustHorn 的这一核心概念, 为 Rust 程序提供了一种用一阶逻辑形式来描述的方法, 使得程序验证更加高效和直观. 例如, 图 13 的左边 Rust 代码 (和图 12 同样的函数) 会被 RustHorn 转成右边的一阶逻辑公式表示, 并验证程序满足第 6 行断言. 在 max_mut 函数中, 第 2 行被 drop 的引用 (如第 1 个分支的 mb) 其值就被决定了 (如相应的 mb.b_o=mb.b). 而另一个被返回的引用, 会在第 5 行被 mc 继续修改.

```
1   #[ensures(*result >= old(*ma) && *result >= old(*mb))]
2   #[ensures(*result == old(*ma) || *result == old(*mb))]
3   #[after_expiry(if old(*ma >= *mb) { (*ma == before_expiry(*result) && *mb == old(*mb))}
4     else {(*ma == old(*ma) && *mb == before_expiry(*result))})]
5   fn take_max<'a>(ma: &'a mut i32, mb: &'a mut i32) -> &'a mut i32 {
6     if *ma >= *mb { ma } else { mb }
7   }
8   #[requires(a < i32::MAX && b < i32::MAX)]
9   #[ensures(result == true)]
10  fn inc_max(mut a: i32, mut b: i32) -> bool {
11    { let mc = take_max(&mut a, &mut b); // borrow a and b
12    *mc += 1; // end of borrow}
13    a != b
14  }
```

图 12　Prusti 程序验证例子

```
1  fn take_max<'a>(ma: &'a mut i32, mb: &'a mut i32) ->
2  &'a mut i32 {if *ma >= *mb { ma } else { mb } }
3  fn inc_max(mut a: Box<i32>, mut b: Box<i32>) {
4      {let mc = take_max(&mut a, &mut b);
5    *mc += 1;
6      assert!( *a != *b); }
7  }
```

$$\text{take\_max}\ (\langle a, a_o\rangle, \langle b, b_o\rangle, r) \Longleftarrow a \geq b \wedge b_o = b \wedge r = \langle a, a_o\rangle$$

$$\text{take\_max}\ (\langle a, a_o\rangle, \langle b, b_o\rangle, r) \Longleftarrow a < b \wedge a_o = a \wedge r = \langle b, b_o\rangle$$

$$\text{inc\_max}\ (a, b, r) \Longleftarrow \text{take\_max}\ (\langle a, a_o\rangle, \langle b, b_o\rangle, \langle c, c_o\rangle)$$

$$\wedge\, c' = c + 1 \wedge c_o = c' \wedge r = (a_o ! = b_o)$$

$$r = \text{true} \Longleftarrow \text{inc\_max}\ (a, b, r)$$

图 13　RustHorn 程序验证例子

RustHorn 实现了一个原型验证器, 针对 Rust 子集进行了实现, 并通过初步实验确认了该方法的有效性, 但该工具在表达能力和性能上具有较大的局限性. RustHorn 作者通过非机械化的证明, 证明了其逻辑编码对于原始 Rust 程序内存的语义的可靠性. 随后, RustHornBelt 利用 RustBelt 框架, 机械化地证明了 RustHorn 风格逻辑编码的可靠性.

后续, Denis 等人[35]基于 RustHornBelt 提出的理论实现了 Creusot 工具. 类似于 RustHorn, Creusot 将 Rust 程序转化为一阶逻辑 (FOL) 公式, 使用 RustHorn 风格的预言 (prophecy-based) 处理可变引用. 此外, Creusot 引入了一个用于描述功能正确性等规范的注释语言 Pearlite, 并通过良好的规范设计提供了对 trait 验证的支持. Creusot 以带有规范注释的 Rust 程序作为输入, 转化为后端引擎 Why3 的中间表示并生成验证条件, 使用 SMT 求解器进行验证.

Verus[34]是首个利用 Rust 语言本身作为规范和证明语言的 Rust 验证器. 该方法使得 Verus 具有较高的验证效率和更好的用户体验. 一方面, Verus 将带有注释的 Rust 程序直接自动转化为 SMT 求解器的输入来检查证明的正确性, 而不需要先转化成某中间语言. 另一方面, 用户用 Rust 写规范 (spec) 和必要的辅助证明 (proof), 如循环不变式, Verus 能够巧妙地利用 Rust 的线性类型和所有权检查来辅助 proof 的构建. Verus 引入 mode system 管理 Rust 写的 spec、proof 以及可执行代码, 确保对 spec mode 的规约程序做类型检查和终止性检查, 对 proof mode 的证明程序做类型检查, 借用检查和终止性检查, 并只编译 Rust 可执行代码. 图 14 的例子中体现了使用 Verus 验证斐波纳契数列函数功能正确性过程中的 3 种 mode. 与 Prusti 相比, Verus 信任 Rust 的类型检查器和借用检查器提供的信息, 因此不需要使用分离逻辑.

```
spec fn fib_spec(input: nat) -> nat {
// We need to ensure something like `fixpoint`
functions terminates.
    decreases(input);
    match input {
        0 | 1 => input,
        _ => fib_spec(input − 1) + fib_spec(input −
2),
    }
}

proof fn lemma_fibo_is_monotonic(i: nat, j: nat) {
    requires(i <= j);
    ensures(fib_spec(i) <= fib_spec(j));
    decreases(j - i);
    // specific proof goes here.
}

fn fib(input: u64) -> u64{
    ensures(|result: u64| result == fib_spec(n));
/// `fib`: implementation code goes here.
}
```

图 14 Verus 程序验证例子

除了支持更多的安全 Rust 特性外, Verus 的一大亮点是它支持一些传统上需要 unsafe 代码的验证, 为 Rust 程序的验证提供了一种新的视角和方法. 在 Rust 中, 将某个特性标记为“unsafe”实际上意味着开发者需要承担起安全使用该特性的责任, 这是 Rust 编译器所无法强制检查的. 而不安全的代码实际上可以被视为“条件性安全”, 即只要满足一定的条件就是安全的. 基于这个思路, 该研究的理念是通过 Verus 验证 Rust 编译器无法检查的条件, 从而提供这样的安全保证. 为此, Verus 引入新的、遵循 Rust 唯一所有权原则的线性对象 (linear object). 为了避免增加额外的数据负担影响程序的性能, 这些新对象还被设计为幽灵对象 (ghost objects), 即这些对象仅在验证过程中可见, 帮助构建验证条件, 但对最终编译的程序没有直接影响. 以指针操作为例, Verus 允许定义一个幽灵对象, 代表“在给定位置读写内存的权利”. Rust 的类型系统保证了这个对象的唯一所有权, 而 Verus 则确保在程序访问指针指向的数据时, 这一权利是被妥善管理的. 这种双重保障机制可以保证程序的内存访问是安全的, 没有数据竞争的风险. 在验证双向链表这类复杂的数据结构时, 不仅按照常规方式组织节点和指针, 还会引入一组额外的幽灵对象代表了访问节点的权利. 通过添加 Verus 的注释, 用户可以精确地描述这些幽灵对象与链表结构之间的关系. 通过此方式能够利用幽灵对象来安全地遍历链表, 而且通过数学证明来确保这一过程的正确性. 换句话说, 实际上 Verus 不直接验证原始指针, 而是提供包含线性幽灵权限变量的抽象原始指针 API, 并提供额外的检查保证调用该 API 代码的正确性实现. 除了针对非安全原始指针, Verus 还把这一理念用到了对内部可变性代码和并发代码的验证上. 通过这种方式, Verus 扩展了 Rust 的能力, 允许开发者在编写高效代码的同时, 享受到严格的安全保证, 从而在安全与性能之间找到了一个平衡点. 实验中在一系列示例上展示 Verus 如何通过幽灵对象和注释来验证包括操作指针的代码 (一个基于异或的双向链表)、具有内部可变性的代码和并发代码. Verus 也存在局限性, 例如, 由于 Verus 不支持函数返回可变引用, 因此不能验证如图 12 的例子.

Foroushaani 等人[49]扩展了 VeriFast[37]模块化符号执行 (MSE) 算法, 提供了 Verifast 工具对 Rust 前端的支持, 是首个支持验证 unsafe Rust 代码片段语义类型安全的工作. 该工作将 RustBelt 对 Rust 类型的语义编码到 VeriFast 中, 验证输入程序即使有 unsafe 代码也不会导致未定义行为, 即程序不会访问未分配的内存或包含数据竞争. 验

证 Rust 程序时, 用户在 Rust 程序中注释 VeriFast 支持的分离逻辑公式信息以描述前置条件和后置条件, 提供引理或幽灵指令 (ghost command) 辅助验证. 目前的 VeriFastRust 前端有一些限制, 除了自动化程度有限需要用户辅助证明过程外, 它不支持多态等特性, 并且只专注于验证语义类型安全性而非功能正确性.

同样针对 unsafe 代码, Ayoun 等人[16]提出了一种混合方法, 将 Rust 程序的安全和不安全部分交由两个不同的工具进行类型安全性和功能正确性验证. 其中, 安全部分由 Creusot 处理, 不安全部分由该研究提出的 Gillian-Rust 处理. Gillian-Rust 是基于 Gillian[50]参数化组合 (parametric compositional) 的符号执行平台, 嵌入了 RustBelt 的生命周期逻辑和 RustHornBelt 的参数预言, 并能够自动推导分离逻辑断言, 是首个验证 unsafe Rust 代码功能正确性的工作. 此外, 相比 VeriFast, Gillian-Rust 提供用户友好的 API, 自动化程度更高. 为了描述 unsafe 代码的可能行为, Gillian-Rust 扩展了 Gillian 的符号内存模型为布局无关的 Rust 内存模型, 并支持做指针运算和位级 (bit-level) 操作. 在验证程序时, 用户注释 Creusot 规约语言 Pearlite 描述的规约, Gillian-Rust 自动将 unsafe 部分的规约编码为 Gillian-Rust 支持的规约语言 Gilsonite 并进行自动推理. 该研究使用 Gillian-Rust 自动验证了 Rust 标准库 LinkedList 模块中一些双向指针列表的函数操作, 表明了该混合方法的有效性. 然而, Gillian-Rust 目前只是初步实现的原型工具, 还缺少对很多 Rust 核心特性如共享引用的支持, 虽然文中表明此类扩展具有明确可行性, 只是工程性的工作.

