arXiv:2602.07455v1 [cs.PL] 7 Feb 2026

RustCompCert：一个针对 Rust 顺序子集的经过验证且支持验证式编译的编译器

Jinhua Wu*、Yuting Wang*、Liukun Yu*、Linglong Meng§上海交通大学，中国* 明尼苏达大学，美国§yuting.wang@sjtu.edu.cn

1 引言

我们的目标是开发一个端到端的经过验证的 Rust 编译器。它应提供两项保证：一是语义保持，即源代码的行为包含目标代码的行为，从而可以将源代码中验证的性质保持到目标代码；二是由验证式编译确保的内存安全 [6]——借用检查过程 [20], 能够简化 Rust 程序的验证，例如允许验证工具专注于功能正确性 [2–5, 10, 13]。

然而，验证 Rust 编译器面临着诸多重大挑战。一些 Rust 编译和检查过程，例如析构展开 [15] 和借用检查 [20],, 涉及复杂的程序分析算法。在现有的经过验证的编译器框架中，对这些分析进行形式化和验证并非易事。现有的 Rust 形式化研究 [7, 22] 主要关注验证其类型健全性，而不是验证分析算法本身。Aeneas [4, 5] 提出了一种基于以借用为中心的语义的颇具前景的借用检查算法，但我们关注的是另一条路径——验证标准 Rust 编译器所使用的算法，即非词法生命周期 [1] 或 Polonius [19]。



图 1. RustCompCert 的结构（绿色：已验证）

我们介绍正在进行的工作：在 Rocq 中验证 Rust 编译器前端，并将其与 CompCert 连接起来 [11, 12]，以构建端到端验证的 Rust 编译器，我们称之为 RustCompCert。其整体结构如图 1 所示，由三个组件组成：(1) 未经验证的解析器，将 Rust 源代码转换为 Rustlight——Rust 的一个顺序子集，已在 Rocq 中形式化；(2) 经过验证的前端，将 Rustlight 编译为 Clight，其中包含 Rust 特有的编译阶段，例如析构展开和借用检查；(3) 基于 CompCert 的经过验证的后端。CompCert 是一种优化编译器，可将 Clight 编译为汇编[1]。

验证 Rust 的所有特性并不现实。因此，我们聚焦于该语言的一个核心子集，并将其形式化为 Rustlight。Rustlight 支持代数数据类型（即结构体和枚举），并通过 RAII 和所有权语义实现 Box 的自动内存管理；支持带有区域（或生命周期）注解的（可变／不可变）引用，以及安全模块与不安全模块的组合（例如，safeRust 和 C）。主要不支持的特性包括并发、多态、特征和高阶函数（例如闭包）。

我们的介绍安排。 我们计划用 20–25 分钟介绍这项工作，并预留 5–10 分钟进行问答。演讲大体上将遵循本文的结构。首先，我们将介绍 RustCompCert 的编译阶段及其主要正确性定理，见 §2。随后，我们将重点介绍如何按照 §3 所述，对借用检查算法进行形式化和验证。受时间和篇幅限制，我们可能不会介绍支持不安全代码的方法；感兴趣的读者可参阅本文 [23]。我们正在进行的形式化工作已公开发布于 https://github.com/SJTU-PLV/CompCert/tree/rust-verified-compiler。

2 RustCompCert 概述

RustCompCert 前端的内部结构如图 2所示。我们遵循标准 Rust 编译器（即 `rustc`）的设计来构建编译流水线。Rustlight 的语法与表层 Rust 相近。我们设计了一种名为 *RustIR*的中间表示，其语法与 MIR 类似。第一遍将 Rustlight 降级为 RustIR，为后续分析做准备，对应于`rustc` [16]中的 MIR 构造。随后，我们执行析构展开 [15], ；这是 `rustc` 中的关键处理过程，用于实现自动资源管理机制。此过程通过分析变量的析构 操作（即析构函数）应在何时执行，来确定具体需要执行哪些操作。该过程的正确性因此极大地依赖于程序分析的可靠性。

