arXiv:2601.19207v1 [cs.PL] 27 Jan 2026

Rust 中的重构与等价性

以新颖方法扩展 REM 工具链，实现自动化等价性证明

Matthew Britton 计算机学院，澳大利亚国立大学matthew.britton@anu.edu.auSasha Pak 计算机学院，澳大利亚国立大学sasha.pak@anu.edu.au

Alex Potanin 计算机学院，澳大利亚国立大学
alex.potanin@anu.edu.au

摘要

重构工具是现代软件开发的核心，提取函数重构在日常工作中应用广泛。然而，对于 Rust 而言，所有权、借用和高级类型系统特征相结合，使自动化提取函数重构尤其具有挑战性。现有工具，包括最初的 Rusty Extraction Maestro (REM) 原型，要么依赖缓慢的基于编译器的分析，要么仅支持语言的有限子集，要么除了“代码仍能编译”之外几乎不提供任何保证。

本文介绍了 REM2.0，这是一个面向 Rust 的新型提取函数与验证工具链。REM2.0 以 rust-analyzer 为基础，作为持久化守护进程运行，复用其分析基础设施，通过 VSCode 前端提供低延迟的提取函数重构。在这一提取引擎之上，它增加了一个修复器：当提取操作暴露出借用检查器问题时，自动调整生命周期和函数签名；同时还提供可选的验证流水线，可连接 CHARON 和 AENEAS，为受支持的 Rust 子集生成原函数与提取出的函数之间等价性的 Coq 证明。

该架构在三个基准测试套件上进行了评估。首先，在原始 REM 工件（包含从 gitoxide 等实际项目中选取的 40 个案例）上，REM2.0 实现了 100% 的兼容性：它涵盖了原型的所有原有成功案例，并修复了此前的若干失败案例。它还将提取延迟从平均约一千毫秒缩短至守护进程内部的个位数毫秒，而用户在编辑器中的感知延迟则远低于四分之一秒。其次，在一个新语料库上，该语料库包含从 20 个星标数很高的 GitHub 代码仓库中选取的 40 个侧重特性的提取案例。REM2.0 成功处理了大多数涉及异步/等待、常量函数、非局部控制流、泛型和高阶 trait 约束的示例；剩余的失败主要集中在动态 trait 对象和复杂泛型约束上。第三，在一组 20 个验证基准上，对于属于当前子集的案例，CHARON/AENEAS 流水线能够构建端到端的等价性证明。与提取相比，这一过程的性能开销显著更高，但也提供了相应更强的保证。

总体而言，结果表明，基于 Rust Analyzer 的设计能够为实际 Rust 程序提供快速且功能丰富的提取函数重构，而可选择启用的验证流水线则能为安全关键型重构提供经机器检查的行为保持保证。因此，REM2.0 为在 Rust 生态系统中实现常规验证式重构提供了一条切实可行的路径。

1 引言

Rust 是一种现代系统编程语言，兼具底层控制能力与强大的安全保障。 [1]这与以往的常见做法大不相同：过去，程序员必须在底层控制能力（使用 C/C 等语言）++和内存安全（使用 Java 等垃圾回收语言）之间做出选择。然而，这些安全保障使自动重构——尤其是“提取方法”——比在垃圾回收语言中复杂得多。此前的工作，例如$Adventure\ of\ a\ Lifetime$[2] 和 Rusty Extraction Maestro（REM）原型[3] 证明了在 Rust 中实现自动提取是可行的，但仅限于该语言的狭窄子集，且严重依赖不稳定的工具。

本项目从基本原理出发重新审视这一问题，将 REM 重新设计为独立的、由 Rust-Analyzer（RA）驱动的工具链，能够处理现实世界中的 Rust 特性，例如异步/等待、泛型、特征对象和常量上下文，同时将执行时间缩短至原来的五分之一。我们还引入了一条可选的验证流水线，用于检查提取结果是否保留了原有行为。总体而言，这些贡献旨在让安全且保留语义的提取方法重构能够用于日常 Rust 开发，而不再仅仅停留于研究原型阶段。

1.1 背景与动机

现代软件很少从零开始编写。相反，工程师会花大量时间扩展、调整和修复现有代码库。随着系统不断发展，crate 承担的职责越来越多，函数变得越来越长，依赖项也愈加复杂。如果不有意识地管理这种复杂性，技术债务就会迅速累积，未来的变更也会变得越来越昂贵。重构是在不改变程序可观察行为的前提下重组其内部机制的工作，是开发者用来保持代码库可维护性的主要工具之一。

经验证据表明，重构并非偶尔进行的清理工作，而是开发实践中的日常环节。JetBrains 对 1,183 名开发者开展的一项大规模调查发现，大多数受访者至少每周进行一次重构，许多人表示单次重构会花费一小时或更长时间 [4]。这表明，重构既十分普遍，手动执行时也可能耗时不少；工具支持对于定期实施小规模、安全的转换起着重要作用。

在各种重构操作中，提取方法 是最常见的操作之一。给定一段连续的代码（例如循环的函数体），提取方法会将这段代码移入一个新函数，并用对该函数的调用替换原始代码。通过为一段逻辑赋予描述性名称并缩短冗长的

函数，这样做能够提高可读性；同时，通过隔离代码的行为，我们还可以实现代码复用和有针对性的单元测试。在 Java 或 C# 等垃圾回收语言中，提取方法相对简单：工具会识别所选代码片段中的自由变量，将它们转化为参数或返回值，并依靠运行时管理对象生命周期和内存。

然而，Rust 也带来了额外的挑战。作为一种现代系统编程语言，Rust 在提供类似 C 语言的内存和性能控制能力的同时，还通过所有权与借用模型保证内存安全 [1]。Rust 不依赖垃圾回收，而是由编译器强制执行静态规则：每个值都有唯一的所有者；借用在数量和种类上受到限制；并且会跟踪生命周期，以确保引用的生命周期不会超过其指向数据的生命周期。这消除了许多常见类别的错误，例如释放后使用和许多数据竞争，但也使控制流结构、数据所有权和别名模式紧密耦合。

因此，直接将垃圾回收语言中的提取方法算法用于 Rust 并不可行。将代码移入新函数，可能会改变值所属的作用域，也可能改变同时作用于该值的可变引用和不可变引用的数量。某种转换在语法层面看似无害，却可能导致借用检查器拒绝该程序，或者改变程序中哪些部分可以修改共享状态。即使行为保持不变，转换后的代码也可能需要显式的生命周期参数或更复杂的类型签名，从而增加开发者阅读和维护代码的难度。

Rust 日益广泛地应用于对性能和安全至关重要的基础设施中。在这些领域，开发者既希望获得重构带来的可维护性优势，也希望保有促使他们选择 Rust 的安全保障。如果重构工具频繁导致代码无法编译、生成难以阅读的代码，或不支持 异步/等待、闭包和泛型等常见语言特性，开发者就不太可能依赖这些工具，而可能转而手动编辑，或干脆忽视问

以往研究表明，在 Rust 中进行非平凡的提取方法重构是可行的。具体而言，*Adventure of a Lifetime*（AoL）及其配套的 Rusty Extraction Maestro（REM）原型引入了一种针对 Rust 的重构流程，能够明确处理所有权、借用和非局部控制流 [2]。然而，最初的 REM 实现与过时的、基于 IntelliJ IDEA 的 Rust 前端紧密耦合，并且严重依赖高度不稳定的 rustc 内部实现，因此最多只能做到在不同 rustc 版本之间行为有所变化。此外，除了非正式地保证通过测试之外，它并不保证语义保持。本项目旨在弥合现实世界 Rust 开发对稳健重构支持的明确需求，与当前缺乏全面、能够感知生命周期并集成至现代 Rust 工作流的工具之间的差距。

1.2 问题陈述

目前，Rust 中提取方法的集成开发环境支持在简单情况下尚可，但当代码涉及实际的借用模式、复杂泛型、宏或异步结构时，支持仍然有限。RA 和基于 IntelliJ 的插件等工具通常能够处理较小的局部

代码片段，但往往会拒绝执行更复杂的重构，或生成无法编译的代码。它们也不会尝试修复生命周期或所有权问题，也无法保证重构后行为保持不变。

来自*Adventure of a Lifetime* [2] 的 Rusty Extraction Maestro（REM）原型通过引入针对 Rust 的分析，尤其是自动化的生命周期和所有权修复循环，解决了其中一些不足。应用提取方法转换后，REM 会根据编译器反馈反复调整提取出的函数的签名，直到借用检查器接受为止。据我们所知，其他 Rust 重构工具均未提供这种形式的自动化生命周期修复。然而，REM 仍是一个研究原型：它依赖过时的 IntelliJ 前端和脆弱的`rustc`内部机制；由于反复调用`cargo check`，它会产生较大的`async`/`await`延迟；并且完全不支持高级语言特征，例如闭包、复杂泛型、宏以及结构体字段的部分移动。

REM 和集成开发环境原生重构都存在另一项局限：无法有力保证转换能够保持程序行为。REM 会执行复杂的结构性更改，包括修改函数边界和所有权模式，还可能引入辅助类型或显式生命周期参数。测试套件可以在一定程度上确保重构后的程序按预期运行，但测试必然不完整，而且只能表明存在错误，无法指出错误是在何处引入的。对于旨在处理非平凡 Rust 代码的重构工具，显然需要一种更新的设计，将 REM 先进的生命周期修复能力与现代工具链支持相结合，并采用轻量级验证流水线，以提供更有力的行为保持证据。

1.3 项目目标

本项目旨在将原有的 REM 原型转变为适用于现代 Rust 开发的实用工具。首要目标是将 REM 重新设计为独立的流水线，不再依赖过时的 IntelliJ 基础架构或脆弱的 `rustc` 内部机制，并扩展其提取方法功能，以便更好地处理现代 Rust 语言特性，例如 异步 / 等待、闭包、宏以及更复杂的泛型用法。

第二个目标是将我们的提取方法重构与等价性检查流程相结合，以便提供比单靠测试更有力的语义保持证据。具体而言，本项目旨在将 REM 与现有的面向验证的工具链（尤其是 CHARON 和 AENEAS）集成，使重构能够转换为适合通过逻辑推理来分析原始代码与重构后代码之间等价性的形式。

第三个目标是通过一种能够自然融入现有工作流程的界面，提供所有这些新功能。这包括适用于脚本编写和批量评估的命令行界面，以及一款 Visual Studio Code 扩展，让开发者能够直接在编辑器中调用 REM。更长远的设想是探索与 RA 本身的集成，例如更直接地复用其中间表示，或将 REM 的部分分析功能嵌入语言服务器；但这超出了本项目的范围。

1.4 主要贡献

为实现这些目标，本项目作出了以下贡献：

面向现代工具链重新设计的 REM core。 原始 REM 实现已重构为独立的提取方法流水线，将分析和转换逻辑与过时的 IntelliJ 前端及脆弱的 rustc 内部实现解耦。新的基于守护进程的方案可轻松兼容现有集成开发环境和工作流程。

与 Rust Analyzer 集成，实现快速的内存内分析。 该项目引入了一种新的提取引擎，基于 RA 的语义信息和内存内工作区模型构建。这使 REM2.0 对代码的语义理解远胜于以往基于抽象语法树的方法。

扩展提取能力，以支持现代 Rust 特性。 提取方法流水线已扩展，可支持广泛的真实场景 Rust 结构，包括对异步/等待、泛型、非局部控制流、常量求值和高阶 trait 约束的特定用法。

通过 rem-server 和 VSCode 实现编辑器/集成开发环境集成。 核心实现以 rem-server 的形式提供，这是一个适用于与任意集成开发环境集成的长时间运行 JSON-RPC 进程。轻量级命令行界面和 VSCode 扩展原型展示了当前的实际用法。[1]