4.3 定理证明方法

基于定理证明的形式验证将“系统满足其规约”这一论断作为逻辑命题, 通过一组推理规则对该命题开展证明. 该方法高度抽象, 具有强大的逻辑表达能力. 这里我们主要探讨交互式定理证明方式, 即在交互式定理证明器/证明助手 (proof assistant) 中完成机械证明, 典型的工具包括 Coq[18], Isabelle/HOL[51]以及 Lean[52]等. 基于定理证明的方法验证 Rust 程序时, 利用 safe Rust 中对可变别名的限定, 自动转化其为某定理证明器语言描述的纯函数程序 (pure functional language), 进而利用现有的底层证明助手后端完成程序的进一步验证. 相比于上文提到的半自动程序验证的方法, 该方法使用外在证明 (extrinsic proofs), 即所有的规范和证明工作都在证明助手中完成, 在 Rust 程序中不需要添加注释. 而使用底层证明助手时可以以人工参与引导证明过程, 可以验证包括功能正确性在内的一些复杂的性质. 其中, 将 Rust 语言嵌入到证明助手中包括浅嵌入 (shallow embedding) 和深嵌入 (deep embedding) 两种方式. 前者的实现相对简单, 直接将程序转成证明器自己的语言的项 (terms) 而不需要显式定义该语言的内部语义, 但是表达能力相对有限, 也不能验证语义正确性相关的性质. 后者显式建模该语言的精准语义, 并将程序描述成这一模型所表达的形式, 这种方式更灵活, 能适应复杂的验证需求, 但具有一定难度. 早期 Rust 验证工作采用浅嵌入的方式, 后面逐渐定义出 Rust 的纯函数式语义. 然而由于定理证明的方法目前实现的函数式转换依赖于 Rust 类型系统提供的所有权保证, 因此只能处理不包含 unsafe 区块的程序. 图 15 描述了利用定理证明方法进行 Rust 代码验证的典型步骤.

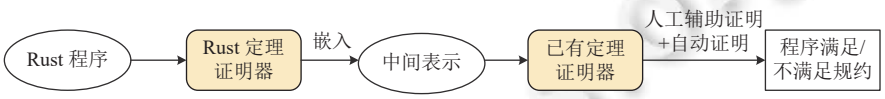

图 15 定理证明方法验证流程

早期的验证工具包括 Ullrich 等人[43]提出的 Electrolysis, 它将经过了编译器检查的 MIR 形式的 safe 程序翻译为纯 lambda 演算, 以浅嵌入的形式描述成 Lean 语言的纯函数 (pure function), 并在后续基于 Lean 证明助手[52]验证功能正确性, 为 Rust 程序验证提供了新的视角. 该方法信任借用检查器提供的内存安全保证, 因此主要检查程序功能正确性. Electrolysis 通过透镜 (lenses) 模拟可变借用, 但因此带来了一些严格的限制, 例如, 函数只能返回对它们第一个参数的借用. 此外, 由于 Electrolysis 没有建立 Rust 的形式模型, 因此没有证明其语义正确性. 它更像是一个实用的“转译器”而不是编译器; 例如, Electrolysis 将 traits 映射到类型类 (type classes), 因为在高层次上它们的工作方式相似. 实验中验证了二叉树搜索程序的功能正确性, 表明该方法的有效性.

Ho 等人[44]实现了针对 Rust 的新型定理证明器 Aeneas. 类似于 Electrolysis, Aeneas 将 safe Rust 程序转换成某函数式语言描述的状态-错误单子 (state-error monad) 形式的函数程序, 并通过现有的底层证明助手后端进行后续

的程序验证. 不同的是, Aeneas 设计了 Rust 以所有权为中心的函数式操作语义, 并使用基于借用 (loan) 的方式对程序进行了借用检查, 而没有相信 Rust 编译器的检查结果. 具体来说, Aeneas 提出了语法类似 MIR 的低级借用演算 (low-level borrow calculus, LLBC), 并赋予其纯粹的函数式的操作语义. 该语义是基于值的, 因此没有内存、地址或指针算术的概念, 减轻了证明过程中复杂的内存的推理. 进一步地, Aeneas 将 LLBC 转为纯 lambda 演算, 允许用户通过指定的定理证明器验证程序. Aeneas 目前提供了对 F*和 Coq 的支持, 后续计划扩展到 Lean, HOL4 等. 为处理函数返回借用这一复杂情况, Aeneas 使用类似于 Creusot 中预言的方法, 通过反向函数 (backward function) 来重构借用结束后返回给原始对象的值. 如, 图 12 的例子会被 Aeneas 自动转为图 16 所示 F*函数式表示. 文章通过实验结果表明, Aeneas 可以验证一个低级的、可调整大小的哈希表的功能性正确性.

```
1   let take_max_fwd (t : Type) (x : t) (y : t) : result t =
2       if x>=y then Return x else Return y
3
4   let take_max_back (t : Type) (b : bool) (x : t) (y : t) (ret : t) :
5   result (t & t) =
6       if b then Return (ret, y) else Return (x, ret)
7   }
8
9
```

```
let inc_max_fwd : result unit =
    val <-- if x>=y then true else false;
    i <-- take_max_fwd i32 1 2;
    z <-- i32_add i 1;
    massert (z = 3); (* monadic assert *)
    (x0, y0) <-- take_max_back i32 val 1 2 z;
    massert (x0 = 1);
    massert (y0 = 3);
    Return ()
```

图 16 Aeneas 程序验证例子

4.4 模型检测方法

模型检测方法将验证对象抽象成一个形式化的状态机模型, 并使用状态空间搜索的方法遍历该模型, 以此检查模型是否满足特定规约. 模型检测方法容易实现验证过程的机器完全自动化, 且当系统性质未被满足时可给出反例轨迹, 帮助用户定位错误位置. 但该技术只能对有限状态进行验证, 且存在状态空间爆炸问题. 因此, 该方法对于复杂性较高、规模较大程序的验证存在局限性. 为了缓解该问题, 有界模型检测 (bounded model checking) 通过限制程序执行情况的搜索空间, 如限制循环或递归的深度, 在牺牲一定准确性的情况下提升模型检测的效率. 此外, 由于并发程序交替执行时, 每一步都可以由两个程序中的任意一个进行, 使得其可能的执行顺序远不止一种, 导致模型检测方法在处理并发问题时效率低下. 为了缓解该问题, 无状态模型检测 (stateless model checking) 不直接追踪程序状态的变化, 而是通过特殊的调度程序遍历所有并发执行调度情况, 从而对并发程序所有可能的执行进行搜索. 图 17 描述了利用模型检测方法进行 Rust 代码验证的典型步骤.

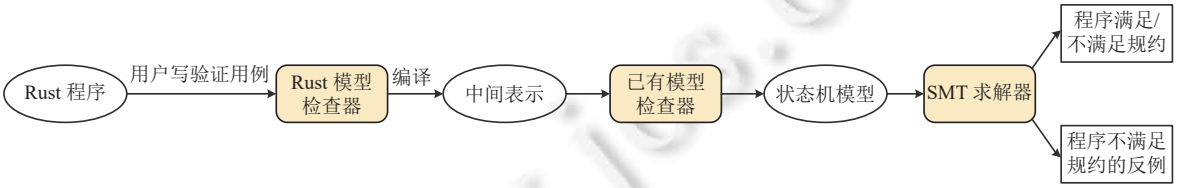

图 17 模型检测方法验证流程

Toman 等人结合穷举测试用例生成和有界模型检测技术提出 Crust[38], 验证 unsafe Rust 代码的内存安全, 并判断 Rust 指针别名的使用是否违反不变量. Crust 以某库模块为输入, 使用基于测试序列生成的方式自动构造调用了目标模块中关键函数的一系列驱动函数. Crust 将这些驱动函数和 Rust 库代码共同编译为 C 中间表示, 使用针对 C 语言的有界模型检测器 CBMC[53]来验证每个被检测函数的内存安全属性. 为了测试 Crust 的有效性, 该研究选择了向量 (vector), 切片 (slice) 和环形缓冲区 (ring buffer) 这 3 个内部调用了 unsafe 代码块的 Rust 标准库模块, 在其中手动引入两个内存错误, 实验结果表明 Crust 能够发现引入的两个内存错误.