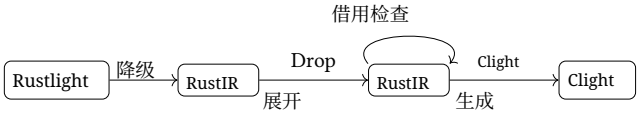

图 2。RustCompCert 前端的结构

随后，我们对 RustIR 执行借用检查。借用检查器由两个组件组成：移动检查过程 [17], , 用于确保每个变量在被移动后不再被使用；以及基于 Polonius [19] 的借用检查过程。相比于非词法生命周期 [1], , 即 rustc 中稳定的借用检查器，它将区域解释为“程序点集合

[1]我们采用 CompCertO 框架 [9, 24], , 这是 CompCert 的扩展，用于支持经过验证的组合式编译。

即某次借用可能存活于其中”，Polonius 可被视为一种别名分析，更适合在 CompCert 提供的程序分析框架中实现。它将区域解释为“引用可能指向的存储位置集合”，这与程序语义中引用可能存储的值自然契合。Clight 生成遍充当代码生成阶段。例如，它将枚举类型翻译为标记联合体，并为需要析构函数的类型生成 drop glue 函数。

编译器正确性。 与 CompCert 为 Clight 程序提供端到端的语义保持一样，RustCom-pCert 也为 Rustlight 程序提供这一保证：

$$\forall\ (M_s:\ \text{Rustlight})\ (M_t:\ \text{汇编}),$$

$$\texttt{RustCompCert}(M_s) = M_t \Rightarrow [\![M_t]\!] \leqslant [\![M_s]\!]$$

这里，$[\![M_t]\!] \leqslant [\![M_s]\!]$ 被定义为标准的反向模拟，这意味着源程序的行为包含于目标程序的行为之中。

借用检查的可靠性。 许多 Rust 验证工具都利用借用检查所确保的内存安全 [2–5, 10, 13] 来简化源级验证，例如专注于其函数正确性。为了提供这一保证，我们提取了一个名为 *RustIRspec* 的函数式规约，它将 RustIR 语义中的低层 CompCert 内存抽象为结构化内存模型。随后，借用检查的可靠性被形式化为 RustIRspec 与 RustIR 之间的精化：

$$\forall\ (M : \text{RustIR}),\ \text{借用检查}(M)\ \checkmark \Rightarrow$$

$$[\![M]\!]_{\text{RustIR}} \leqslant [\![M]\!]_{\text{RustIRspec}} \wedge \texttt{safe}\ [\![M]\!]_{\text{RustIRspec}}$$

对于通过借用检查的程序，我们可以得出结论：其规范是安全的，即不存在未定义行为。[2] 谓词 safe 可视为部分正确性的定义 [23]。当程序是开放的时，它将指定前置条件和后置条件，以支持组合。

端到端验证。 通过结合借用检查的可靠性和编译器正确性，RustIRspec 中验证的属性可以保留到汇编：

$$\forall\ (M_s : \text{RustIR})\ (M_t : \text{汇编}),\ \text{RustCompCert}(M_s)\ = M_t \Rightarrow \text{安全}\ [\![M_s]\!]_{\text{RustIRspec}} \Rightarrow \text{安全}\ [\![M_t]\!]_{\text{Asm}}$$

3 借用检查算法的形式化与验证

***Polonius* 的形式化。** 一种可行的策略是在 Rocq 中重新实现当前的 Polonius，例如将其代码从 rustc 移植过来。然而，由于它依赖于复杂的图构建和求解算法 [18],这种方法在实践中难以在 Rocq 中验证。我们转而在 CompCert 的 Kildall 框架 [8]中实现它，方法是定义其抽象域和传递函数。随后，通过 Kildall 求解器计算出的不动点即可得到 Polonius 的结果。

抽象域。 朴素定义是一个从区域到借用集合的借用映射（该术语表示被借用的存储位置）。然而，借用检查依赖变型规则 [21] 来维护区域之间的相等关系。因此，分析不仅必须在区域和程序点之间传播借用，还必须传播区域之间的相等关系。如果两个区域被记录为相等，它们必须包含相同的借用集合。我们通过在抽象域中使用并查集结构来维护区域相等性，从而实现这一点。要访问与某个区域相关联的借用，我们使用其在并查集中的代表元来索引借用映射，而不是直接使用该区域本身。因此，抽象域是类型为（$M \times U$）的二元组，由从区域到借用的映射和并查集结构组成。为了将该域集成到 Kildall 框架中，我们进一步证明它构成一个半格。