基于 CHARON 和 AENEAS 的原型等价性检查流程。REM2.0 利用程序验证领域的最新进展，将 Rust 转换为面向证明的函数式语言（如 Coq），从而能够自动检查提取方法重构的行为等价性。

在众多真实 Rust 项目上进行实证评估。 我们在一系列广泛使用的 Rust 项目上评估 REM 流水线，测量其覆盖率、性能和失效模式。这表明，新架构在显著提升实际适用性和语言覆盖范围的同时，保留了 REM 的优势，尤其是自动化生命周期和所有权修复能力。

1.5 方法概述与作用域

总体而言，本项目为 Rust 构建了一个多阶段的提取方法流水线，并在真实代码库上对其进行评估。核心提取和分析阶段复用了 RA 的内存工作区和语义信息，以识别候选代码片段并生成初步重构。随后，专用修复器会处理提取出的函数，修复生命周期和所有权模式，直到转换后的代码成为有效的 Rust 代码。对于选定的部分示例，原始程序和重构后的程序会通过 CHARON 和 AENEAS 转换为面向验证的中间表示，从而能够对语义保持进行推理。最终的工具链通过 JSON-RPC 服务器（rem-server）[2]，、轻量级命令行界面（rem-cli）[3]，和 Visual Studio Code 扩展提供，并在一组开源 Rust crate 上进行评估。本工作的范围仅限于提取方法重构；不考虑其他重构操作符。对宏和极其复杂的泛型模式的支持仍不完善，而且验证流水线仅应用于有代表性的案例研究，而非

[1] REM-VSCode 可在 VSCode Marketplace 上获取，
https://marketplace.visualstudio.com/items?itemName=MatthewBritton.remvscode[2]
https://crates.io/crates/rem-server[3]https://crates.io/crates/rem-command-line

整个代码库。将其全面集成到 RA 本身被视为一项长期方向，而非近期目标。

2 背景

Rust 的所有权与借用纪律，使我们无需垃圾回收器即可编写内存安全、无数据竞争的系统代码，但也提高了自动化重构和对程序行为进行形式化推理的门槛。 [1, 5]本节概述本报告其余部分所需的背景知识。我们首先回顾重构，尤其是提取方法转换。随后介绍 Rust 的所有权、借用、生命周期和控制流特征，并说明它们如何与提取方法交互。在此基础上，我们介绍 Rust 自动化重构方面的既有工作，重点关注 *Adventure of a Lifetime*以及 Rusty Extraction Maestro（REM）原型 [2, 3]。最后，我们概述 Rust 验证和形式语义方面的相关工作，并简要讨论现有工具及其与集成开发环境的集成。这些内容共同说明了本项目的贡献如何立足于当前最先进的工作，旨在让 Rust 代码更易于演进和推理。

2.1 重构概述

重构是通过重新组织现有代码来改善其内部结构，同时不改变其外部可观察行为的过程。它是大型软件项目中提升代码可维护性和控制技术债务的重要实践 [4]。现代集成开发环境（如 Eclipse 和 IntelliJ IDEA）提供了丰富的自动化重构功能，使开发者能够快速应用保持语义的变换，原则上也能安全地完成这些操作 [2, 6]。

如第 1.1 节所述，实证研究表明，重构是专业开发中的常规工作，但自动重构工具并未总能得到充分利用。开发者往往更倾向于手动修改，原因可能是他们不熟悉现有的重构功能，也可能是他们不确定工具能否保留原有行为。这促使人们研究更可靠（通过更强有力的保证）、操作过程也更透明的重构工具。

本报告聚焦于 提取方法 重构，前文第 1.1 节已对此作了介绍。在重构领域，提取方法通常被视为一种基础构件：将一段连续的代码移入新创建的函数中，并用对该函数的调用替换原始代码。这种方法可用于将冗长的方法拆分为更小的单元、隔离内聚的行为，也可作为提取类或移动方法等更复杂重构的起点 [6]。在 Java 或 C# 等垃圾回收语言中，大多数实现主要通过语法层面进行处理：识别所选代码片段中的自由变量，将它们转换为参数或返回值，并依赖运行时管理对象生命周期和内存。

清单 1 展示了这种转换风格在一种无需担心显式内存管理的语言中的应用。例如，一个包含验证和转换逻辑的长方法，可以通过……

```
// Before:
public void doComplicatedWork(List<Integer> nums) {
    int sum = 0;
    for (int n : nums) { // Consider this as a
        sum += n;        // Lengthy piece of
    }                    // Business logic
```

选择验证代码块，将其提取为专用方法，再用一次调用替换原代码块，从而进行重构。这样得到的代码更清晰，而且无需改变程序的行为，就能单独对验证逻辑进行测试。

```
    System.out.println("Sum is: " + sum);
}

// After:
public void doComplicatedWork(List<Integer> nums) {
    int sum = doSmallPartOfWork(nums); // Logic is now reusable
    System.out.println("Sum is: " + sum);
}

// This function can be unit tested on its own
private int doSmallPartOfWork(List<Integer> nums) {
    int sum = 0;
    for (int n : nums) {
        sum += n;
    }
    return sum;
}
```

图 1：Java 中提取方法重构示例

在 Rust 中，提取方法的高层意图相同，但这一变换与所有权、借用和生命周期的交互方式，在典型的垃圾回收语言中并不会出现。因此，在讨论 Rust 的自动化提取方法重构之前，了解这些语言特性至关重要。

2.2 Rust 的所有权模型与语义

Rust 通过静态的所有权与借用纪律来确保内存安全。每个值都有唯一的所有者；当所有者离开作用域时，该值就会被丢弃[4]。Rust 不依赖垃圾回收器，而是利用类型系统和编译时借用检查器来防止释放后使用、数据竞争以及其他常见的内存错误 [1, 5, 8]。其他语言也曾探索过类似的所有权纪律，尤其是面向 Java 的泛型所有权类型 [9, 10]。虽然这些保证对系统编程很有吸引力，但它们使手动和自动化的程序转换都变得更加复杂，其中包括提取方法。

[4]如果这是你第一次接触生命周期，我们建议阅读 Rust By Example 的这一章节 [7]，以便更好地理解生命周期！https://doc.rust-lang.org/rust-by-example/scope/lifetime.html

2.2.1 所有权与移动

Rust 模型的核心理念是每个值都有一个唯一的所有者。赋值或传递一个值通常会将所有权转移给新的变量或函数，使之前的绑定失效。右侧的列表 2展示了一个简单示例：

```rust
fn main() {
    let s1 = String::from("Hello");
    let s2 = s1; // Ownership of the string data moves to s2
    // The following line would be invalid if uncommented:
    // println!("{}", s1);
    // ^ value borrowed here after move
    println!("{}", s2); // Prints "Hello"
}
```

清单 2：Rust 中的所有权与移动

在此示例中，值从 s1 移动到 s2，因此之后 s1 将无法再使用。提取方法转换如果按值将 s1 传入新函数，可能会改变值的所有权所属作用域，或改变值被丢弃的时间。因此，自动重构工具必须考虑这些移动，以避免引入细微的所有权错误或导致编译失败。

2.2.2 借用

Rust 鼓励传递引用，而不是转移所有权。一个值可以被不可变借用（使用 `&T`）任意多次，也可以被可变借用（使用 `&mut T`），但同一时间只能借用一次。借用检查器会确保可变借用与不可变借用不会冲突，从而防止数据竞争以及许多类型的别名错误 [11]。清单 1 展示了基本模式。

```rust
fn print_message(message: &str) {
    println!("{}", message);
}

fn main() {
    let greeting = String::from("Hello, world!");
    print_message(&greeting); // Borrows greeting immutably
    println!("Still own greeting: {}", greeting);
}
```

（a）不可变借用：允许多个读取者

```rust
fn add_exclamation(message: &mut String) {
    message.push('!');
}

fn main() {
    let mut greeting = String::from("Hello");
    add_exclamation(&mut greeting); // Borrows greeting mutably
    println!("{}", greeting);       // Prints "Hello!"
}
```

(b) 可变借用：单一写入者

图 1：Rust 中的借用

将一段代码提取为新函数时，工具必须针对每个捕获变量决定是移动它、以不可变方式借用它，还是以可变方式借用它。选择不当可能导致借用检查器拒绝重构后的程序，或迫使工具引入不必要的克隆，从而改变性能特征。这是设计 Rust 提取方法算法时面临的核心挑战之一。

2.2.3 生命周期与推断

Rust 编译器使用生命周期参数跟踪引用保持有效的时长。许多简单情况不需要显式注解，因为编译器可以推断输入和输出生命周期之间的关系。例如，清单 2 中的函数可以不写显式生命周期参数：

```
函数回显(消息: &str) -> &str { 消息 }
```

清单 2：开发者如何编写 Rust 代码

从概念上讲，编译器会将其视为程序员编写了清单 3，为参数和结果引入一个显式的生命周期，a：

```
函数回显<'a>(消息: &'a 字符串) -> &'a 字符串 { 消息 }
```

清单 3：Rust 编译器“看到”的 Rust 代码

然而，将代码提取到新函数中时，引用之间的关系可能会变得更加复杂，编译器也可能无法再自动推断生命周期。在这种情况下，必须为提取出的函数显式指定生命周期参数。因此，自动重构工具必须能够在转换过程中引入、传播，并且最重要的是简化生命周期注解。

2.2.4 非局部控制流

Rust 允许在马环中使用 break 和 continue。将包含此类语句的代码片段提取到独立函数中，会改变控制流结构：原始函数中的提前 return 会变成从提取出的函数中 return，切实可行的 Rust 提取方法实现，都必须处理这类情况，或显式编码并重建非局部控制流。第 2.3.2 节讨论 REM 工具时将再次探讨这一点。

总的来说，Rust 的所有权、借用、生命周期和控制流结构提供了强有力的安全保证，但也使数据流、控制流和作用域紧密耦合。因此，即使是看似局部的重构（例如提取方法），也比在限制较少、采用垃圾回收的语言中复杂得多。

2.3 Rust 自动重构技术2.3.1 重命名与简单重构

G. Sam 等人（2017年）开展了最早一批构建 Rust 重构工具的工作之一，他们创建了一个利用 Rust 编译器内部 API 的概念验证重构框架。团队与 Mozilla Research[5] 合作，率先实现了 Rust 特有的重构功能：生命周期省略 和生命周期具体化。这使他们的程序能够在编译器原本会隐式推断生命周期的情况下，引入显式的生命周期参数——

[5]Mozilla Research 是 Rust 项目最早的主要赞助方，并在过去十年间一直是推动其发展的重要力量。 [12]。 https://blog.mozilla.org/en/mozilla/mozilla-welcomes-the-rust-foundation/

这项重构在 Rust 中前所未有。遇到的挑战表明，Rust 更严格的作用域规则和遮蔽规则，要求在重命名时谨慎处理名称冲突（例如，避免重命名变量时意外遮蔽另一个变量） [5]。此外，2017年的研究得出结论：Rust 编译器基础设施支持许多重构，但要确保行为保持 （尤其是所有权转移方面），则需要额外的静态分析，而不受 Rust 约束的语言并不需要这些分析。他们在生命周期省略 方面的工作后来成为 REM 工具链的一部分；该工具链利用 Rust 编译器提供的丰富反馈，确保最终转换既是有效的 Rust 代码，又尽可能易于阅读 [2], [3]。