同样基于 CBMC, AWS 团队[40]提出 Rust 语言的有界模型检查器 Kani. 相比于 Crust, 除了验证包含 unsafe 代码程序的内存安全性外, Kani 允许验证用户在 Rust 程序中自定义的验证条件, 并支持检测运行时 panic, 算数溢出等未定义行为. 后续 Byrnes 等人还扩展了 Kani 对 trait 的支持[54]. 用户使用 Kani 手动编写类似单元测试的验证用例, Kani 求解器会遍历状态空间树以检查验证用例是否正确, 如果不正确将报告反例分支. 例如, 下面的代码示例展示了 Kani 对数组越界访问的检查. 图 18 所示 Kani 验证用例示例中, #[kani::proof] 属性用于标记一个函数作为证明框架 (proof harness); kani::any 的方式符号化地表示变量为所有可能的输入值, 并通过 assume 设置前置条件约束. 该证明框架类似于测试框架, 特别是基于属性的 (property-based) 测试框架, 使得验证过程对开发者来说更加自然熟悉. 由图 18 可见, 针对上文的例子, 用户不需要在 take_max 函数手动插桩对返回可变引用的限制条件, Kani 就可以验证通过 inc_max 函数中的 assert 条件. 此外, Kani 还可以测试 unsafe 代码, 对于图 6 例子, Kani 会发现指针访问越界的解引用错误. 而相比之下, unsafe 代码可以绕过 Rust 基于属性的动态测试. 相比于早期的其他模型检查器, Kani 覆盖更广泛的 Rust 特性, 是较为成熟的工业级工具, 已经广泛用于 Rust 标准库, Tokio, Firecracker 虚拟机监视器等多种真实程序的验证. 然而, Kani 也有很多局限. 首先, 它提供较为通用和简单的断言机制, 而其他如 Prust 这类的工具则允许更精细的描述和验证特定的语言特性, 例如所有权和借用检查. 如图 19 的例子, 当所有权从原对象 p 转移到 pp 时, 右边 Kani 在第 6 行 assert 声明的断言中不能再使用原对象 p, 因此会报错. 为了解决这个问题, 开发者可以在调用 shift 的函数中引入中间变量暂存 p 的值, 再在调用 shift 后和返回的 pp 值作比较, 但该方法不够直观且增加了代码的复杂性. 相比之下, Prusti 后置条件中基于 old 关键字提供了一种更为方便的方式来描述和验证所有权转移前后对象的状态. old(shift) 表示了 shift 函数执行前 (即所有权移动前) 的值, 使得可以直观且精确地表达对象 p 和 pp 之间的关系, 实现函数功能正确性验证. 其次, 其执行的是有界模型检查而不是无界验证, 由于循环展开次数有限, 得到的结果没有可靠性保证. 此外, Kani 将 Rust 程序编码成类似 C 的表示形式, 为每种结构实例化一个特定的布局并在该上下文中执行模型检查, 使得其不能描述布局敏感的 Rust 代码. 同时, Kani 只能验证顺序执行的程序, 不能验证数据竞争等性质.

```
1  fn inc_max(mut a: i32, mut b: i32) -> bool {
2      {
3          let mc = take_max(&mut a, &mut b);
4          *mc += 1;
5      }
6      let result =   a!=b;
7      assert!(a!=b);
8      result}
```

```
#[kani::proof]
fn main() {
    let mut a = kani::any();
    let mut b = kani::any();
    kani::assume (a < i32::MAX);
    kani::assume (b < i32::MAX);
    let c = inc_max(a, b);
}
```

图 18　Kani 验证例子 1

```
// Prusti code
1  #[requires(u32::MAX - *p.x >= s)]
2  #[ensures(*result.x == old(*p.x) + old(s))]
3  #[ensures(*result.y == old(*p.y))]
4  fn shift(p: Point, s: u32) -> Point {
5      let mut pp = p;
6      let x = pp.x;       // p.x is moved out.
7      pp.x = add(x, s); // x is moved into add,
8      pp
9  }
```

```
//Kani code
fn shift (p: Point, s: u32) -> Point {
    let mut pp = p;
    let x = pp.x;       // p.x is moved out.
    pp.x = add(x, s); // x is moved into add,
    assert!(*pp.y==*p.y+s); // error: value used here after move
    pp}
}
```

图 19　Kani 验证例子 2

Baranowski 等人[39]扩展了 SMACK 工具对 Rust 的支持. SMACK 以 LLVM IR 程序为输入, 并将该程序依次转为 Boogie 代码、SMT 输入的验证条件, 最终调用 SMT 求解器进行求解, 具有较好的通用性. 该工作扩展了 SMACK 对 rustc 生成的一些 Rust 特有 LLVM IR 代码的支持, 并借助已有工具链实现 Rust 程序的验证. 由于 SMACK 可以复用其 C 模型, 因此支持 Rust 对 C 的跨语言调用. 实验表明 SMACK 能够处理一些 Rust 关键语言特性, 如 trait, closure, 并支持内存安全验证, 例如 unsafe Rust 中 C 分配的数组不会导致缓冲区溢出 (buffer overflow) 或内存泄漏 (memory leak).

Loom[41]是常用的并发验证器. Loom 会根据 C11 内存模型规定的有效的执行方式, 对输入程序可能的并发执行进行排列组合, 探索各种可能执行排列的方法并检查是否出现并发错误. Loom 使用状态简化 (state-reduction) 技术来避免组合爆炸的情况. 然而, Loom 这样可靠的状态无关模型检查工具 (sound stateless model checker) 在可扩展性方面还不够, 即使是相对较小的测试也涉及数万个原子步骤的交织执行 (interleaved execution)[14].

同样针对并发问题, Zhang 等人[42]提出了并行 Rust 程序检测工具 TRustPN. 它通过一系列转换规则自动将 MIR 形式的程序转换为 Petri 网模型, 然后分析由双重锁定 (double lock) 和读写冲突 (read-write conflict) 等引起的死锁问题. TRustPN 在转换输入程序时主要提取 MIR 中与死锁问题相关的锁定 (lock)、解锁 (unlock)、生成 (spawn) 和加入 (join) 等语句, 可以省略无关的 Rust 代码, 从而扩大处理代码的规模.

5 基于自动化工具的 Rust 程序验证成果及使用建议

Rust 语言以其在系统级安全编程领域的卓越性能和可靠性, 正逐渐成为重写和开发软件系统的首选[55–57]. 谷歌的报告指出, 自从 Android 系统引入 Rust 进行关键部分的重写, 内存安全问题得到了极大的缓解, 其比例由 76% 大幅下降至 25%[58], 表明 Rust 语言的内存安全特性为软件系统提供了一种内在的安全保障. 本文进一步提出, 通过结合 Rust 编程和形式化验证的方法, 能够为软件系统的安全性和可靠性提供双重保障, 为软件系统开发带来新的机遇. 一方面, 相较传统非安全编程语言, Rust 所有权模型和借用检查机制提供的安全保障简化了程序验证的推理过程, 另一方面, 通过融合严谨的形式化验证方法, 不仅能够证明软件系统的功能正确性等复杂特性, 还能够为与不安全 Rust 代码的交互过程提供严格的限制. 本节将列举近年来针对 Rust 软件系统形式化验证的实例, 展示其在实际开发中的应用情况和带来的益处. 一些工程实践中 Rust 验证的应用实例由于缺乏公开的学术论文和开源代码, 难以全面掌握这些工作的细节, 因此本文不对其进行详细讨论. 本节最后将综合分析验证案例, 并结合作者对验证工具的实际使用经验, 提出工具选择建议供用户参考. 随着 Rust 语言和相关验证工具链的不断成熟, 在未来应该可以看到更多类似的工作.

5.1 针对 Rust 系统的自动化验证

RedLeaf[59]是基于 Rust 编程语言隔离的微内核架构操作系统, 首次提出利用 Rust 的性质对 Rust 操作系统进行高效的形式化验证. RedLeaf 开发基于 SMACK 校验器、Boogie 中间验证语言和 Z3 SMT 求解器的验证工具链, 提供 Floyd-Hoare 风格的模块化验证, 要求开发人员以前置条件、后置条件和循环不变式的形式手动为程序标注过程合约. 基于 Rust 所有权系统, 框架规则可以根据函数签名自动推导, 可变引用不会互相别名, 验证工具链可利用这些信息减少用户需要标注的堆不变式, 并通过将程序堆建模从传统的字节数组替换为一组有类型的变量, 以提高验证效率和可扩展性. 该研究作为先导性工作提出通过 Rust 的所有权系统减轻传统验证器的负担, 但是并没有给出验证需要的工作量.

Brun 等人[60]指出, 尽管现在有不少经过验证的操作系统, 但这些系统的规约并不能与应用程序的验证结合起来, 给出一个端到端验证的高性能软件栈. 他们根据在操作系统上运行的应用程序的行为定义操作系统的高级规约, 并将其分为两类: 操作系统提供的执行模型 (反映内存和 CPU 的虚拟化); 支持的系统调用 (网络、文件系统以及并发原语). 然后基于 Verus 对一个 Rust 开源操作系统 NrOS[61]的页表管理模块进行了功能正确性验证. 他们设计了描述硬件环境的硬件规约和描述映射、解映射和地址解析行为的高级规约, 然后证明, 在预定硬件环境中运行, 页表实现可以精化高级规约.

Chen 等人[62]使用 Rust 开发了类 L4 的微内核 Atmosphere, 并基于 Verus 进行验证. 该工作首先定义了高层规约描述 Atmosphere 接口的行为, 再证明系统调用的实现是高层规约的精化. 他们还改造了 Verus: 不再使用未经验证的通用内存分配器, 而是基于自己实现的验证安全的简单内存分配器; 设计了一个过程宏以支持高效的域可变借用. 证明代码与原始代码比率为 7.5:1, 低于之前的方法.