转移函数。 转移函数 F：$\mathbb{N} \to (M \times U) \to (M \times U)$ 描述类型为（$M \times U$）的抽象状态如何随节点 $n \in \mathbb{N}$处的指令演化。它主要规定借用如何从一个区域（例如赋值方）流向另一个区域（例如被赋值方）。此外，区域之间可能建立新的相等关系（例如通过变型规则 [21]）。相反，当关系中的某个区域变为死亡（即不再被使用）时，某些关系可能需要被移除。我们依据 NLL RFC [1] 中的子类型与再借用规则及其在 rustc 中的实现来设计该转移函数。

对 *Polonius* 的验证。 回顾一下，我们将借用检查的可靠性定义为 RustIRspec 相对于 RustIR 的精化。要证明任何通过借用检查的程序都满足这种精化，关键步骤包括：（1）建立借用检查分析所产生的抽象状态与语义中的具体状态（即内存模型）之间的近似关系；（2）证明在该关系下，RustIRspec 结构化内存模型中的每个操作，都能对应于 RustIR 的 CompCert 内存模型中一个不会陷入停滞的操作。特别是，这意味着不存在内存错误。第一步更具挑战性。将 Polonius 视为一种别名分析后，我们可以把每个区域与可能和包含该区域的存储位置发生别名的内存位置集合关联起来；这一性质更容易形式化 [14]。这种抽象关系也为提升 Polonius 算法的精度和性能提供了指导。例如，我们可以放宽别名集合以牺牲精度换取更好的性能（例如，让 Polonius 采用字段不敏感的方式），同时仍然能够使用相同的证明技术证明其可靠性。

[2]事实上，我们无法证明借用检查器能够排除所有未定义行为。例如，除零是借用检查无法阻止的典型未定义行为。因此，借用检查器所确保的安全性是部分安全性：语义中仍可能出现某些未定义行为。要实现完全安全性（不存在任何未定义行为），仍有赖于证明者排除这些未定义行为。