2.3.2 提取方法（REM）

Rust 自动化重构的一项重大进展是 Rusty Extraction Maestro（REM）。Costea 等人的 *Adventure of a Lifetime: Extract Method Refactoring for Rust* [2] 提出了理论框架，而 Thy 等人的*Borrowing Without Sorrowing* 则以 IntelliJ 插件的形式提供了实际实现 [3]。REM 将 Rust 中的提取方法分解为一系列转换，每项转换都旨在处理 Rust 所有权、借用、生命周期和控制流中的某个特定方面。

该过程首先进行一次朴素的提升¨，将选中的代码块移入一个新函数，并用一次调用替换原来的代码片段。这种初始转换后的代码通常无法通过类型检查。接着，REM 会执行一系列自动修复，这些修复由称为预言机的专门分析提供指导。所有权分析会针对每个跨越新函数边界的值，确定应按值传递、按共享引用（`&T`）传递，还是按可变引用（`&mut T`）传递。其他处理步骤会将非局部控制流具体化：把提前返回、跳出和继续语句编码到一个[illegible]助枚举中，使调用方能够[illegible]果进行模式匹配来还原原有行为。最后的修复循环会反复调用 R[illegible]器[illegible]借用[illegible]错误消息作为反馈，并引入或调整生命周期参数，直到提取出的函数通过类型检查。

图 2 （改编自 [2]）展示了 REM 流水线的整体结构。每个阶段都刻意设计得很简单，但组合起来便能实现一种可处理（大多）真实 Rust 代码的转换。在对真实项目的评估中，REM 能够完成 IDE 工具无法处理的提取，包括涉及多个相互作用的借用和嵌套生命周期的情况 [2]。据我们所知，REM 自动化的生命周期与所有权修复循环在 Rust 重构工具中仍独树一帜。

尽管具备这些优势，最初的 REM 原型仍与过时的、基于 IntelliJ 的 Rust 前端以及不稳定的 `rustc`内部机制紧密耦合，正如第1节所述。这限制了它在现代工具链中的适用性。我们的项目以 REM 的概念设计为基础，尤其借鉴了其生命周期和所有权修复能力，同时围绕 Rust Analyzer 和独立的守护进程架构重新设计实现，使提取方法重构在当代 Rust 开发中切实可行。

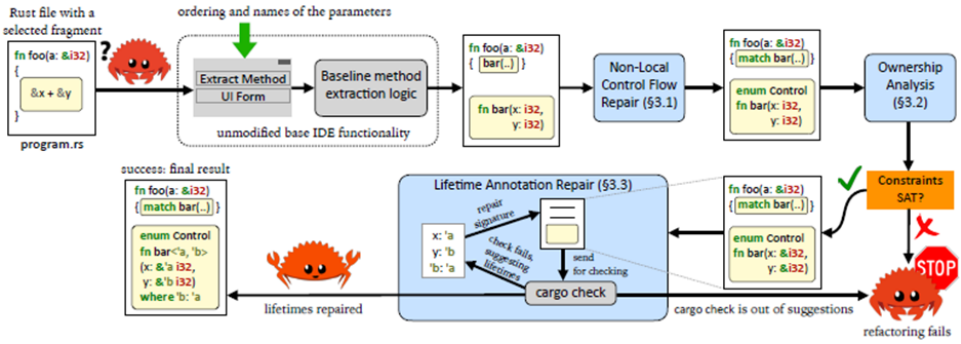

图 2：REM 提取方法流水线概览（改编自 Costea 等人 [2]）

2.3.3 所有权错误的自动修复

另一项与 Rust 重构相关的研究聚焦于自动修复违反所有权规则或借用规则的代码——将无法编译的程序转换为可编译的程序，同时不改变其语义。*Rust-Lancet* (Yang 等人，ICSE 2024) 正是针对这一问题，旨在修复违反所有权规则的情况，同时保持程序行为不变。对于无法编译的 Rust 文件，Rust-Lancet 使用 `syn` 和 `quote`分析其抽象语法树，应用有针对性的转换，并反复重新检查程序，直到错误消除。典型的修复包括插入 `clone()` 以满足所有权约束，或调整语句顺序以避免无效借用。尤为重要的是，Rust-Lancet 包含行为保持检查：它使用一种专门用于推理尚无法编译的程序 验证每项建议的修复都不会改变可观察程序输出。

在 160 个真实所有权违规案例中，Rust-Lancet 成功修复了绝大多数问题，且没有任何假阳性——从未引入错误修复，其表现优于编译器建议和多个基于大语言模型的基线。尽管它被定位为错误修复工具，而非重构引擎，但其保持语义的抽象语法树重写流水线与自动化重构有很大重合。其结果凸显了 Rust 严格的所有权模型如何指导自动化转换：一旦编译器接受修复后的代码，且行为保持检查通过，该工具就能确信补丁正确 [13]。

2.4 Rust 的验证与形式化方法

Rust 在构建可靠系统方面的吸引力，推动了大量关于形式语义和验证的活跃工作与研究。传统的演绎验证和模型检验必须进行调整，以考虑 Rust 的所有权、生命周期和不安全代码；但与 C 或 C++ 的传统方法相比，Rust 强大的静态保证也让许多验证任务更易处理++。

2.4.1 RustBelt 与类型系统可靠性

Jung 等人的 **RustBelt** 首次为 Rust 类型系统的一个实用子集给出了经机器验证的可靠性证明 [14]。RustBelt 将类 Rust 演算（ λRust ）嵌入 Iris 并发分离逻辑，并将其与低级内存模型相结合，由此证明，类型正确的安全 Rust 代码不会出现未定义行为，即使使用的库包含不安全内部实现也是如此。这一奠基性结果使后续研究能够假定安全 Rust 代码在内存方面和

线程方面都是安全的，并将验证工作集中于函数正确性，或仔细挑选的不安全代码片段。

2.4.2 通过函数式转换进行验证

另一种互补方法是将 Rust 程序转换为纯函数式语言，以便使用现有的证明助手和验证器。CHARON[6]/AENEAS[7] 工具链采用了这一策略。CHARON 将 Rust 转换为一种名为低级借用演算 (LLBC) 的中间表示。它与 Rust 的 MIR 相关，但会明确表示所有权和借用结构。随后，AENEAS 为 LLBC 赋予纯粹的、基于值的语义，并将其转换为适用于在 Coq 或 F* 等工具中进行验证的 λ 演算 [15, 16]。这种方法建立在使用证明助手编码语言语义的悠久传统之上 [17]。对于一大类 Rust 程序，这种转换无需直接推理指针和堆：证明工程师可以基于纯函数式模型开展工作，而 CHARON/AENEAS 的设计则保证，已证明的性质能够回溯到原始 Rust 代码。事实上，在本报告后文（第 4.4 节）中，我们将依靠这一保证来支撑验证流水线。

最近，Ho 等人提出了 Rust 的符号语义，并证明它们构成了 LLBC 程序具体基于堆执行的一种可靠抽象 [16]。一项重要成果是经过验证的符号解释器，可作为 LLBC 的经证明正确的借用检查器。这有力地保证了基于 CHARON/AENEAS 构建的工具拥有坚实的语义基础；更重要的是，它们在 Rust 非正式的所有权直觉与数学上精确的模型之间建立了桥梁。

2.4.3 静态分析器与模型检查器

除了完整的函数式验证之外，还有一些工具可为 Rust 提供较轻量级的保证。*Prusti* 会将带注释的 Rust 代码转换为 Viper 验证语言，并利用 Rust 自身的所有权和生命周期信息来编码权限，以及证明用户指定的前置条件、后置条件和不变量 [18, 19]。*RefinedRust* 进一步拓展了这一思路，为 Rust 引入精化类型系统，使丰富的安全性和正确性属性能够表达为类型，并在 Coq 中完成证明，即使是某些不安全代码也不例外 [20]。

在模型检测方面，*Kani* 是一种面向 Rust 的有界模型检查器，运行于 MIR 上，并为 SAT/SMT 求解器生成验证条件 [21]。通过在 MIR 层级工作，Kani 能够精确地推理特征、泛型和动态 trait 对象 [22]，，并已应用于密码学库等关键组件。像*Miri* 这样的动态分析器则通过在能够检测未定义行为的解释器中执行 Rust 程序来补充这些工具，即使不进行完整验证，也能在测试时提供有力的检查。

RustBelt、CHARON/AENEAS、Prusti、RefinedRust、Kani 和 Miri 共同勾勒出一个丰富的验证生态系统。它们表明，Rust 的所有权纪律不仅是一种约束，也是一种强大的抽象层，验证工具可以加以利用。

[6] https://github.com/AeneasVerif/charon/

[7] https://github.com/AeneasVerif/aeneas

2.5 与集成开发环境和语言服务器的集成

要让重构和验证工具在实践中得到采用，就必须顺畅地融入开发者的工作流程。对于 Rust，这方面的工作一直围绕编译器和语言服务器协议（LSP）展开。最初的 Rust 语言服务器（RLS）试图将`rustc`的内部机制改用于集成开发环境支持，但在性能以及跟上语言演进方面举步维艰。此后，社区逐渐转向*Rust-Analyzer* (RA)，这是一种维护自身语法树并执行增量类型推断的语言服务器，能够快速响应编辑器查询 [23]。RA 将重构和代码转换作为作用于其内部表示的“辅助操作”提供；其中已经包含许多常见的重构。

工业级集成开发环境也承载了 Rust 重构研究。最初的 REM 原型被实现为 IntelliJ Rust 插件的扩展，表明研究级重构可以基于现有编辑器基础设施进行原型开发 [2, 3]。在实践中，维护此类集成颇具挑战：集成开发环境 API 和 Rust 工具链都在快速演进。为缓解这一问题，一些工具会刻意避开 rustc 内部实现，转而依赖 syn/quote 等稳定 crate 来进行解析和代码生成，Lancet 的设计便是如此 [13]。

在编译器层面，*rustfix* 等项目展示了另一种集成方式：rustc 会针对某些错误和弃用项发出可由机器应用的建议，而 rustfix 可以在整个代码库中自动应用这些建议。虽然这并非通用的重构引擎，但这种机制让编译器能够直接驱动小规模、保持行为的编辑（例如更新已弃用语法或插入缺失的生命周期说明符）。随着 Rust 稳定分析 API 的日趋成熟，以及 RA 辅助操作框架不断扩展，外部工具（例如本报告所述的 rem-server 守护进程）有了越来越多的机会接入这些生态系统，提供更先进的语义感知重构，而不受限于单一集成开发环境。

2.6 Rust 之外的相关工作

Rust 重构中的许多挑战与机遇，都与其他语言领域的早期工作遥相呼应。Java 和 C# 长期以来都拥有成熟的集成开发环境支持，可用于 重命名、提取方法和提取类等重构，而这一生态系统也启发了大量研究。除泛型感知重构等经典示例外， [24], 近期研究更加关注如何系统地迁移到新的语言特性。举例来说，Zhang 等人的 *ReFuture* 工具通过结合静态分析（访问者模式、别名分析）与转换规则，自动将 Java 异步代码重构为使用现代 `CompletableFuture` 应用程序接口，并已在 Hadoop 和 ActiveMQ 等大型项目上进行评估 [25]。在 的 *ASYNCDROID* 会将 Android `AsyncTask` 的不当用法重构为更安全的结构，例如 `IntentService`，以解决常见的内存泄漏和用户界面错误问题；研究还表明，领域特定重构规则可以封装为集成开发环境插件 [26]。

重构也在其他编程范式和生态系统中得到研究。在 Haskell 中，*HaRe*工具将结构化重构引入了惰性纯函数式语言，并处理了类型类和非严格求值等挑战 [27]。在 JavaScript 和 Python 等动态语言中，由于缺乏静态类型，重构方法往往依赖启发式方法、运行时分析或开发者引导的转换。与此同时，Facebook 的 *JSCodeshift* 等工具