Ijaz 等人[63]提出了语内设计 (intralingual design) 的概念, 即利用 Rust 的类型系统, 在编译期保证部分不变式, 从而减少需要交给约束求解器的验证条件. 这可以用来保证系统资源管理的正确性, 即用一种类型 (不实现 copy 特征, 字段私有) 表示一种系统资源, Rust 的类型规则就可以适配到该系统资源: 使用 Rust 类型状态 (typestate) 特性实现访问控制, 通过所有权转移或者借用实现资源委托, 系统资源自动回收; 也可以用来表明一个操作是否完成: 让执行操作的函数返回一个特定类型的实例 (但是需要确保只有该函数会实例化这种类型), 如果要保证一个操作序列, 可以让下一个函数的参数包含该特定类型, 消费掉这个实例. 但是语内设计的表达能力有限, 如不能证明算术属性. 另外, 尽管 Rust 的类型系统保证一种类型的不同实例 (内存对象) 是不相交的, 但不能保证这些实例对应的系统资源也是不相交的. 他们将该设计应用到开源 Rust 操作系统 Theseus[1]的内存子系统, 并且对页框分配器使用 prusti 验证其返回的页不会重叠. 由于约束求解器只需要对页框分配器返回页框的唯一性进行判定, 所以最终的证明代码与原始代码比率为 1:76, 非常轻量化.

文件系统最重要的属性之一是在崩溃或断电时保持其完整性和用户数据, 构建崩溃一致性文件系统是一件很有挑战性的任务. SquirrelFS 文件系统[64]引入了一种新的崩溃一致性机制–同步软更新 (synchronous soft updates), 将崩溃安全归结为对文件系统元数据的更新之间的强制排序. 在 SquirrelFS 中, Rust 语言的类型状态 (typestate) 特性被用来实现这一排序. 每个持久性结构都有其对应的类型状态, 这些类型状态定义了结构可能处于的一系列状态, 并且通过 Rust 的类型系统来保证文件系统操作按照正确的顺序执行. 这种方法不仅提高了代码的可读性和可维护性, 而且通过编译时检查减少了运行时错误的可能性. 此外, SquirrelFS 的开发团队还使用了 Alloy 建模语言来对文件系统设计进行建模, 并保持 Rust 中的类型状态模型与 Alloy 模型在设计上的一致性; 通过在 Alloy 中运用模型检查的方法, 验证了通过类型状态强制的操作顺序确实能够满足崩溃一致性的要求.

HyperEnclave[65]是基于高特权软件 (hypervisor) 的可信执行环境, 通过模拟 TEE 指令集兼容已有 TEE 生态 (如 SGX), 并且已经在蚂蚁集团得到部署. Dai 等人[66]设计了一个模块化验证框架 MIRVerif, 验证 HyperEnclave 的内存隔离机制实现的正确性. MIRVerif 可根据 Rust MIR 的轻量级语义验证 Rust 系统代码. 他们修改了 Rust 编译器, 将改造过的 HyperEnclave 内存管理模块代码转译进而嵌入到 Coq, 得到实现层的规约, 沿用 CertiKOS 和 SeKVM 使用的 CCAL 方法, 基于函数调用图将涉及的函数分为 15 层, 并且增加了对不同层之间指针参数传递的支持, 逐层求精, 最终证明实现层精化抽象层. 在抽象层上定义了系统状态迁移的操作 (内存读写和 hypercall), 证明了页表映射满足的不变式, 进而证明了系统的无干扰性. 该工作首次用形式化方法对一个工业界实际部署的 Rust 系统进行功能正确性验证. 但他们对于架构相关代码以及第三方库的代码选择相信其正确性并不加以验证, 另外, 他们对 RustMIR 的语义形式化依靠人工阅读文档校验.

AMD SEV-SNP[67]机密虚拟机架构可以创建虚拟机级别的可信执行环境, 其中, 运行在高虚拟机特权级别 (VMPL) 的安全模块可为客户机提供远程认证、完整性保护、中断注入监控等关键安全功能. VERISMO[68]是第一个功能正确性、信息流安全性经形式化验证的安全模块, 基于 Verus 完成了验证过程. 该工作将证明分为两层. 机器模型层定义了抽象机器模型 ψ, 其中包含如寄存器、内存、页表与 RMP 等硬件资源, 定义了 hypervisor、VMPL0 安全模块、VMPL3 客户机这 3 种实体及对应的操作原语 (内存操作、RMP 操作、页表操作等), 证明了客户机的私有内存不能被其他实体读取或更改、VMPL3 的客户机内核不能读取或修改 VMPL0 安全模块的私有内存, 以及客户机地址翻译的正确性, 保证了客户机的机密性与完整性. 实现层扩展了 Verus 的内存权限, 并且将权限的概念扩展到锁和寄存器, 以证明功能正确性, 为 VERISMO 的变量维护猜测空间, 证明密钥传播时相对其他实体始终为机密变量. VERISMO 的验证只需 6 min, 验证过程中在 AMD 官方安全模块中发现恶意的 hypervisor 可以将 VMPL0 的信息暴露给 VMPL3, 并将该漏洞上报并得到确认.

5.2 Rust 安全关键应用的自动化验证

Rust 形式化验证在安全关键型应用的实例, 重点探讨其在可信计算、机密计算、云存储服务等方面的贡献. 这种结合形式化验证的方法, 不仅提升了 Rust 在高安全性能领域的应用前景, 也为确保现代计算系统的可靠性和安全性打下了坚实基础.

加密库是现代应用程序可信计算基础的核心, 形式化验证能系统地防止加密软件中的缺陷, HACL-rs 项目[69]的目标是提供一个经过验证的、纯粹的 Rust 实现的加密原语库来确保安全性和可靠性, 致力于将底层使用 F*语言编写经过严格形式化验证的密码库 HACL*转化为纯 safe Rust 代码. HACL-rs 计划取代当前 libcrux 库中使用的 HACLC 代码部分从而消除 C 语言外部函数接口 (FFI) 的依赖, 使 libcrux 成为一个完全使用安全 Rust 编写的库, 提升代码的整体安全性并简化开发和维护过程. 在 HACL-rs 项目的推动下, Zulip 团队[70]设计了一种用于为密码构造编写简洁、可执行、形式化规范的语言 HACSPEC, 从语法上看, HACSPEC 是安全 Rust 的一个纯函数子集, 它不允许可变引用, 因此避免了有状态 Rust 程序的大部分复杂性. Rust 应用可以直接使用 HACSPEC 代码作为所需加密组件的原型实现, Rust 强大的类型安全保证意味着未经验证的应用程序中的错误不会影响经过验证的加密代码的正确性或安全性, 此外 HACSPEC 还能自动转换为 F*, Coq 和 EasyCrypt 语言. Hax[71]项目进一步扩展了 HACSPEC 的范围, 成为一种用于验证 Rust 程序的工具, Hax 能够将 Rust 的大部分子集翻译为 F*或 Coq 等形式化语言, 但不支持包含 unsafe 代码块和可变引用返回类型的代码, 通过 Hax 开发者可以更为可靠地验证和确保 Rust 程序的安全性.

机密计算即服务 (CCaaS) 是一种新兴的服务模式, CCaaS 需要向数据提供商保证, 服务不会将其隐私泄露给任何未经授权的一方, 并在服务完成后清除其数据. Chen 等人[72]在 Coq 中形式化地定义了安全目标 PoBF, 包含无泄漏 (no leakage) 和无残留 (no residue), 并证明了在哪些安全约束下可以满足 PoBF, 这些约束指导 PoBF 兼容框架 (PoCF) 的设计实现. PoCF 包括 Rust 语言实现的用于不同硬件 TEE 的通用库、CCaaS 原型 enclaves 和用于证明 PoBF 满足性的验证器, 它利用 Rust 的强大类型系统和安全特性构建了一个具有隐私保护契约的验证状态机, 由 Rust 编译器检查内存和类型安全性, 由 Rust 自动验证工具 Prusti 保证类型状态实现与其规范的一致性, 最后 MIRAI[73]进行污点分析, 确保 CC 任务的函数调用参数不包含敏感数据结构, 防止机密信息泄露. 实验表明 PoCF 所引入的保护机制在运行时的性能开销很小.

Bornholt 等人[14]在 Amazon S3 云对象存储服务中对一种新的键值存储节点实现的正确性进行了验证, 采用了轻量级形式化方法, 这种方法结合了基于属性的测试和无状态模型检查, 将系统的预期正确性分解为独立的属性, 并分别利用最适合的工具检测不同属性. 在并发代码的验证过程中, 他们特别采用了两种无状态模型检查器: Loom 和该文章提出的 Shuttle. Loom 被用于可靠地检查小型但关键的并发代码的所有可能交错情况, 例如自定义的并发原语. 而 Shuttle 则用于随机检查那些 Loom 无法扩展到的大型测试场景中的交错情况. 作为规范, 他们使用了一个可执行的参考模型, 该模型用 Rust 语言编写, 定义了系统中组件允许的顺序和无崩溃的行为. 这种选择使得工程团队能够更容易地编写和维护参考模型, 因为它与实现系统的语言相同. 他们的轻量级形式化方法通过测试实现, 其中形式化验证测试与常规单元测试类似, 都是使用 Rust 的内置测试基础设施来运行. 这种方法不仅提高了验证过程的效率, 而且使得形式化验证与日常开发流程无缝集成. 通过这项工作, 他们成功地阻止了 16 个潜在问题进入生产环境, 这些问题包括崩溃一致性和并发性问题. 这一成果表明, 轻量级形式化方法不仅能够有效地发现和预防生产环境中的问题, 而且随着系统的演化, 这种方法还可以不断地扩展和改进.