引用

[1] Aaron Turon、Konrad Borowski、Hidehito Yabuuchi 和 Dan Aloni。2017年。非词法生命周期。https://rust-lang.github.io/rfcs/2094-nll.html 访问日期：2025年10月12日。[2] Vytautas Astrauskas、Peter Müller、Federico Poli 和 Alexander J. Summers。2019年。利用 Rust 类型进行模块化规约与验证。《ACM 程序设计语言汇刊》3，OOPSLA，文章 147（10月 2019 30 10 1145 3360573），页。doi：。/ Xavier [3] Denis、Jacques-Henri Jourdan 和 Claude Marché。2022年。Creusot：一个用于 Rust 程序演绎验证的工具平台。载于《形式化方法与软件工程：第 23 届形式化工程方法国际会议，ICFEM 2022》，西班牙马德里，2022 年 10月24日至27日，会议论文集（西班牙马德里）。施普林格出版社，柏林，海德堡， 90 105 10 1007 978 3 031 17244 1_6 –。doi：。/ - - - - Son Ho [4] 、 Aymeric Fromherz 和 Jonathan Protzenko。2024 年。通过符号语义对 Rust 进行可靠的借用检查。《ACM 程序设计语言汇刊》8，ICFP，文章 251（2024年8月），29页。doi:10.1145/ 3674640[5] Son Ho 和 Jonathan Protzenko。2022年。Aeneas：通过函数式转换验证 Rust。《ACM 程序设计语言汇刊》6，ICFP，文章 116（ 2022 31 10 1145 3547647 年8月），页。doi：。/ Tony [6] Hoare。. 验证编译器：计算研究的一项重大挑战。《ACM 杂志》50，1（ 2003 2003年1月），63–69。doi:10.1145/ 602382.602403[7] Ralf Jung、Jacques-Henri Jourdan、Robbert Krebbers 和 Derek Dreyer。2017年。RustBelt：保障 Rust 编程语言的基础。《ACM 程序设计语言汇刊》2，POPL，文章 66（2017年12月）， 34 10 1145 3158154 页。doi：。/ Gary A. Kildall。. 一种统一的全局程序优化方法。载于《第一届 ACM [8] SIGACT-SIGPLAN 编程语言原理年度研讨会论文集》（马萨诸塞州波士顿）（POPL '73）。计算机协会，纽约，NY，美国，194–206。 1973 doi:10.1145/512927.512945[9] Jérémie Koenig 和 Zhong Shao。2021年。CompCertO：编译经认证的开放式 C 组件。载于《2021年 ACM 编程语言设计与实现会议论文集》（PLDI'21）。ACM，纽约，NY，美国，1095–1109。doi:10.1145/3453483.3454097[10] Andrea Lattuada、Travis Hance、Chanhee Cho、Matthias Brun、Isitha Subasinghe、Yi Zhou、Jon Howell、Bryan Parno 和 Chris Hawblitzel。2023年。Verus：使用线性幽灵类型验证 Rust 程序。《ACM 程序设计语言汇刊》7，OPSLA1，文章 85（2023年4月），30页。doi:10.1145/3586037[11] Xavier Leroy。2009年。现实编译器的形式化验证。《ACM 通讯》 52, 7 2009 107 115 10 1145 1538788 1538814 （），–。 [12] 2009 doi：。/。Xavier Leroy。. 一个经过形式化验证的编译器后端。《自动推理杂志》43，4（2009年），363–446。doi:10.1007/s10817-009- 9155-4[13] Yusuke Matsushita、Takeshi Tsukada 和 Naoki Kobayashi。2021年。RustHorn：基于约束霍恩子句的 Rust 程序验证。《ACM 编程语言系统汇刊》43，4，文章 15（2021年10月），54页。doi:10.1145/ 3462205[14] Valentin Robert 和 Xavier Leroy。2012年。一种经过形式化验证的别名分析。载于《经认证的程序与证明》，Chris Hawblitzel 和 Dale Miller（编）。施普林格出版社，柏林，海德堡，11–26。[15] Rust 编译器团队。2025年。析构展开。https://rustc-dev-guide.rust-lang.org/mir/drop-elaboration.html [16] Rust 编译器团队。2025年。MIR 构造。https://rustc-dev-guide.rust-lang.org/mir/construction.html [17] Rust 编译器团队。2025年。跟踪移动和初始化。https://rustc-dev-guide.rust-lang.org/borrow_check/moves_ and initialization.html [18] Rust 贡献者。2026年。Rust 编译器中 Polonius 的实现。https://github.com/rust-lang/rust/tree/main/compiler/rustc_borrowck/src/polonius 访问日期：2026年2月2日。

[19] Rémy Rakic 和 Niko Matsakis 代表 Polonius 工作组。2023年。Polonius 更新。https://blog.rust-lang.org/inside-rust/2023/10/06/polonius-update.html 访问日期：2025年10月12日。[20] Rust 团队。2025年。MIR 借用检查。https://rustc-dev-guide.rust- lang.org/borrow_check.html[21] Rust 团队。2025年。子类型与变型。https://doc.rust-lang.org/nomicon/subtyping.html[22] Aaron Weiss、Olek Gierczak、Daniel Patterson 和 Amal Ahmed。2019年。Oxide：Rust 的本质。*arXiv* 预印本 *arXiv:1903.00982*（2019年）。[23] Jinhua Wu、Yuting Wang、Liukun Yu 和 Linglong Meng。2025年。通过已验证与验证式编译实现程序安全的端到端组合验证。arXiv：2510.10015 [cs.PL] https://arxiv.org/abs/2510.10015[24] Ling Zhang、Yuting Wang、Jinhua Wu、Jéré mie Koenig 和 Zhong Shao。2024年。通过开放模块间的直接精化实现完全可组合且充分的已验证编译。《*ACM* 程序设计语言汇刊》8，POPL，第72号文章（2024年1月），共31页。doi：10.1145/3632914