以及 Python 的 *Bowler* 允许开发者针对抽象语法树编写自定义代码修改脚本[28]。

最后，重构与代码异味检测、自动程序修复和大规模迁移等相关领域密切相连。一些修复系统会采用类似重构的转换，在应用修复前或后，通过保留行为的步骤简化有缺陷的代码；而语言服务器也日益成为众多编辑器实现这些功能的通用基础 [2]。本节综述的 Rust 相关工作正是这一广阔领域的一部分：它将成熟生态系统（Java、Android 和 Haskell）中的理念应用于一种基于所有权的语义需要专门分析的语言，同时也利用该语言强有力的保证，为重构和验证工具提供助力。

3 扩展 REM 的能力

长期以来，Rust 中的自动化重构一直受限于该语言严格的所有权、借用和生命周期规则。Rusty Extraction Maestro 的早期原型表明，提取方法 重构可以有效实现，但适用范围相对有限。它们不支持泛型类型和异步代码等复杂语言特性。此外，整个工具链依赖于与 IntelliJ 的脆弱集成，很快便过时了。因此，REM 始终停留在学术性的概念验证阶段，而未能成为开发者能够在实际中采用的工具。

本节详细介绍 REM 如何从一个脆弱的原型发展为稳健、模块化且可扩展的系统。如今，REM2.0 的核心采用精简的多组件流水线，旨在提升速度和灵活性，并与现代 Rust 工作流集成。

REM 服务器通过 JSON-RPC 接口统筹整个流程，协调提取、修复和等价性检查（后者将在第4节讨论）。提取工具直接构建于 Rust Analyzer 的增量分析引擎之上，可快速执行语义提取，同时整合 REM 原有控制器和借用者模块的功能。修复工具改编自原始 REM 工具链，在重构期间解决生命周期及其他可能出现的语义不一致问题方面，继续发挥着关键作用。

最后，一个 Visual Studio Code（VSCode）扩展原型连接到 REM 服务器，提供近乎即时的反馈——端到端通常约为 200 毫秒（见第5节）。这些组件共同将 REM 打造成一个实用的重构系统，能够融入日常 Rust 开发，同时保留其作为持续研究平台的基础。

3.1 动机：为什么 REM 需要改变

最初的 REM 原型证明了 Rust 中自动化提取方法重构的可行性，但其设计也暴露了若干根本性限制。REM 的核心依赖于反复调用 cargo check、一系列极不稳定的 rustc 内部实现，以及 IntelliJ IDEA Rust 插件；后者后来被 RustRover 取代，而且在本项目的生命周期内一直不太稳定。因此，工具链

速度缓慢、脆弱不堪，并且对 Rust 生态系统和底层编辑器集成中的细微变化都极其敏感。它作为研究原型以外的工具，实际上毫无实用可能：在开展任何新工作之前，光是将原始实现恢复到可用状态就花了一个多月。

这些架构选择也严重限制了 REM 能够处理的 Rust 程序类型。即使是小型的现代 Rust 代码片段，也经常会用到 REM 根本无法处理的语言特性。例如，REM 连最简单的异步函数都无法提取：

```
1 async fn fetch() { 2 let v = client.get("/").await; // REM 无法识别“await” 3 } 它也无法处理涉及泛型参数或约束的函数：1 fn min<T: Ord>(a: T, b: T) ->T { /* ... */ } 而且，它也难以处理非局部控制流，例如循环中的 break 或 return：1 for x in xs { 2 if x < 0{ break; } // REM 难以将其干净地提取到新函数中 3 }
```

如今在生产代码库中已司空见惯的语言特性，例如`async/await`、常量求值、带有复杂约束的泛型、特征对象、高阶 trait 约束以及非局部控制流[8]，都完全超出了 REM 能够可靠支持的语言子集。尽管原始代码库中有许多零散的待办事项提及未来将支持泛型或异步，但并没有一套连贯的计划来说明如何将这些特性整合到流水线中。实际上，这意味着 REM 有时能够成功处理某些大型项目，例如`gitoxide`，但在我们的测试中，面对 Deno、Tokio 或 Vaultwarden 等更复杂的代码库时，它经常会失败或超时；这些代码库中广泛使用了上述高级特性。

或许最重要的是，REM 没有提供正式的安全保障。它的提取决策主要由基于语法的抽象语法树重写驱动，而非对程序进行深入的语义理解；同时，也没有验证流水线来检查转换是否保留了行为。重构可能在程序仍能编译的情况下悄然改变程序语义，而该工具没有机制检测或防止此类回归。这些限制共同促使系统进行彻底重设计：若要让提取在日常 Rust 开发中发挥作用，REM 就必须变得更快、更稳健、功能更完整，并在语义上值得信赖。

3.2 高层愿景：REM2.0 需要实现什么

REM2.0 正是针对这些不足而设计的，并确立了一组明确的高层目标。首要目标是性能：原始 REM 在

[8]非局部控制流 *is* 受 REM 支持，但其实现依赖一次性的自定义泛型，而非现代`std::ops::ControlFlow`对象。

IntelliJ 完成初始提取大约需要一秒，这对于交互式重构工作流来说是不可接受的。要在集成开发环境中提供自然流畅的体验，提取必须在远低于一秒的时间内完成，理想情况下应控制在 500 ms 以内。为此，必须放弃“重新运行编译器”的模式，转而采用持久化语义引擎。与此同时，REM2.0 还需要突破其前身所支持的受限语言子集，基本支持任何实际的 Rust 代码。这意味着必须将异步函数、常量求值、泛型（包括复杂约束）、特征对象、高阶 trait 约束以及非局部控制流作为真正的语言特性来处理，而不能再将它们当作源代码树中的待办事项。

第二个同样重要的目标是易用性。最初的 REM 主要是一项研究成果，构建、配置和使用它都需要投入大量精力；现实中，极少有日常开发者能够从中受益。相比之下，REM2.0 旨在成为一款易于采用、几乎无需额外适配的工具：它应能顺畅融入常见的开发工作流，通过熟悉的 VSCode 界面提供功能，并且不与不稳定的编译器内部实现交互[9]。尽管界面简单直观，系统仍必须提供强有力的正确性保证。重构操作具有确定性，并基于一种稳定表示，以支持可靠的转换和验证。在正确性后端的协助下，该工具不仅确保安全性，也确保行为等价性。总之，REM 2.0 旨在实现快速、兼容现代 Rust，且绝不会悄无声息地破坏代码。

3.3 REM2.0 概览：核心贡献

在深入探讨架构和实现细节之前，先从整体上概述 REM2.0 的关键贡献很有帮助。以下各项变更／升级共同将 REM 从一个脆弱的研究原型转变为一套适用于现代 Rust 的实用且可扩展的重构工具链。

• **基于 Rust-Analyzer 的提取引擎**。 REM 2.0 不再直接依赖 rustc 内部实现和特定编辑器插件，而是采用一种以 Rust-Analyzer 为后端的新提取引擎。自定义的“单文件工作区”抽象使我们能够针对当前文件运行提取方法，同时充分利用 Rust-Analyzer 完整的语义分析能力。

• **全面覆盖提取所需的语言特性**。 借助 Rust-Analyzer 的能力，REM2.0 支持在存在异步函数、常量求值、泛型（包括复杂约束）、特征对象、高阶 trait 约束以及非局部控制流的情况下进行提取，而不再局限于 Rust 中狭窄且经过人工挑选的特性子集。

• **保留并扩展 REM 的生命周期修复能力**。 新系统保留了原版 REM 的所有核心功能，尤其是提取后修复生命周期注解及其位置的能力，同时将这些修复集成到一个实用性和易用性都大幅提升的流水线中。REM2.0 使开发者能够在正确性最大化、响应速度最大化或兼顾两者之间自行选择。

• **用于行为等价的验证后端**。REM2.0 引入了一条端到端的

[9]已安装程序可用的这些内部机制版本，完全取决于开发者所使用的编译器，因此，原始 REM 可能与某个代码库完全不兼容。

验证流水线，它通过 Charon 和 LLBC 将提取的代码转换为 AENEAS 和 Coq，从而自动证明（或证伪）原函数与重构后函数之间的等价性。完整说明请参见第 3.6.1 节和第 4 节。

- **用于交互式重构的 VSCode 集成**。全新的 VSCode 扩展将 REM 2.0 打造成一款交互式重构工具，提供提取方法命令、预览和撤销功能，并直接向开发者展示验证器结果。

这些贡献共同使 REM 2.0 成为一套快速、功能完备且具备语义感知能力的提取方法工具链，既适用于研究，也适用于日常 Rust 开发。

3.4 REM2.0 架构概述

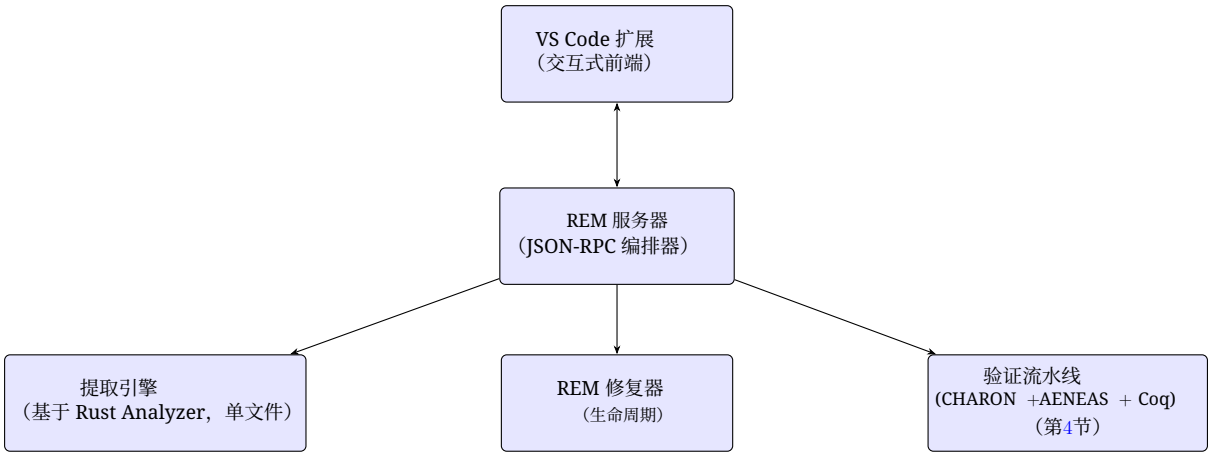

图3：扩展后的 REM架构高级概览：VS Code 与 REM 服务器进行双向通信，后者将请求分发至提取引擎、REM 修复器和 REM验证流程。

扩展后的 REM架构围绕模块化、面向服务的设计进行组织，能够将交互式编辑与语义分析及其外围逻辑解耦。图3 展示了系统在最高层级上的结构。

系统的核心是 REM 服务器，这是一个轻量级的协调器，通过 JSON-RPC（远程过程调用）接口向 VSCode 扩展提供服务。服务器还可以重新导出为命令行界面，以便集成到本地程序中 [10]。服务器负责协调所有主要操作——提取、修复和验证，处理错误，并即时反馈某项操作是否可行。重要的是，该接口完全通用，设计上可以非常轻松地适配任何其他集成开发环境。

提取引擎（REM提取器）构成了新工具链的分析核心。它直接构建于 Rust Analyzer 之上，其关键创新在于能够对单个源文件 而非整个 crate 执行语义提取。这种适配是 REM 速度的关键：Rust Analyzer 通常会分析整个项目以生成其高级中间表示（HIR），而我们的引擎能够利用内部特征和接口，在无需考虑更广泛的模块／crate 上下文的情况下，对单个文件进行分析。给定一个