5.3 Rust 自动化验证工具使用建议

(1) 形式化与非形式化方法的结合: 当前许多 Rust 程序验证案例中不仅采用形式化方法, 还融入了如污点分析、模糊测试、基于属性的测试、编译器静态检查等其他非形式化技术. 例如, 亚马逊在验证 S3 云对象存储服务时[14], 结合了基于属性的测试和无状态模型检查, 将系统预期的正确性分解为独立的属性, 并利用最适合的工具分别检测这些属性. 在对 CCaaS 的验证过程中, 除了利用 Prusti 进行类型状态实现与规范的一致性检查外, 还进行污点分析以防止机密信息泄露. 在验证 Rust 操作系统 Theseus 内存子系统时[63], 为验证物理内存和虚拟内存之间

的双射映射性质, 该工作主要基于 Rust 类型状态特性实现资源访问控制, 并描述成编译期间检查的不变式, 而仅对不能使用该方法描述的部分进行了半自动化验证. 在 SquirrelFS 中[64], 同样利用了 Rust 语言的类型状态特性实现编译时检查. 此外, 该工作使用 Alloy 建模语言对 SquirrelFS 文件系统在设计层建模, 并确保 Rust 类型状态模型与 Alloy 模型在设计上的一致性, 同时在 Alloy 中运用模型检查的方法验证了高层性质. 蚂蚁集团近期公布的"星绽"操作系统中[74], 除了使用形式化验证外, 还通过模糊测试对安全的 Rust 代码部分进行测试, 以有效发现未处理的 panic.

(2) 综合使用多种形式化工具: 鉴于 Rust 语言的验证工具尚处于发展初期, 支持的功能特性有限, 一些 Rust 程序验证案例中会对现有工具进行扩展以满足特定的验证需求. 例如, 在 Kubernetes 管理控制器的验证中[75], 通过扩展 Verus 并结合 TLA+来支持活性 (liveness) 验证, 这要求用户对该工具有深入了解. 相对地, 有些研究可能会采用不同的工具来验证程序的不同属性. 例如"星绽"操作系统中, 利用 Verus 对内存管理系统进行建模和验证, 证明了代码正确实现了内存映射和页面管理等功能. 同时, 该工作没有扩展 Verus, 而是直接使用 TLA+验证了星绽中的并发原语, 确保了并发的安全性 (safety) 和活性 (liveness). 亚马逊在验证 S3 云对象存储服务并发执行情况时, 利用 Loom 可靠地检查小型但关键的并发代码的所有可能交错情况, 例如自定义的并发原语; 利用 Shuttle 随机检查那些 Loom 无法扩展到的大型测试场景中的交错情况.

(3) 单一形式化工具的选择建议: 从文档维护、社区活跃度、工具易用性及特性支持度等方面考虑, 作者建议用户优先选择 Verus、Kani 和 Prusti. 相比之下, 一些形式化工具尚未开源 (如 Gillian-Rust, TRustPN), 或仍处于科研探索阶段而支持的功能特性不足 (如 VeriFast), 或没有后期的维护和扩展 (如 Electrolysis, Crust, SMACK), 或没有稳定版本和清晰的指导手册 (如 Creusot). 这些研究工作虽然在 Rust 形式化验证领域提供了很好的借鉴和启发意义, 但不适合直接用于实际工程项目的验证. Verus、Prusti 和 Kani 这 3 个工具在近年来均有实际的验证案例, 且文档维护和使用体验方面表现较好. 其中, Prusti 仅支持 safe Rust 的验证, 且在验证速度上较慢, 但在 safe Rust 特性的支持上有自己的优势 (如 Verus 不支持如图 12 函数返回可变引用的情况). Kani 和 Verus 均支持 unsafe Rust 程序的验证, 但前者支持范围更小 (如图 6 的例子, Verus 不支持验证变量直接取地址的情况, 而 Kani 则可验证并发现指针访问越界的问题) 且可能需要修改源代码以使用自己的库 (如抽象原始指针 API), 而 Kani 则受到循环展开次数约束等限制, 且仅支持插桩简单的验证条件, 不能像半自动化工具一样提供灵活的模块化功能正确性验证 (如图 19 的对比). 如后文第 7 节所述, 并发程序的验证是有挑战性的工作, Loom 和 Verus 提供了一定程度的支持.

6 Rust 形式化验证的技术挑战

现有的 Rust 验证工作已取得一些进展和研究成果, 但仍然面临以下技术挑战.

(1) 目前, Rust 语言尚未完全标准化, Rust 社区尚未形成一份统一精准的规范文档. 这意味着, 对于 Rust 应该实现哪些功能以及 Rust 编译器应展现怎样的行为, 目前还没有一个明确的定义. 这与 C 和 C++等已经过标准化过程的编程语言形成对比. C 语言的发展历程中, 从最初的 ANSI C (即 C89) 到后续的 ISO C (C90)、C95、C99、C11 乃至 C17, 每一个版本都伴随着国际认可的标准文档, 确保了不同编译器和平台上编程体验的一致性. Rust 的这种状况对 Rust 的形式化验证领域产生了显著的影响.

首先, 缺乏一个统一的规范文档使得无法对 Rust 的操作语义和内存模型进行完全精确的形式化描述. 当前一些验证工作的做法是将 Rust 编译为 C 类似的表示, 直接使用 C 原有的内存模型以绕过该问题然而, Rust 有一些 C 没有的新特性, 例如非常规模数类型 (exotically-sized types) 如零大小类型 (zero-sized types) 以及多态性等. 再加上后文展开的 Rust 编译器利基优化技术, 这些特性使得直接应用 C 语言验证工具中现有内存模型变得复杂且具有挑战性. 一些工作通过转化 Rust 为纯函数式的表示, 避免显式描述内存模型, 但该方法目前仅限于 safe Rust 子集. 还有一些工作, 如 RustBelt, 引入 Rust 模型语言 (而非真实的 Rust 语言) 并描述其操作语义, 同时使用简化的内存模型表示. 这种方法在理论上提供了对 Rust 语言核心概念的理解, 但不能直接用于处理真实 Rust 程序.

其次, 由于缺少规范, 现有的验证工具难以独立于 rustc 开发. 它们往往需要集成 rustc 的部分代码或依赖其特定的实现细节, 这限制了验证工具的通用性和可移植性. 而且, 由于 rustc 本身处于积极开发状态, 其内部实现经常发生较大变化, 这导致验证工具需要不断适应这些变化, 进而影响了工具的稳定性和可靠性. 在缺乏规范的背景下, 某些 rustc 的实现行为甚至难以评估其合理性. 例如, Rust 编译器可能会采用一些特殊的利基优化手段, 在不影响表达能力的情况下调整结构体的内存布局以减小其整体大小. 例如, 两个结构体 struct A {u32 x; u64 y}; struct B {u32x; u64 y}; 在 rustc 里可能有不同的二进制表示方式, A 有可能编译成先 x 后 y, 而 B 则是先 y 后 x. 在处理某些底层指针操作, 如堆操作时, 结构体的内存布局就显得尤为关键. 正确的内存布局对于确保程序的正确性和安全性至关重要. 然而, 由于缺乏明确的规范, 我们无法断言哪种布局是正确的, 这导致了在进行形式化验证时存在不确定性. 针对这个问题, Gillian-Rust 通过自己的符号式编码方式来编码地址, 使得其是内存布局不相关的 (layout-independent). RefinedRust 也提出任意的布局算法 (arbitrary layout algorithm) 对其验证过程进行了参数化, 确保验证的正确性不受具体数据布局的影响. 而 Kani 实现为 rustc 的插件, 每次验证时从 rustc 获取编译后生成的具体信息, 包括当前确定的数据结构布局.

然而, 定义 Rust 语言规范并非易事. 现有的资源如 Rust 参考手册、标准库文档、非安全代码指南等, 都不够完整或不适宜作为权威参考. 一个准确的定义需要深入考虑 Rust 的设计原则、语言特性、生态系统和社区实践. 在过去一两年中, Rust 社区已经开始了一些旨在构建规范的项目. 例如, Ferrocene[76]语言规范项目主要用于安全攸关系统, 它提供了一个全面的框架, 覆盖了从语法到名称解析, 再到编译器的全面功能. Ferrocene 规范是用英语以公理化的风格编写的, 类似于 C/C++规范的描述形式. 此外, a-mir-formality[77]项目致力于构建 Rust 类型系统的形式化模型, 特别是对 Rust 语言借用检查器的规则进行形式化定义. 而 MiniRust[78]项目提供一个轻量级 Rust 语言子集的操作语义, 描述 Rust 核心语言程序在执行时可能表现出的行为. 每个项目都有其独特的侧重点, 如何将这些项目与现有的 Rust 官方文档整合, 形成一套连贯、全面的规范, 是一个亟待解决的问题. 我们可以看到, 形式化语言的严谨性和无歧义性使其成为描述 Rust 规范的一个理想工具. 而一旦 Rust 规范得到明确定义, 它将反过来促进形式化验证的发展.

(2) 对 Rust 标准库的依赖性在形式化验证中带来了额外的考量. 因为它要求验证过程不仅局限于用户编写的代码, 还需涵盖标准库的实现, 以确保整个程序的正确性. 目前, 许多现有的验证工具在进行程序验证时, 通常采取信任标准库的策略, 并不对其进行深入验证. 这不仅降低了验证的精确程度, 而且和依赖于特定的编译器版本一样, 依赖于特定标准库版本也限制了验证器的通用性, 还增加了后期维护工作的难度. 有些工具, 如 Verus, 通过重写部分标准库并提供一套抽象化的 API, 使得它们验证的程序基于这些抽象 API. 在验证某程序时, 如何清晰描述并验证标准库接口的功能规约是需要解决的问题.

(3) Safe Rust 和 unsafe Rust 程序的联合验证是一项挑战性的工作. 安全 Rust 拥有许多高级特性, 例如高阶函数、迭代器等高级抽象, 这些特性为开发者提供了便利和强大的表达能力, 但为自动化验证增加了难度. 而另一方面, 不安全 Rust 中则涉及复杂且底层原始指针等操作, 且由于没有编译器的保障, 需要开发者确保所有的内存操作都是合法的而不会导致段错误, 这就需要对内存地址的有效范围进行推理. 为了开发同时处理两者的工具需要对 Rust 语言特性深入的理解, 还需要在设计上实现一种精巧的平衡, 既能提供对底层操作低级别的精准控制, 又能自动高效地处理安全 Rust 中使用的高级特性, 可能还需要确保 unsafe Rust 不会破坏 safe Rust 的安全保障并利用该安全保障来简化验证流程.