[10] https://crates.io/crates/rem-command-line

提取请求（文件路径、选区范围、函数名），引擎会构建该文件的最小语义模型，确定选区的精确边界，并合成提取出的函数及相应的调用点编辑。引擎还能够根据调用模式和下游组件的需求，沿调用链向上返回提取出的语义信息。

提取完成后，生成的转换结果会（可选地）传递给 REM 修复器。该修复器直接继承自最初的 REM 原型，但经过重新设计，以提升性能和模块化程度。它的任务是根据 `cargo check` 的指导，补全并纠正所有缺失的生命周期，然后应用一系列生命周期省略规则，尝试将最终的混乱结果整理成（大致）易于人类阅读的形式。其核心算法仍与《Adventure of a Lifetime》中讨论的相同 [2], ，因此本报告省略了对该算法的大篇幅讨论。

最后，VSCode 扩展为系统提供了面向用户的界面层。它作为轻量级客户端，将提取请求序列化后发送至 REM 服务器，并将返回的编辑应用到当前打开的缓冲区。通过 JSON-RPC 进行双向通信，可将从用户操作到视觉反馈的往返时间控制在 200 毫秒以内。从用户的角度来看，REM 的表现就像集成开发环境的原生功能，而不是外部工具。

3.5 基于 Rust-Analyzer 的单文件工作区

REM2.0 能够快速运行的一个关键条件，是可以按需执行提取方法，而不必重新分析整个项目，甚至无需真正访问整个项目。然而，Rust-Analyzer（RA）的设计围绕 crate 级工作区展开：它要求先有 Cargo.toml、依赖关系图和完整的模块树，之后才会执行语义分析。为了弥合这一差异，REM2.0 会构建一个轻量的合成“单文件工作区”，将当前正在重构的文件视为一个独立的 crate。这个工作区仅包含 RA 解析、索引和类型检查该文件所需的最少元数据。

这种抽象让我们能够快速增量地分析语义，同时避免反复分析整个代码仓库的开销。然而，我们发现，单文件工作区并不了解标准库：默认可用的类型，例如 `Vec<T>`、`Option<T>`、`String`，甚至 `Result<T,E>`，都无法使用，除非用户出于某种原[illegible]与[illegible]。当提取[illegible]引用 std 提供的类型时，Rust-Analyser 会为这些类型指定占位类型（内部表示为“ ”或不完整推断变量）。

图 4清楚地凸显了这一限制有多么重要。提取清单 4a 应生成清单 4c中完整推断出的函数，但在单文件工作区（未加载 `std`/ `core`）中，RA 生成的却是清单 4b所示的不完整签名[illegible]个占位类型会传播到调用点和返回位置，生成的代码将无法编译。

```rust
fn main() {
//Rust-Analyzer(single-file) can't
//know what `Vec` is here.
//Even if `Vec` was strong typed
    let mut v = Vec::<i32>::new();
    for i in 0..10 {
        v.push(i);
    }

    println!("{:?}", v);
}
//
//
```

(a) 原始代码

```rust
fn main() {
    let v = build_vec();
    println!("{:?}", v);
}

//Rust-Analyzer's *generated*
//signature uses a placeholder
//as it can't find `Vec`
fn build_vec() -> _ {
    let mut v = Vec::<i32>::new();
    for i in 0..10 { v.push(i); }
    v
}
```

(b) 提取失败
由于 std 不在作用域内

```rust
fn main() {
    let v = build_vec();
    println!("{:?}", v);
}

//With std in scope,
//This return type can be
//Correctly inferred
fn build_vec() -> Vec<i32> {
    let mut v = Vec::<i32>::new();
    for i in 0..10 { v.push(i); }
    v
}
```

(c) 正确提取
现在 std 位于作用域内

图 4：比较 std 和 core 修复前后的提取结果。修复前，我们会返回一个“ ”作为占位符，因为 Rust-Analyzer 无法在缺少 std 提供的语义信息时推断类型。

为了解决这个问题，REM 2.0 会构建第二个持久化工作区，其中包含完整的标准 crate 集合及其依赖图。VSCode 扩展启动时会初始化此工作区（这是一次性但耗时的步骤），并假定它在各次重构期间保持稳定。提取过程中，REM 会合并 `std` 工作区与单文件工作区的语义图，从而生成一个“有效”工作区，同时完整掌握用户代码和标准库。有了这一视图，R能重新推断出 `Vec<T>`等类型，从而恢复清单 4c中完整推断出的提取结果。旁边展示了导入此持久化工作区的完整标准 crate 集合。

为解决这一问题，REM2.0 会构建第二个持久化工作区，其中包含完整的标准 crate 集合及其依赖关系图。该工作区在 VSCode 扩展启动时初始化一次（这是一次耗时较长的操作），并假定在重构过程中保持稳定。提取时，REM 会将 `std`工作区与单文件工作区的语义图合并，生成一个同时完整了解用户代码和标准库的“有效”工作区。有了这一视图，RA 便能重新推断诸如 `Vec<T>` 之类的类型，从而恢复 Listing 4c中展示的完全推断结果。旁边列出了导入此持久化工作区的完整标准 crate 集合。

```
crate Idx::<CData>(0) ->core crate Idx::<CData>(1) ->
编译器内置项 crate Idx::<CData>(2) ->alloc crate
Idx::<CData>(3) ->恐慌_中止 crate Idx::<CData>(4) ->
libc crate Idx::<CData>(5) ->unwind crate Idx::<
CData>(6) ->恐慌_unwind crate Idx::<CData>(7) ->
std_detect crate Idx::<CData>(8) ->cfg_if crate Idx::<
CData>(9) ->hashbrown crate Idx::<CData>(10) ->
rand_core crate Idx::<CData>(11) ->rand crate Idx::<
CData>(12) ->rand_xorshift crate Idx::<CData>(13) ->
rustc_demangle crate Idx::<CData>(14) ->std crate
Idx::<CData>(15) ->rustc_literal_escaper crate Idx::<
CData>(16) ->proc_macro crate Idx::<CData>(17) ->
getopts crate Idx::<CData>(18) ->test
```

列表 4：std 工作区中的 crate

3.5.1 利用 Rust-Analyzer 的提取方法

有了有效工作区后，REM2.0 直接依赖 RA 内置的提取函数辅助功能。REM2.0 不会重新实现提取逻辑，而是向 RA 发出与客户端编辑器相同的内部请求：提供文件内容、选定范围和期望的函数名。随后，RA 会返回一项结构化编辑，其中描述了：（1）一个新函数，其类型、泛型、生命周期和 where 子句均已完整推断；（2）在提取位置重写后的调用点。

这种委托至关重要。RA 已经包含了复杂的名称解析、trait 和 impl 查找、async/await 语法糖展开、常量求值、高阶 trait 约束以及非局部控制流分析逻辑。通过将其提取方法实现作为语义事实来源，REM2.0 继承了对完整 Rust 语言特性的支持，包括那些手动处理起来极其困难或脆弱的特性。这也使 REM2.0 能够充分受益于该领域多年来的持续开发，让我们可以专注于扩展整个 REM2.0 工具链的能力，而不是尝试重写已有的功能。我们还可以查询 RA 返回的语义分析结果，以改进后续的修复和验证阶段。

REM 2.0 将 RA 返回的编辑视为语义蓝图，仅在生命周期修复、调用点更新和可选验证方面按需调整。REM 无需维护自己的并行程序状态，而是可以直接向 RA 查询任何额外的类型或生命周期信息，从而在利用 RA 成熟转换的基础上，添加自身的结构层和正确性检查。

3.6 VSCode 集成与 REM 守护进程

为了让日常 Rust 开发者能够便捷地使用提取功能，REM2.0 通过专用的 VSCode 扩展提供服务，该扩展由一个持久运行的“REM 守护进程”支持。守护进程作为长期运行的进程运行，并通过轻量级 JSON-RPC 协议经由 `stdin/stdout`与扩展通信，从而实现极速重构，避免反复启动进程所带来的开销。启动时，扩展会执行环境检查，以确保 Daemon、CHARON、AENEAS 和 Coqc 均可用；如果不可用，扩展会直接在编辑器中显示针对性的安装说明。从用户角度来看，提取功能表现为标准的 VSCode 代码操作：选择一个区域后，会触发一个面板，其中提供新函数命名选项；在启用等价性检查时，还会明确反馈该重构是否保持语义。该架构在隐藏底层流水线复杂性的同时，为开发者提供了无缝且熟悉的使用体验。图 5 下方列出了呈现给开发者的选项，这些选项还可以设置键盘快捷键，以便使用。

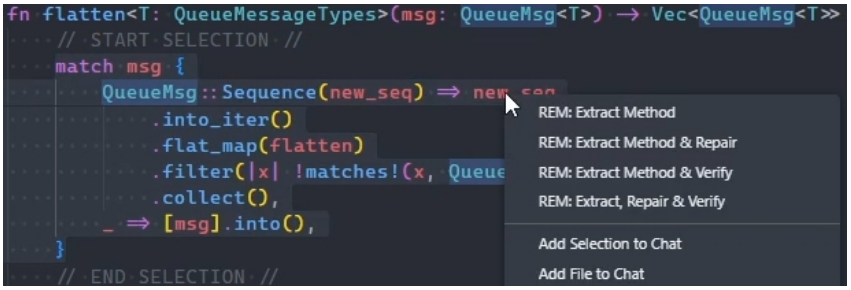

图 5：用户选择要提取的代码区域后显示的选项。

3.6.1 为验证准备重构

尽管验证流水线在第 4 节中有详细介绍，并在第 4.5 节中说明了具体的实现细节，但 REM2.0 会在提取时执行少量准备工作，以确保能够高效地检查重构。对于每次提取，REM2.0 都会将原始函数的函数体和新生成的函数分别捕获到相互隔离的“虚拟 crate”中，同时保留 RA 返回的准确语法和推断类型。这些 crate 构成了 CHARON 所需的最小输入，使验证器能够仅比较与转换相关的代码，而不是周围的项目。

4 重构代码的自动等价性证明

严格来说，重构被定义为一种保持行为的代码转换。在实践中，只有保留原始程序的行为，重构才有价值。在 Rust 中，所有权和生命周期会强制保证严格的不变量，因此仅仅因为重构后的代码能够编译，并不能保证其等价性。此外，由于 REM2.0 在提取过程中会执行复杂的编译器引导的修复 [2], ，引入程序行为细微变化的风险也随之增加。别名或生命周期结构上的细微变化，可能导致程序通过编译器检查，却在语义上偏离原始程序 [29]。此外，正如第 5.4.3 节所示，自动提取方法工具可能会生成仍能编译的错误提取结果。对于高保障要求的领域而言，这种风险不可接受：自动化工具不仅必须生成能够编译的代码，还必须提供转换正确的证据。

本节介绍一种新颖的等价性证明流水线，它扩展了 REM2.0，可自动、无需注释地证明原始代码与重构后代码之间的等价性。我们的方法结合了现有的形式化方法工具链——CHARON [11]，它将 Rust 转换为显式表示所有权的中间形式；以及 AENEAS [12]，它通过一种基于纯 λ 演算的中间表示生成与原始 Rust 源代码等价的 Coq 代码。随后，我们使用这些 Coq 代码，以形式化方式证明重构后的代码与原始代码等价。由此形成一个端到端系统：在 Extract →Repair 循环之后执行验证阶段，并在数秒内完成证明，不给开发者增加额外负担。通过将自动化等价性证明直接嵌入重构流程，我们让开发者能够

[11]更多信息请访问其 GitHub，https://github.com/AeneasVerif/charon[12]更多信息请访问其 GitHub，https://github.com/AeneasVerif/aeneas