7 未来工作展望

结合面临的挑战与当前的研究进展, 本文认为 Rust 形式化验证方向未来值得关注的研究工作主要包括以下几方面.

Rust 全语言的形式化语义建模: 有的方法建模了 Rust 语义的一个子集, 而不是实际的 Rust 语言, 无论是在 Rust 的表面语法层面还是在中间表示 (MIR) 层面的描述, 都包含了简化的假设来处理语言的复杂性. Rust 作为一种多范式编程语言, 它巧妙地融合了命令式、函数式、面向对象和泛型编程等多种编程范式. 在这样的背景下, 探

索如何在一个统一的框架内对这些编程范式进行综合的形式化建模, 成为程序语言研究领域的一个重要且具有挑战性的课题. 这不仅能够推动 Rust 语言本身的理论发展, 也能为理解和应用 Rust 提供更深层次的洞见. 然而, 鉴于前文提到的 Rust 语言缺乏统一规范的问题, 这一研究过程需要 Rust 验证人员与 Rust 社区的开发者进行紧密的交流与合作, 共同推进.

Rust 自动化验证工具的优化及提升: 在自动化验证工具的研发领域, 扩展对 Rust 语言更多特性的支持是提升验证准确性和实用性的关键. 鉴于 Rust 的许多语义特性尚缺乏官方规范, 实现工具与现有及开发中的 Rust 规范的整合显得尤为重要. 此外, 不同的验证技术各有优势, 适用于不同的场景. 探索如何将这些技术融合到统一的框架之中, 实现优势互补, 对于拓宽 Rust 验证领域的研究视野和应用范围具有重大意义. 如表 2 所示, 不同的验证工具针对不同的语言特性和验证性质提供了不同程度的支持. 尽管目前 Rust 的验证研究在处理某些技术时存在一定程度的相互借鉴, 但各研究团队或项目大多独立开展工作, 且在测试过程中因针对不同的测试集而缺少直接的比较. 因此, 对现有技术进行全面的比较分析, 深入探讨不同工具在性能、功能支持和准确性等方面的表现, 并在此基础上进行优势整合, 对于推动 Rust 在学术界和工业界的应用至关重要. 例如, 在统一验证框架中, 验证不同性质, 如内存安全性, 功能正确性时可以分别使用不同的工具; 验证包含 safe 和 unsafe 代码的程序, 使用不同工具分别验证 safe 和 unsafe 的部分并保证语义一致性和结果整合后整体的正确性.

对非安全 Rust 代码的验证: 形式化验证能够帮助开发者更安全、更有效地使用 unsafe 代码. 然而现有研究仅触及了冰山一角, 大多停留在学术概念验证阶段. 要实现其在工业级别的广泛应用, 仍需跨越重大的技术障碍. 此外, 已有针对 unsafe 代码的验证主要涉及两方面. 一方面是针对原始指针, 即确保任何时候该指针的访问都是合法和安全的. 另一方面是针对 Rust 的核心库调用一些底层不安全函数的场景, 例如直接使用原始指针进行内存分配和释放的函数或将 Rust 类型强制转换为其他任意类型的函数. 实际上, Rust 中还有其他一些 unsafe 操作, 比如利用 Rust 中调用不安全函数的能力 (FFI) 实现与 C 代码互操作, 这在 Rust 的 unsafe 代码使用中非常普遍且存在安全隐患 [79]. 任何提供 C 接口的外部库中的函数都可以作为不安全函数从 Rust 中调用, 这为跨语言集成提供了便利. 然而, 一个关键的挑战是如何确保这些函数在跨不同编程语言编写的程序中不会违反 Rust 类型系统所提供的安全保证. 这是因为, 尽管 Rust 的类型系统在语言内部提供了内存安全保证, 但当涉及外部 C 函数时, 这些保证可能不再适用. Rust 社区探索了很多方法来处理跨语言调用的问题, 包括提供更安全的抽象、改进的文档和工具支持, 而形式化验证也可以帮助确保所有的外部调用都符合 Rust 的安全性原则. 汪宇霆 [80]正在进行的验证工作是该研究领域的一项探索性尝试. 该研究通过设计 Rust 核心语言的开放语义和语言接口并设计带所有权的语义接口, 作为 safe Rust 和 unsafe Rust 的交互协议. 然后再将该语义接口与编译器的语义接口整合, 获得支持分离编译的语义接口, 借助 CompCert 分离编译框架验证 safe Rust 函数和 unsafe C 函数之间是否正确交互, 并保证完整程序编译和链接后的正确性. 而形式化可以用于的另一种场景是对静态可变变量的 unsafe 代码的逻辑推理. Rust 语言中的静态可变变量 (static mut) 允许创建全局可变状态, 这使得多个线程能够访问并可能同时修改这一全局数据. 在这种情况下, 开发者需要采取额外的预防措施来预防数据竞争的发生.

对并发 Rust 代码的验证: 目前针对并发 Rust 程序的形式化验证工作较少. 现有工作中, RustBelt 基于并发分离逻辑可以支持 Rust 中并发程序的证明, 证明了 Mutex, RwLock 等同步原语库的线程安全, 但是该工作没有开发基于并发分离逻辑的 Rust 语言程序的自动化验证工具; 现有基于模型检测的并发程序自动验证方法只针对死锁等部分问题, 还有大部分空缺亟需填补, 例如 unsafe 代码中操作共享内存造成的数据竞争问题, 消息传递造成的与预期不符合的时序逻辑错误等. 实际上, 不仅限于形式化验证领域, 从整体上看, 现有对 Rust 安全研究的工作也主要还是以研究 Rust 内存安全为主, 对 Rust 并发安全的研究有所欠缺. 但在现实 Rust 软件中, 并发漏洞的数量仅次于内存漏洞 [81]. Rust 在编译时防止了许多潜在的并发错误, 然而, 并发编程在涉及复杂的生命周期、互斥锁的死锁或异步任务的协作等复杂的并发场景中仍然充满挑战, 需要开发者小心设计和实现以避免逻辑错误. 形式化的验证方法使用数学对程序和系统进行精确的建模和严格的验证, 这对于构建高可靠性、高性能的并发系统尤为重要. 然而, 并发技术的特性也决定了其验证是相对困难的. 由于组件间的相互影响, 一般难以对单独一个组件的正确性进行验证, 这也导致了在规模较大的系统中, 无法使用简单的验证方法进行模块化的验证. 因此, 开发能够支

持模块化并具有良好可扩展性的 Rust 程序并发验证方法, 是一个重要课题.

对 Rust 编译器的形式化验证: Rust 语言因其内存安全的保证, 被认为是开发安全关键型应用的理想选择. 但是, 目前基于 LLVM 的 Rust 编译器工具链并未提供足够的安全验证, 这限制了 Rust 在高安全需求场景下的应用潜力. 尽管社区内对于开发一个经过验证的 Rust 编译器工具链的呼声日益高涨, 但这一目标的实现尚未取得显著进展. 一个可能发展方向是将 Rust 编译到 Vellvm[82]或 CompCert 的 Clight[83]中间表示, 可以复用一部分现有的经过验证的编译器工具链. 这种验证工作不仅能够增强 Rust 编译器本身的可靠性, 也将为构建更为安全的软件生态系统提供坚实的基础.

大模型驱动的 Rust 形式化验证: 近年来, 各大语言模型 (large language model, LLM) 蓬勃发展, 形式化验证领域的许多学者也开始积极探索大语言模型的应用潜力. 这些模型在代码理解和自然语言到形式化规范语言的转换方面展现出巨大潜力. 利用大语言模型来解释代码的功能、约束和预期行为, 有助于降低形式化专家学习 Rust 语言的成本, 并在人工指导下将这些知识转化为形式化规范. 目前, 学者们已经成功地将大语言模型应用于提高定理证明过程的自动化水平. 例如, Song 等人[84]设计大型语言模型辅助的 Lean Copilot 框架用于证明定理过程中的自动化, 能够实现证明步骤建议、自动完成中间证明目标、选择相关前提条件等, 实验结果表明, 这种方法在定理证明过程中相较于传统的基于规则的自动化方法, 更有效地辅助了人工操作, 提高了自动化程度. Wu 等人[85]和 Jiang 等人[86]也分别针对 Isabelle/HOL 利用大语言模型将数学问题翻译为形式化语言实现自动化定理证明. 尽管这些尝试取得了显著进展, 但目前大语言模型尚未能够解决复杂实际系统应用程序的形式化验证问题. 未来如果能构建一个专门针对 Rust 程序及其验证的大规模数据库, 用于训练大语言模型, 将有望扩展其在 Rust 形式化验证中的应用范围, 并提高验证的准确度. 目前, 将大语言模型与 Rust 验证工具相结合的研究还处于初期阶段. 尽管已有尝试, 如将 ChatGPT 与 Kani[87]或 Verus[88]等验证工具结合, 辅助编写断言条件等程序属性, 但这些尝试仍然有限. 如果能够将大语言模型更深入地集成到 Rust 验证框架中, 实现轻量级的形式化验证或混合代码审查与测试, 将有望形成一个更加完整、便捷且高度自动化的 Rust 形式化验证生态系统, 从而提升形式化验证的实用性和影响力.