超越编译成功，实现真正的语义保证，同时弥合理论与面向开发者的实用工具之间的鸿沟。

在本节（以及前文）中，我们偶尔会将等价性检查器称为“验证器”。我们承认，它所执行的工作严格来说并不属于“验证”，但这个称呼更符合本文的语言表述。

4.1 Whatif提取出错?

自动提取并非万无一失：对所有权或可变性的错误处理可能会在不知不觉中改变程序语义。为了说明我们构建验证器的原因及其在 REM2.0 中的作用，我们考虑一个简单但具有代表性的示例：提取器错误地处理了一个可变绑定[13]。验证器正确识别出两者不等价，并通过 VSCode 扩展界面向开发者报告失败，同时提供自动恢复到先前状态的选项。图 6展示了我们所讨论的情况，包括正确提取（将可变引用传递给 x，即 `&mut x`）以及错误提取（改为传递克隆的“x”，即 `x.clone()`）。由于编译器在编译时无法证明 `x` 的情况，这一错误绕过了 Rust 的安全保证，因此会导致运行时崩溃，如列表5所示。

```rust
fn main() {
    let mut x = 0;
    x = 3;
    assert!(x == 3);
}
//
//
//
//
```

(a) 原始代码

```rust
fn main() {
    let mut x = 0;
    mutate_x(&mut x);
    assert!(x == 3);
}

fn mutate_x(x: &mut i32) {
    *x = 3;
}
```

(b) 正确提取

```rust
fn main() {
    let mut x = 0;
    mutate_x(x.clone());
    assert!(x == 3);
}

fn mutate_x(mut x: i32) {
    x = 3;
}
```

(c) 错误提取

图 6：原始程序、正确提取以及错误提取的比较；错误提取采用克隆而非可变借用。

```
线程'主函数' (543251) 在 src/main.rs:6:5 处发生恐慌：断言失败：x == 3提示：设置‘RUST_BACKTRACE=1‘ 环境变量以显示回溯
```

清单 5：运行错误提取时的终端输出。程序在运行时崩溃，而不是在编译时。

从这里开始，我们生成一个语义等价的文件，但使用 Coq 证明助手语言编写。在下方的图7 中，我们列出了这些转换的关键部分。其他方面（例如将 Rust 类型转换为 Coq 类型的复杂定义）暂不涉及，稍后将在第4.5节讨论。还请注意，rustc 已对

[13]我们的提取流水线不会错误地执行这项提取，但以前的工具，尤其是在 Adventure of a Lifetime 中接受评估的工具，可能会犯类似的错误。这个示例展示了这样一次提取失败。

转换前的代码。在此阶段，无法在 CHARON 中禁用它们，因此代码清单7a 在视觉上与 Rust 实现不同。

```
(** [orig::main]:
    Source: 'src/main.rs', lines
     1:0-5:1 *)
Definition main : result unit :=
  massert (3%i32 s= 3%i32).
(* *)
(* *)
(* *)
(* *)
(* *)
```

(a) 原始代码

```
Definition mutate_x (x : i32) :
    result i32 :=
  Ok 3%i32.

Definition main : result unit :=
  x <- mutate_x 0%i32; massert (x
     s= 3%i32).
(* *)
(* *)
(* *)
```

(b) 正确提取

```
Definition mutate_x (x : i32) :
    result unit :=
  Ok tt.

Definition main : result unit :=
  let i :=
     core_clone_impls_CloneI32_clone
      0%i32 in
  _ <- mutate_x i;
  massert (0%i32 s= 3%i32).
```

(c) 错误提取

图 7：原始程序、正确提取以及错误提取的对比；错误提取采用克隆而非可变借用。

获得 AENEAS 生成的正确提取结果和错误提取结果的 Coq 表示后，我们现在执行自动等价性检查。此步骤具体实现了第4.4节所述的证明义务。本质上，验证器会构造并完成一项证明，证明对于调用方函数的所有可能输入，原始程序与重构后的程序都会产生相同的结果。由于验证是在 Coq 纯函数式语义中进行的，不受隐藏副作用或可变状态的影响，因此所得证明为行为等价提供了有力保证。

```
(** EquivCheck.v Autogenerated by REM-verification **)
Require Import Primitives.
Import Primitives.
Require Import Orig.
Require Import Good.
Local Open Scope Primitives_scope.


Lemma main_equiv_check :
    Orig.main = Good.main.
    Proof.
    intros.
    unfold Orig.main.
    unfold Good.main.
    reflexivity.
    Qed.
```

(a) 等价性检查（正确提取）

```
(** EquivCheck.v Autogenerated by REM-verification **)
Require Import Primitives.
Import Primitives.
Require Import Orig.
Require Import Bad.
Local Open Scope Primitives_scope.


Lemma main_equiv_check :
Orig.main = Bad.main.
Proof.
intros.
unfold Orig.main.
unfold Bad.main.
reflexivity.
Qed.
```

(b) 等价性检查（错误提取）

```
COQ_PROJECT: cases/4.1_examples/good_verif/_CoqProject
EQUIVCHECK: cases/4.1_examples/good_verif/EquivCheck.v
PRIMITIVES: cases/4.1_examples/good_verif/Primitives.v
SUCCESS: true
OUTPUT_END
```

(c) 验证器输出（正确提取）

```
COQ_PROJECT: cases/4.1_examples/bad_verif/_CoqProject
EQUIVCHECK: cases/4.1_examples/bad_verif/EquivCheck.v
PRIMITIVES: cases/4.1_examples/bad_verif/Primitives.v
SUCCESS: false
OUTPUT_END
```

(d) 验证器输出（错误提取）

图 8：正确和错误提取结果的等价性检查结果对比。验证器确认第一种情况下语义等价，并正确拒绝克隆变体，判定其不等价。

在此示例中，错误提取的版本用克隆值（`x.clone()`）替代可变借用，从而改变[illegible]义，同时仍能生成可编译的 Rust 代码。通过图 8中展示的等价性检查，等价性检查器通过对两个版本的 Coq 表示进行符号比较，检测到这一差异，并拒绝将转换后的函数判定为等价。在 REM2.0 工作流中，VSCode 扩展会直接向开发者显示这一失败，并提供清晰的错误消息，同时附带自动恢复到提取前状态的选项。

4.2 展示非平凡验证

上一节展示了一个刻意简化的案例，旨在介绍这一概念，并说明等价性检查器如何防范明显的语义破坏。相比之下，实际的重构可能涉及嵌套循环、常量泛型、所有权细节以及类型级不变量；它们相互作用的方式，远没有表层语法所呈现的那么直观。本节将探讨一个更贴近实际的例子，该例子受到 Vaultwarden 代码库中发现的一处错误提取的启发。在这个例子中，重构后的代码能够编译并通过类型检查，表面上也合乎情理，却在不知不觉中改变了程序的行为。这里旨在说明，验证器并非只检查语法结构，也不是轻易就返回成功，而是能够检测出仅凭 Rust 类型系统无法排除的细微语义差异。

```rust
pub fn bump<const N: usize>(
    arr: &mut [i32; N],
    k: usize,
    mark: &mut bool,
) where [i32; N]: Copy, {
    let mut i = 0;
    let limit = if k < N { k } else { N };
    while i < limit {
        if arr[i] == 0 {
            *mark = true;
        }
        arr[i] = arr[i] + 1;
        i += 1;
    }
}
fn main() {
    let mut a = [0, 0, 5, 9];
    let mut touched_zero = false;
    bump(&mut a, 3, &mut touched_zero);
    assert_eq!(a, [1, 1, 6, 9]);
    assert!(touched_zero);
}
//
//
//
//
//
```

(a) 原始代码

```rust
fn bump_extracted<const N: usize>(
    mut arr: [i32; N], i: usize, mut mark: bool
) where [i32; N]: Copy, {
    if arr[i] == 0 {
        mark = true;
    }
    arr[i] = arr[i] + 1;
}
pub fn bump<const N: usize>(
    arr: &mut [i32; N],
    k: usize,
    mark: &mut bool,
) where [i32; N]: Copy, {
    let mut i = 0;
    let limit = if k < N { k } else { N };
    while i < limit {
        bump_extracted::<N>((*arr).clone(), i, *mark);
        i += 1;
    }
}
fn main() {
    let mut a = [0, 0, 5, 9];
    let mut touched_zero = false;
    bump(&mut a, 3, &mut touched_zero);
    assert_eq!(a, [1, 1, 6, 9]); // fails: a: [0, 0, 5, 9]
    assert!(touched_zero); // fails
}
```

(b) 错误提取

图 9：引擎生成的原始函数和错误提取。示例的灵感来自 Vaultwarden 提取失败，参见第 5.4.3

乍看之下，这种错误提取似乎无伤大雅：循环结构得以保留，原地更新仍嵌套在迭代器中，而由于数组满足 Copy 约束，Rust 类型系统也接受了这种转换。然而，由于提取出的函数接收数组时采用的是按值传递，而非引用，它修改的只是一个临时副本，随后便将其丢弃。这种细微的偏差很难通过人工发现，编译器也无法拒绝。不过，等价性检查器通过比较两个版本的 Coq 翻译，发现转换后的程序没有改变调用方的数组，从而检测出了这一不匹配。如图 10所示，对这处错误提取进行的等价性检查未能通过[14]，，而 REM2.0 能够将这一失败直接反馈给开发者，避免应用这项会悄然引入错误的重构。

[14]清单 9b中的断言在运行时也会失败；这里使用它们是为了演示 Rust 中的失败，并模拟单元测试。

```
Definition bump
  {N : usize} (coremarkerCopyArrayI32NInst :
     core_marker_Copy (array i32 N))
  (arr : array i32 N) (k : usize) (mark : bool) :
  result ((array i32 N) * bool)
  :=
  if k s< N then bump_loop0 k arr mark 0%usize else
      bump_loop1 arr mark 0%usize
.


(* *)
(* *)
(* *)
(* *)
(* *)
(* *)
(* *)
(* *)
(* *)
(* *)
(* *)
(* *)
(* *)
(* *)
(* *)
(* *)
(* *)
(* *)
(* *)
```

(a) 原始代码

```
Definition bump
  {N : usize} (coremarkerCopyArrayI32NInst :
     core_marker_Copy (array i32 N))
  (arr : array i32 N) (k : usize) (mark : bool) :
  result ((array i32 N) * bool)
  :=
  if k s< N
  then (
    _ <- bump_loop0 coremarkerCopyArrayI32NInst arr
      k mark 0%usize;
    Ok (arr, mark))
  else (
    _ <- bump_loop1 coremarkerCopyArrayI32NInst arr
      mark 0%usize;
    Ok (arr, mark))
.

Definition bump_extracted
  {N : usize} (coremarkerCopyArrayI32NInst :
     core_marker_Copy (array i32 N))
  (arr : array i32 N) (i : usize) (mark : bool) :
  result unit
  :=
  i1 <- array_index_usize arr i;
  if i1 s= 0%i32
  then (_ <- i32_add i1 1%i32; _ <-
     array_index_mut_usize arr i; Ok tt)
  else (_ <- i32_add i1 1%i32; _ <-
     array_index_mut_usize arr i; Ok tt)
.
```

(b) 错误提取

```
COQ_PROJECT: cases/4.3_examples/verif/_CoqProject
EQUIVCHECK: cases/4.3_examples/verif/EquivCheck.v
PRIMITIVES: cases/4.3_examples/verif/Primitives.v
SUCCESS: false
OUTPUT_END
```