References:

[1] Boos K, Liyanage N, Ijaz R, Zhong L. Theseus: An experiment in operating system structure and state management. In: Proc. of the 14th USENIX Conf. on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2020. 1.

[2] Redoxfs. 2017. https://gitlab.redox-os.org/redox-os/redoxfs

[3] Lyu S, Rzeznik A. Going serverless with the Amazon AWS Rust SDK. In: Lyu S, Rzeznik A, eds. Practical Rust Projects: Build Serverless, AI, Machine Learning, Embedded, Game, and Web Applications. 2nd ed., Berkeley: Apress, 2023. 201–246. [doi: 10.1007/978-1-4842-9331-7_6]

[4] Database drivers and ORMs for Rust that are ready for production. 2020. https://blog.logrocket.com/11-database-drivers-and-orms-for-rust-that-are-ready-for-production

[5] Servo. 2017. https://github.com/servo/servo

[6] Smoltcp. 2017. https://github.com/smoltcp-rs/smoltcp

[7] Solana. https://solana.com/zh

[8] Xu H, Chen ZB, Sun MS, Zhou YF, Lyu MR. Memory-safety challenge considered solved? An in-depth study with all Rust CVEs. ACM Trans. on Software Engineering and Methodology (TOSEM), 2022, 31(1): 3. [doi: 10.1145/3466642]

[9] Mergendahl S, Burow N, Okhravi H. Cross-language attacks. In: Proc. of the 29th Network and Distributed System Security Symp. San Diego: The Internet Society, 2022. 1–17.

[10] system-pclub. 2020. https://github.com/system-pclub/rust-study

[11] Hu S, Hua BJ, Ouyang WR, Fan QL. A survey of Rust language security research. Journal of Cyber Security, 2023, 8(6): 64–83 (in Chinese with English abstract). [doi: 10.19363/J.cnki.cn10-1380/tn.2023.11.06]

[12] Cui M, Chen C, Xu H, Zhou Y. SafeDrop: Detecting memory deallocation bugs of Rust programs via static data-flow analysis. ACM Trans. on Software Engineering and Methodology, 2023, 32(4): 1–21.

[13] Jung R, Jourdan JH, Krebbers R, Dreyer D. RustBelt: Securing the foundations of the Rust programming language. Proc. of the ACM on

Programming Languages, 2018, 2(POPL): 66. [doi: 10.1145/3158154]

[14] Bornholt J, Joshi R, Astrauskas V, Cully B, Kragl B, Markle S, Sauri K, Schleit D, Slatton G, Tasiran S, van Geffen J, Warfield A. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In: Proc. of the 28th ACM SIGOPS Symp. on Operating Systems Principles. ACM, 2021. 836–850. [doi: 10.1145/3477132.3483540]

[15] Gäher L, Sammler M, Jung R, Krebbers R, Dreyer D. RefinedRust: A type system for high-assurance verification of Rust programs. Proc. of the ACM on Programming Languages, 2024, 8(PLDI): 192. [doi: 10.1145/3656422]

[16] Ayoun SÉ, Denis X, Maksimović P, Gardner P. A hybrid approach to semi-automated Rust verification. arXiv:2403.15122, 2024.

[17] Polonius. 2019. https://rust-lang.github.io/compiler-team/working-groups/polonius

[18] Huet GP, Kahn G, Paulin-Mohring C. The Coq proof assistant: A tutorial: Version 6.1. 1997. https://inria.hal.science/file/index/docid/69967/filename/RT-0204.pdf

[19] Matsushita Y, Denis X, Jourdan JH, Dreyer D. RustHornBelt: A semantic foundation for functional verification of Rust programs with unsafe code. In: Proc. of the 43rd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. San Diego: ACM, 2022. 841–856. [doi: 10.1145/3519939.3523704]

[20] Weiss A, Gierczak O, Patterson D, Ahmed A. Oxide: The essence of Rust. arXiv:1903.00982, 2021.

[21] Lehmann N, Geller AT, Vazou N, Jhala R. Flux: Liquid types for Rust. Proc. of the ACM on Programming Languages, 2023, 7(PLDI): 169. [doi: 10.1145/3591283]

[22] Payet É, Pearce DJ, Spoto F. On the termination of borrow checking in featherweight Rust. In: Proc. of the 14th Int'l Symp. on NASA Formal Methods. Pasadena: Springer, 2022. 411–430. [doi: 10.1007/978-3-031-06773-0_22]

[23] Reed E. Patina: A formalization of the Rust programming language. Technical Report, UW-CSE-15-03-02, Washington: University of Washington, 2015.

[24] Jung R, Dang HH, Kang J, Dreyer D. Stacked borrows: An aliasing model for Rust. Proc. of the ACM on Programming Languages, 2020, 4(POPL): 41. [doi: 10.1145/3371109]

[25] Wang F, Song F, Zhang M, Zhu XR, Zhang J. KRust: A formal executable semantics of Rust. In: Proc. of the 2018 Int'l Symp. on Theoretical Aspects of Software Engineering. Guangzhou: IEEE, 2018. 44–51. [doi: 10.1109/TASE.2018.00014]

[26] Kan SL, Chen Z, Sanan D, Lin SW, Liu Y. An executable operational semantics for Rust with the formalization of ownership and borrowing. arXiv:1804.07608, 2020.

[27] Cardelli L. Type systems. ACM Computing Surveys (CSUR), 1996, 28(1): 263–264. [doi: 10.1145/234313.234418]

[28] Igarashi A, Pierce BC, Wadler P. Featherweight Java: A minimal core calculus for Java and GJ. ACM Trans. on Programming Languages and Systems (TOPLAS), 2001, 23(3): 396–450. [doi: 10.1145/503502.503505]

[29] Jung R, Swasey D, Sieczkowski F, Svendsen K, Turon A, Birkedal L, Dreyer D. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Mumbai: ACM, 2015. 637–650. [doi: 10.1145/2676726.2676980]

[30] Jung R, Krebbers R, Jourdan JH, Bizjak A, Birkedal L, Dreyer D. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming, 2018, 28: e20. [doi: 10.1017/S0956796818000151]

[31] Dang HH, Jourdan JH, Kaiser JO, Dreyer D. RustBelt meets relaxed memory. Proc. of the ACM on Programming Languages, 2020, 4(POPL): 34. [doi: 10.1145/3371102]

[32] Sammler M, Lepigre R, Krebbers R, Memarian K, Dreyer D, Garg D. RefinedC: Automating the foundational verification of C code with refined ownership types. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. Virtual: ACM, 2021. 158–174. [doi: 10.1145/3453483.3454036]

[33] Astrauskas V, Müller P, Poli F, Summers AJ. Leveraging Rust types for modular specification and verification. Proc. of the ACM on Programming Languages, 2019, 3(OOPSLA): 147. [doi: 10.1145/3360573]

[34] Lattuada A, Hance T, Cho C, Brun M, Subasinghe I, Zhou Y, Howell J, Parno B, Hawblitzel C. Verus: Verifying Rust programs using linear ghost types. Proc. of the ACM on Programming Languages, 2023, 7(OOPSLA1): 85. [doi: 10.1145/3586037]

[35] Denis X, Jourdan JH, Marché C. CREUSOT: A foundry for the deductive verification of Rust programs. In: Proc. of the 23rd Int'l Conf. on Formal Methods and Software Engineering: Int'l Conf. on Formal Engineering Methods. Madrid: Springer, 2022. 90–105. [doi: 10.1007/978-3-031-17244-1_6]

[36] Matsushita Y, Tsukada T, Kobayashi N. RustHorn: CHC-based verification for Rust programs. ACM Trans. on Programming Languages and Systems (TOPLAS), 2021, 43(4): 15. [doi: 10.1145/3462205]

[37] Jacobs B, Smans J, Philippaerts P, Vogels F, Penninckx W, Piessens F. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Proc. of the 3rd Int'l Conf. on NASA Formal Methods. Pasadena: Springer, 2011. 41–55. [doi: 10.1007/978-3-642-20398-5_4]

[38] Toman J, Pernsteiner S, Torlak E. Crust: A bounded verifier for Rust (N). In: Proc. of the 30th IEEE/ACM Int'l Conf. on Automated Software Engineering. Lincoln: IEEE, 2015. 75–80. [doi: 10.1109/ASE.2015.77]

[39] Baranowski M, He SB, Rakamarić Z. Verifying Rust programs with SMACK. In: Proc. of the 16th Int'l Symp. on Automated Technology for Verification and Analysis. Los Angeles: Springer, 2018. 528–535. [doi: 10.1007/978-3-030-01090-4_32]

[40] Kani Rust verifier. 2022. https://github.com/model-checking/kani

[41] Loom. 2021. https://github.com/tokio-rs/loom?tab=readme-ov-file

[42] Zhang KW, Liu GJ. Automatically transform Rust source to Petri nets for checking deadlocks. arXiv:2212.02754, 2022.

[43] Ullrich S. Simple verification of Rust programs via functional purification [MS. Thesis]. Karlsruhe: Karlsruhe Institute of Technology, 2016.

[44] Ho S, Protzenko J. Aeneas: Rust verification by functional translation. Proc. of the ACM on Programming Languages, 2022, 6(ICFP): 116. [doi: 10.1145/3547647]

[45] Reynolds JC. Separation logic: A logic for shared mutable data structures. In: Proc. of the 17th Annual IEEE Symp. on Logic in Computer Science. Copenhagen: IEEE, 2002. 55–74. [doi: 10.1109/LICS.2002.1029817]

[46] Hoare CAR. An axiomatic basis for computer programming. Communications of the ACM, 1969, 12(10): 576–580. [doi: 10.1145/363235.363259]

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

[48] Müller P, Schwerhoff M, Summers AJ. Viper: A verification infrastructure for permission-based reasoning. In: Proc. of the 17th Int'l Conf. on Verification, Model Checking, and Abstract Interpretation. Petersburg: Springer, 2016. 41–62. [doi: 10.1007/978-3-662-49122-5_2]

[49] Foroushaani NR, Jacobs B. Modular formal verification of Rust programs with unsafe blocks. arXiv:2212.12976, 2022.

[50] Fragoso Santos J, Maksimović P, Ayoun SÉ, Gardner P. Gillian, part I: A multi-language platform for symbolic execution. In: Proc. of the 41st ACM SIGPLAN Conf. on Programming Language Design and Implementation. London: ACM, 2020. 927–942. [doi: 10.1145/3385412.3386014]

[51] Nipkow T, Wenzel M, Paulson LC. Isabelle/HOL: A Proof Assistant for Higher-order Logic. Berlin: Springer, 2002.

[52] Avigad J, De Moura L, Kong S. Theorem proving in Lean. 2024. https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf

[53] Clarke E, Kroening D, Lerda F. A tool for checking ANSI-C programs. In: Proc. of the 10th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Barcelona: Springer, 2004. 168–176. [doi: 10.1007/978-3-540-24730-2_15]

[54] Byrnes T, Takashima Y, Jia LM. Automatically enforcing Rust trait properties. In: Proc. of the 25th Int'l Conf. on Verification, Model Checking, and Abstract Interpretation. London: Springer, 2024. 210–223. [doi: 10.1007/978-3-031-50521-8_10]

[55] Thurrott P. Microsoft is rewriting parts of the windows kernel in Rust. 2023. https://www.thurrott.com/windows/282471/microsoft-is-rewriting-parts-of-the-windows-kernel-in-rust

[56] Rust for Linux. 2024. https://github.com/Rust-for-Linux

[57] Jansens D. Supporting the use of Rust in the Chromium project. 2023. https://security.googleblog.com/2023/01/supporting-use-of-rust-in-chromium.html

[58] Stoep JV, Chrome Security Team. Memory safe languages in Android 13. 2022. https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

[59] Narayanan V, Baranowski MS, Ryzhyk L, Rakamarić Z, Burtsev A. RedLeaf: Towards an operating system for safe and verified firmware. In: Proc. of the 2019 Workshop on Hot Topics in Operating Systems. Bertinoro: ACM, 2019. 37–44. [doi: 10.1145/3317550.3321449]

[60] Brun M, Achermann R, Chajed T, Howell J, Zellweger G, Lattuada A. Beyond isolation: OS verification as a foundation for correct applications. In: Proc. of the 19th Workshop on Hot Topics in Operating Systems. Providence: ACM, 2023. 158–165. [doi: 10.1145/3593856.3595899]

[61] Bhardwaj A, Kulkarni C, Achermann R, Calciu I, Kashyap S, Stutsman R, Tai A, Zellweger G. NrOS: Effective replication and sharing in an operating system. In: Proc. of the 15th USENIX Symp. on Operating Systems Design and Implementation. USENIX Association, 2021. 295–312.

[62] Chen XD, Li ZF, Mesicek L, Narayanan V, Burtsev A. Atmosphere: Towards practical verified kernels in Rust. In: Proc. of the 1st Workshop on Kernel Isolation, Safety and Verification. Koblenz: ACM, 2023. 9–17. [doi: 10.1145/3625275.3625401]

[63] Ijaz R, Boos K, Zhong L. Leveraging Rust for lightweight OS correctness. In: Proc. of the 1st Workshop on Kernel Isolation, Safety and

Verification. Koblenz: ACM, 2023. 1–8. [doi: 10.1145/3625275.3625398]

[64] LeBlanc H, Taylor N, Bornholt J, Chidambaram V. SquirrelFS: Using the Rust compiler to check file-system crash consistency. In: Proc. of the 18th USENIX Symp. on Operating Systems Design and Implementation. Santa Clara: USENIX Association, 2024. 387–404.

[65] Jia YK, Liu S, Wang WH, Chen Y, Zhai ZD, Yan SM, He ZY. HyperEnclave: An open and cross-platform trusted execution environment. In: Proc. of the 2022 USENIX Annual Technical Conf. Carlsbad: USENIX Association, 2022. 437–454.

[66] Dai ZY, Liu S, Sjoberg V, Li XP, Chen Y, Wang WH, Jia YK, Anderson SN, Elbeheiry L, Sondhi S, Zhang Y, Ni ZZ, Yan SM, Gu RH, He ZY. Verifying Rust implementation of page tables in a software enclave hypervisor. In: Proc. of the 29th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. La Jolla: ACM, 2024. 1218–1232. [doi: 10.1145/3620665.3640398]

[67] AMD. AMD secure encrypted virtualization (SEV). 2024. https://www.amd.com/zh-tw/developer/sev.html

[68] Zhou ZQ, Anjali, Chen WT, Gong SS, Hawblitzel C, Cui WD. VERISMO: A verified security module for confidential VMs. In: Proc. of the 18th USENIX Symp. on Operating Systems Design and Implementation. Santa Clara: USENIX Association, 2024. 32.

[69] Kiefer F. evercrypt-rust. 2024. https://github.com/franziskuskiefer/evercrypt-rust

[70] Merigoux D, Kiefer F, Bhargavan K. Hacspec: Succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust. Inria: HAL, 2021.

[71] Hax. 2023. https://github.com/hacspec/hax

[72] Chen HB, Chen HH, Sun MS, Li K, Chen ZF, Wang XF. A verified confidential computing as a service framework for privacy preservation. In: Proc. of the 32nd USENIX Conf. on Security Symp. Anaheim: USENIX Association, 2023. 265.

[73] Njor EJ, Gústafsson H. Static taint analysis in Rust [MS. Thesis]. Aalborg: Aalborg University, 2021.

[74] Asterinas. 2024. https://github.com/asterinas/asterinas

[75] Sun XD, Ma WJ, Gu JT, Ma ZC, Chajed T, Howell J, Lattuada A, Padon O, Suresh L, Szekeres A, Xu TY. Anvil: Verifying liveness of cluster management controllers. In: Proc. of the 18th USENIX Conf. on Operating Systems Design and Implementation. Santa Clara: USENIX Association, 2024. 35.

[76] Ferrocene. 2024. https://ferrocene.dev/en/

[77] A-mir-formality. 2022. https://github.com/rust-lang/a-mir-formality

[78] Minirust. 2022. https://github.com/minirust/minirust

[79] Li HY, Guo LW, Yang YX, Wang SG, Xu MW. An empirical study of Rust-for-Linux: The success, dissatisfaction, and compromise. In: Proc. of the 2024 Conf. on Usenix Annual Technical Conf. Santa Clara: USENIX Association, 2024. 27.

[80] Wang YT. Compiler validation methods for Rust core language mechanisms. 2024 (in Chinese). https://jhc.sjtu.edu.cn/~yutingwang/files/posters/rust-hyl.pdf

[81] Zheng XY, Wan ZY, Zhang Y, Chang R, Lo D. A closer look at the security risks in the Rust ecosystem. ACM Trans. on Software Engineering and Methodology, 2024, 33(2): 34. [doi: 10.1145/3624738]

[82] Zhao JZ, Nagarakatte S, Martin MMK, Zdancewic S. Formalizing the LLVM intermediate representation for verified program transformations. In: Proc. of the 39th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Philadelphia: ACM, 2012. 427–440. [doi: 10.1145/2103656.2103709]

[83] Blazy S, Leroy X. Mechanized semantics for the Clight subset of the C language. Journal of Automated Reasoning, 2009, 43(3): 263–288. [doi: 10.1007/s10817-009-9148-3]

[84] Song PY, Yang KY, Anandkumar A. Towards large language models as copilots for theorem proving in lean. arXiv:2404.12534, 2024.

[85] Wu YH, Jiang AQ, Li WD, Rabe MN, Staats C, Jamnik M, Szegedy C. Autoformalization with large language models. In: Proc. of the 36th Int'l Conf. on Neural Information Processing Systems. New Orleans, 2022. 32353–32368.

[86] Jiang AQ, Welleck S, Zhou JP, Li WD, Liu JC, Jamnik M, Lacroix T, Wu YH, Lample G. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. arXiv:2210.12283, 2022.

[87] Sunday E. Using Kani to write and validate Rust code with ChatGPT. 2023. https://blog.logrocket.com/using-kani-write-validate-rust-code-chatgpt

[88] Sun CY, Sheng Y, Padon O, Barrett C. Clover: Closed-loop verifiable code generation. In: Proc. of the 1st Int'l Symp. on AI Verification. Montreal: Springer, 2024. 134–155.

附中文参考文献:

[11] 胡霜, 华保健, 欧阳婉容, 樊淇梁. Rust 语言安全研究综述. 信息安全学报, 2023, 8(6): 64–83. [doi: 10.19363/J.cnki.cn10-1380/tn.

2023.11.06]
[80] 汪宇霆. Rust 核心语言机制的编译验证方法. 2024. https://jhc.sjtu.edu.cn/~yutingwang/files/posters/rust-hyl.pdf

张卓若(1998－), 女, 博士生, CCF 学生会员, 主要研究领域为形式化方法, 程序验证, Rust 验证.

杨申毅(2001－), 男, 硕士生, CCF 学生会员, 主要研究领域为程序分析, 可信执行环境.

常瑞(1981－), 女, 博士, 副教授, 博士生导师, CCF 杰出会员, 主要研究领域为硬件辅助安全, 形式化验证, 程序分析.

陈芳(1999－), 女, 硕士生, 主要研究领域为形式化方法, Rust 验证.