(c) 显示验证失败的终端输出

图 10：Coq 翻译及 REM2.0 等价性检查器的输出。// 为简洁起见，省略了部分 Coq 翻译内容。完整的转换请参见附录F 。

总之，这些结果表明，等价性检查器确实执行了非平凡的推理：只有当两个程序在行为上完全相同时，它才会通过；即使编译器接受了某种转换，且其句法结构看似有效，只要该转换破坏了行为等价性，它就会予以拒绝。这证明等价性阶段提供了实质性的正确性保证，而非表面化或总会通过的检查。

4.3 处理可能出现的边缘情况

虽然前面的章节通过深入探讨示例，说明了等价性检查器的必要性，但本节旨在展示我们如何处理使用提取引擎时可能出现的边界条件。我们的目标是展示：(i) rustc 会尽早拒绝无效模式，并将相关信息作为可操作的诊断信息呈现在 REM2.0 中；(ii) 类型正确但“退化”的形式（无参数、单元返回值、局部推断）能够顺利通过

CHARON/AENEAS 并按预期进行验证；(iii) 已知限制（例如 !）能够被识别，且不会导致虚假的“成功”。下表展示了这些边界情况以及我们对其处理结果的预期：

Case	最简示例	rustc / Cargo	翻译	等价性结果 / 注释
无输入参数、单元类型返回值，或两者兼有	fn ping() {/* ... */ } fn reset(x: &mut 32 位整数) { *x = 0; }	OK	OK	能够展开的证明引擎不带 全称量词 子句[自反性保证等价性。
单元返回值（显式或推断）	fn touch(x:&mut 32 位整数){ *x += 1; } fn f()->() { }	OK	OK	通过相同内容进行等价性检查后状态；返回值为单元类型。保留了副作用建模。
_ 在参数中或返回类型	函数 g(x: _)-> i32 { x+1 } 函数 h(x:32 位整数)->_ { x*2 }	错误（无效签名形式）	N/A	已被编译器在上游拒绝。未尝试进行转换或证明。
_ 在局部绑定（位于函数体内）	let x: _ = 42; let y = x+1;	OK	OK	类型在 MIR 之前或在 MIR 阶段被具体化；Coq 具有具体的整数类型。验证器照常成功。
!（永不）返回 type	fn abort()->! { panic!("fail") }	OK	失败	不纳入验证集合。REM 2.0 报告不支持的功能诊断信息；可以继续提取但不会进行验证。
非局部控制流位于提取区域	内部的跳出/继续循环体	OK	OK	控制流表示为 std::ops::ControlFlow。验证器进行模式匹配并证明行为等价性。

表1：提取与验证的边缘情况覆盖。“Translate”指 CHARON/AENEAS；“Verifier”指等价性检查器。

综合来看，这些案例表明，流水线既不会沦为平凡证明，也不会默默接受不支持的构造。涉及的无效签名会被 rustc 拦截并告知用户；结构简单但类型正确的形状_(无参数、单元返回值、局部 ）可以顺利提取并通过验证；而已知的限制（例如，!）则会被明确报告，并排除在证明之外。这既广泛覆盖了各种边缘情况，也确保了验证器所提供的保证有意义且值得信赖。尽管设计规范和实现细节见第 4.5 节和附录H，但我们在此展示的内容体现了等价性检查器的核心理念：在能力所及的范围内，它绝不能声称两段实际上并不等价的代码彼此等价。

4.4 证明义务

在验证流水线的最后阶段，我们会在 Coq 中证明一项形式化的 等价性义务 。直观地说，我们必须证明，对于所有有效输入，重构后的程序与原始程序的行为完全相同。核心思想是函数等价性：两个程序的可观察输出必须一致。

4.4.1 简化义务

我们用 $[\![f]\!]$ 表示函数 f 经 AENEAS 转换为 Coq 后的语义解释（含义）。简单来说，这项义务可以表述为：

$$\forall x \in \mathrm{Dom}(f).\ [\![f]\!](x) = [\![f']\!](x)$$

其中，f 是原始函数，f' 是其重构后的对应函数。这表示，对于所有可能的输入构造 x，转换后的函数都会返回相同的结果类型。在实践中，我们尝试自动证明的就是这种形式，如图 8a所示。

4.4.2 全程序上下文

要进行更完整的刻画，我们必须承认，函数从来不是孤立执行的，而是作为外围程序的一部分执行。因此，可以将义务强化为全程序等价：

$$\forall x \in \mathrm{Dom}(f).\ [\![f]\!](x) = [\![f']\!](x)$$

其中，P 是原始程序，P' 是重构后的程序，两者的区别仅在于 f 被替换为 f'。这体现了等价性必须不受函数内部使用方式的影响。也可以用上下文等价来表达同一概念：

$$\forall x \in \mathrm{Dom}(f).\ [\![f]\!](x) = [\![f']\!](x)$$

其中，$C[\cdot]$ 表示任意程序上下文。这强调了将 f 替换为f' 后，在任何调用环境中都能保持行为不变。只有在我们转换到的无副作用函数式环境中，这一点才成立。

4.4.3 与 PriorWork 的关系

早期关于 AENEAS 的工作以一种有意保持一般性的形式陈述等价性，其中包括前置条件 $\Phi(x)$、效应轨迹 τ，以及将结果投影到可观察行为上的 $obs(\tau)$。这些因素对于处理非纯特性、部分程序和更丰富的效应语义是必要的。相比之下，我们的流水线有意将关注范围限制在更简单的设置中。REM2.0 生成类型正确且经过借用检查的 Rust 代码片段，而 AENEAS 则应用于不包含内部可变性或并发的安全 Rust 代码。因此：

- 可以省略前置条件，因为 REM 确保了类型正确性和借用安全。
- 可以省略效应轨迹，因为除返回值之外，我们的代码片段不产生副作用。
- 可观察投影可简化为结果之间的普通相等比较。
- AENEAS 保证其转换结果可直接映射回 Rust 源代码。如果某件事在 AENEAS 的转换中成立，那么它在 Rust[15, 16] 中也成立。因此，为了维护这一点，我们无需额外进行证明工作（就我们而言）

这种简化产生了一个更易于表述和自动化处理的义务，同时仍然涵盖了重构保持行为不变这一核心要求。

4.4.4 实际影响

简化义务凸显了一个指导性设计原则：该流水线旨在补充提取算法，而不是对其进行过度近似。因此，最好

在遇到不支持的构造和语言特性时尽早失败，而不是冒险错误地为程序出具认证。AENEAS 将不支持的特性（例如闭包或嵌套借用）报告为转换失败，而 Coq 实现则明确区分无法证明的等价性与不支持的语言子集。这种分离确保开发者永远不会收到误报：代码要么验证成功，要么明确失败，并提供清晰的诊断信息。由此自然引出了覆盖范围这一问题：当前 AENEAS 支持哪些 Rust 特性，还有哪些仍是尚待解决的问题？我们将在本节最后一部分，即第4.6节中对此进行讨论。

4.5 等价性流水线的实现

REM2.0 的自动等价性检查设计遵循两项原则：（1）不要求开发者提供任何额外注解或输入；（2）必须快速完成，以适应交互式 IDE 工作流。为此，流水线将原始程序 P 和重构后的程序 P' 依次转换为结构更完善的语义表示，最终得到观察等价性的机器验证证明。为简洁起见，本节仅对架构作高层次概述；附录 H包含完整的技术细节和示例。

从整体上看，流水线分为五个阶段：**提取**、**REM 修复**、**CHARON 翻译**、**AENEAS 函数化**和**Coq 验证**。每个阶段都以前一阶段为基础，逐步揭示重构后代码的语义，直至我们在 Coq 中获得两个版本的纯函数式模型。

4.5.1 提取与修复

流水线从 REM2.0 的提取引擎开始。给定用户选区后，提取器会将该区域提升为一个新函数，并将其替换为调用点适配器。提取器无法最终确定生命周期，因此会将这项工作交由 REM 的提取后修复阶段处理。随后，REM 会执行生命周期修复，使提取后的程序 P' 符合 Rust 借用检查器的要求。它将编译器视为“预言机”，反复修复生命周期，直至程序编译通过；并在可能的情况下应用生命周期省略，以恢复惯用类型，并尽可能使输出“易读”。最终得到的是一个经过修复且能够编译的程序版本。

4.5.2 通过 CHARON 和 AENEAS 进行转换

接下来的阶段会将 Rust 程序转换为面向验证的表示形式。CHARON 获取 P 和 P' 的编译后 MIR，并生成 LLBC（低级借用演算），这是一种仅供机器读取的形式，其中 Rust 背后的所有逻辑都被 100% 明确地表示出来。为了确保这一过程不会因用户工作区中的编辑而受到影响，REM2.0 会构建临时的“虚拟”crate，它们是与原始 crate 在内存中隔离的副本，其中包含原始函数或重构后的函数。这些虚拟 crate 是启动 CHARON 所必需的。

接下来，AENEAS 会将 LLBC 转换为纯函数式模型，在保留 Rust 程序语义的同时，移除显式的栈和堆操作。在这种形式下，程序被表示为对值进行运算的全函数，而不是使用可变引用，这种形式更适合

证明助手。在我们的流水线中，AENEAS 面向 Coq，但原则上也可以生成适用于其他系统（如 Lean 、 HOL4 或 F* ）的编码。

4.5.3 在 Coq 中进行等价性检查

最后一个阶段会组装一个 Coq 项目，其中包含 P 和 P'（由 AENEAS 生成）的函数式编码，以及一个自动生成的模块 `EquivCheck.v`。该模块会实例化这两个版本，生成第 4.4 节所述的等价定理，并调用必要的自动化工具，以证明对于所有输入，它们都会产生相同的可观察结果。Coq 项目编译完成后，等价性检查便完全自动化：如果证明成功，我们就能获得机器检查的保证，确认重构保留了原有行为。关于底层 Coq 基础设施、替代后端、示例证明脚本等内容，附录 H 中有详尽讨论。

4.6 当前验证的限制

目前，我们的流水线所支持的验证保证受限于 AENEAS 能够转换的 Rust 子集。AENEAS 专为安全 Rust 设计；使用 `unsafe` 结构的程序不在其作用范围内。尽管这排除了部分底层系统代码，但这是有意为全 Rust 已涵盖大多数惯用 Rust 用法，长期计划是将 AENEAS 与面向 `unsafe` 代码的互补工具集成。目前的支持情况汇总于表 2。

功能	状态
安全 Rust	完全支持（本表中注明的情况除外）
不安全 Rust	不支持（超出 AENEAS 的作用域）
循环	支持（不支持嵌套循环）
函数指针 / 闭包	不支持（正在开发中）
特征	支持
类型参数性	受限：不支持带借用的类型参数
函数签名中的嵌套借用	不支持（正在开发中）
内部可变性（Cell、RefCell 等）	不支持（计划通过幽灵状态实现）
并发	暂不在范围内（长期研究目标）

表 2：AENEAS 当前对 Rust 特性的支持情况

在安全 Rust 中，仍存在若干技术限制。循环目前仅支持受限形式（不支持嵌套循环），不过这方面正在积极开发中。函数指针和闭包尚不可用，但目前已支持基于 trait 的抽象，且正在进行的工作旨在将支持扩展至一等函数。类型参数性也受到限制：目前无法将泛型类型参数实例化为自身包含借用的类型。同样，函数签名目前也不能包含嵌套借用，不过这一问题也正在解决。内部可变性是另一项尚待解决的挑战；当前计划是使用幽灵状态来捕获其影响。最后，并发执行不在支持范围内：AENEAS 仅对顺序 Rust 程序建模，并行性则被视为长期研究目标。

需要强调的是，验证流水线旨在补充抽取算法。因此，遇到不支持的特性时，验证最好尽早失败，而不是冒险错误地认证代码。因不支持的语言结构而导致的验证失败，与证明义务未能通过所导致的失败，会以不同方式报告给开发者

。这种区分可以避免假阳性——即无法验证的代码却被视为已通过验证的情况。。

总之，当前流水线最适合被理解为一个针对顺序执行的安全 Rust 程序的等价性框架，不支持嵌套借用、闭包或并发等特征。这些限制反映的是 AENEAS 的当前状态，而非根本性障碍；目前的工作正逐步扩展其支持的子集。随着 AENEAS 的不断发展，我们预计等价性检查器将能够处理完整的 crate。关于这些限制以及完整 REM2.0 流水线目前预期的使用场景，将留待第 6 节详细讨论。

5 评估与实验结果

本节从两个不同方面评估新版 REM2.0：其对真实 Rust 代码执行提取函数重构的能力，以及验证这些重构是否保留行为的能力。我们首先介绍基准测试语料库和实验设置，然后定义用于评判成功与否的指标和标准。接着，我们展示提取流水线的结果，随后介绍基于 CHARON 和 AENEAS 构建的可选验证流水线的结果，并着重说明相较于原始 REM 原型所取得的改进以及仍然存在的限制。

5.1 评估目标与研究问题

本次评估的总体目标是确定 REM2.0 在真实 Rust 代码上的表现：既要评估它作为快速、日常使用的提取函数重构工具的表现，也要评估它与验证流水线结合时，能否作为形式化保证的可选来源。具体而言，我们关注其正确性、特征覆盖范围和性能，并希望了解验证器带来的权衡。我们将这些目标归纳为以下研究问题：

- **RQ1（提取正确性与覆盖范围）**。新的提取工具链在多大程度上能在原始 *REM* 基准套件上保留类型和行为？是否存在两者都无法处理的情况？

- **RQ2（新特性支持）**。新的提取引擎如何将覆盖范围扩展到现异步/等待、常量函数、泛型、高阶 *trait* 约束、动态 *trait* 对 [illegible] 特性？这些特性中呈现出哪些成功与失败模式？

- **RQ3（性能）**。新的架构在真实项目上能达到怎样的提取延迟？与原始 *REM* 原型相比，量化结果如何？

- **RQ4（验证）**。对于 *CHARON* 和 *AENEAS* 当前支持的 *Rust* 子集，验证流水线在确立等价性方面的效果如何，其性能特征又是什么？

各小节5.2–5.4 主要通过以提取为重点的实验回答 RQ1–RQ3，而第 5.5小节则通过在一组精心挑选的

简单案例与真实场景。

5.2 基准语料库与实验设置

本小节介绍用于评估 REM2.0 提取和验证组件的基准测试语料库与实验设置。我们首先概述实验，然后介绍本节其余部分使用的三个语料库：原始 REM 基准套件、一组新的真实世界特性案例，以及一个根据 CHARON 和 AENEAS 当前能力定制的验证基准。

5.2.1 实验概述

为同时涵盖向后兼容性和新功能，我们设计了三个互补实验：

1. **原始 REM 案例（基线提取）**。我们在评估原始 REM 原型时所使用的完整基准测试案例套件上，重新运行新的提取流水线。该实验主要用于通过将正确性和延迟与基线设计进行比较，回答 RQ1 和 RQ3。

2. **新的真实世界特性案例（扩展提取）**。我们从大型且积极维护的 Rust 代码仓库中选取提取位置，构建了一个包含四十个案例的新语料库。每个案例的选取依据是：它至少涉及一种原始 REM 无法可靠支持的现代 Rust 特性。该实验旨在回答 RQ2 和 RQ3。

3. **验证基准（等价性检查）**。我们为验证流水线汇集了二十个示例：十个用于单独检验核心正确性属性的小型“平凡”程序，以及十个取自开源代码的简单但来自现实世界的函数。这些示例均选自当前受 CHARON 和 AENEAS 支持的 Rust 语言片段，并用于回答 RQ4。

5.2.2 原始 REM 基准测试套件

为提供比较基线，我们重新使用了原始 REM 评估中的全部提取案例。[15] 每个案例都由一个包含带注释源文件的 Rust 项目构成，其中预期提取区域通过以下任一方式标记：行注释（`// START SELECTION //` 和 `// END SELECTION //`）或块注释（`/* START SELECTION */` 和 `/* END SE`

每个基准测试案例都由原始 REM 工件中的选择标记定义。我们沿用这些标记，以确保原始原型和新的提取流水线都在完全相同的 Input 上运行。原始套件中的一个案例（案例 #16）不含选择标记，因此未纳入我们的定量比较；其余案例的结果和耗时见表4。

5.2.3 新的真实案例

为了评估新的提取引擎在现代 Rust 代码中“实际存在”的模式上的表现，我们从大型、热门的开源项目中选取了四十个提取位置，构建了第二个语料库

[15] 测试代码仓库直接复制自 VerseLab 小组发布的工件，地址为 https: //github.com/verse-lab/rem。

项目。该语料库旨在考验原始 REM 原型无法支持的语言特性，同时确保尽可能多的示例反映开发者在实践中实际进行（或合理可能进行）的提取操作。

我们首先依据 Evan Li 维护的 GitHub [30] 排名，选取了星标数最高的一批 Rust 代码仓库作为样本，并以此B作为基准。所选代码仓库的总结（包括抽样时的星标、复刻和未解决问题数量）见附录 8。

对于这一组中的每个代码仓库，我们扫描了其 Git 历史，以定位可能与方法或函数提取相关的提交。自动扫描会在提交消息中搜索具有特征性的短语，例如 `extract`、`extract method`、`factor out`和 `refactor:extract`，随后结合正则表达式和 `git grep` 查询，标记出所有候选文件。对于每个代码仓库，我们将搜索深度限制为其最活跃的三个分 100 个提交。这一启发式过程不可避免地会产生假阳性（例如，提交提到“extract”却处于无关上下文中），但为我们提供了一个高效的起点。随后，我们手动检查每个被标记的提交/文件，寻找合适的提取位置，并使用前文所述的选择标记标注这些区域。

已识别提交的完整列表及各代码仓库的详细信息见附录B。这些案例的全部提取结果和计时结果将在本报告后文的表5、第5.4.2小节中列出。

5.2.4 验证基准

第三个语料库针对验证流水线，该流水线基于`CHARON` 和 `AENEAS`，用于证明提取出的函数在行为上与其原始版本等价。由于这些工具目前仅支持 Rust 的一个子集，我们特意构建了一个小型基准集，使其符合这些工具支持的语言子集，同时仍能检验非平凡的推理能力。完整的限制列表见第4.6 节。在此基础上，验证语料库包含二十个示例，分为两组：

- **简单/针对性案例（10 个示例）**。小型且独立的 Rust 函数，用于隔离核心属性，例如对整数进行纯计算、通过引用进行简单可变性操作，或基本控制流模式，且不涉及堆分配或复杂的借用。这些示例旨在暴露转换或证明生成流水线中的错误，而非提取引擎本身的问题。

- **简单的现实案例（10 个示例）**。我们编写了一个新的 crate，用于模拟现实中业务逻辑风格的 Rust 代码。函数在多个分层结构体上运行，依赖 trait 实现，并在 AENEAS 当前支持的 Rust 子集范围内，尽可能贴近生产模式。为此确实需要一些额外的脚手架代码，例如手动实现 `PartialEq`。更重要的是，这个 crate 具有外部依赖，包括 `smallvec`和 `anyhow`。

对于每个 验证基准中，我们对原始函数和提取出的函数运行完整流水线 s:

实验	目标	评估标准
原始案例	建立 正确性 and 新提取器相较于原始 REM 原型的性能（RQ1、RQ3）。	•原始 REM 基准测试套件。 • 编译结果持平或有所改善与原型相比。 • 提取延迟降低。
新功能案例	特征描述 支持 for 此前不受支持或支持不佳的现代 Rust 特性 此前不受支持或支持不佳支持者 REM （RQ2, RQ3）。	•各特征类别的提取成功率（GEN、ASYNC、CONST、NLCF、HRTB、DTO）。 •提取的代码的编译结果，其中失败按根本原因分类。 • 大型真实项目上的提取延迟及其在不同特征间的变化。
验证流水线	评估其有效性及 cost of the 等价性- 检查 流水线在 sup- 上的表现 移植的 Rust 代码片段（RQ4）。	•尝试进行验证并成功完成的案例比例已尝试并成功完成。 • 证明与预期语义的一致性（没有虚假的成功；失败可归因于工具限制或程序不等价）。 • 端到端验证时间 ，分解为 LLBC 转换、 Coq 转换和证明解析/构造/检查。

表 3：实验目标与评估标准总结。

，通过CHARON将其转换为低级借用演算（LLBC），通过AENEAS生成 Coq 证明义务，然后检查生成的证明。我们记录是否尝试了验证、验证是否成功，以及各阶段的耗时。这些结果列于第7表，见第5.5.2小节。

5.3 指标与评估标准

在全部三项实验中，我们测量：(i) 正确性与覆盖率 （提取或验证是否成功，以及失败发生的频率），(ii) 编译行为 （重构后的代码是否能够编译，以及与原始代码相比有何差异），以及 (iii) 性能 （端到端延迟，以及验证各个阶段的成本）。RQ1 和 RQ2 主要关注正确性、覆盖率和编译；RQ3 关注提取延迟；RQ4 则考察验证流水线的成功率和性能。表 3总结了各项实验如何确定这些要求。

5.4 提取评估

本小节针对提取流水线回答 RQ1—RQ3。我们提出三个具体问题：(i) REM2.0 在正确性和稳健性方面是否至少与原始 REM 原型在原始基准测试套件上表现相当（RQ1）；(ii) 它对现代 Rust 特性的覆盖范围有多广，例如 异步 、常量函数、泛型、高阶 trait 约束、动态 trait 对象和非局部控制 Q (iii) 它在实际应用中达到怎样的延迟表现，以及何时需要单独的 REM- Repairer 阶段才能获得可编译的结果（RQ3）。我们首先与原始 REM 工件直接比较，然后考察新的真实世界特性语料库，最后总结其定量表现和仍存的限制。

5.4.1 与原始 REM 的基线比较

#	项目（代码行数）	结果 REM	结果 REM2.0	时间（毫秒） REM	时间（毫秒） Raw	时间（毫秒） User	修复器必需
1	petgraph (20,157)	✓	✓	370	3.511	184.32	✗
2		✓	✓	1020	3.753	217.91	✗
3		✓	✓	1470	2.166	231.08	✓
4		✓	✓	1700	3.337	196.77	✗
5		✓	✓	850	2.018	128.54	✗
6		✓	✓	980	2.148	142.66	✗
7		✗	✓	550	2.030	255.19	✓
8	gitoxide (20,211)	✓	✓	930	2.851	199.45	✗
9		✓	✓	1240	3.240	173.82	✓
10		✓	✓	640	2.823	208.37	✗
11		✓	✓	810	2.991	244.68	✗
12		✓	✓	810	3.282	188.05	✗
13		✓	✓	860	2.995	163.49	✗
14		✓	✓	690	5.345	226.91	✗
15		✓	✓	680	5.468	203.74	✗
16		✓	无数据	540	N/A	N/A	✗
17		✓	✓	1200	3.283	179.40	✓
18		✓	✓	920	5.241	210.62	✓
19		✓	✓	2320	3.283	226.46	✓
20		✗	✓	1150	5.430	192.92	✓
21		✓	✓	690	3.396	123.84	✗
22		✓	✓	640	2.984	264.15	✗
23		✓	✓	700	3.153	132.97	✗
24		✓	✓	640	3.230	147.53	✗
25		✓	✓	720	3.457	221.84	✗
26	kickoff (1,502)	✓	✓	1030	3.802	185.09	✗
27		✓	✓	1010	8.557	237.26	✗
28		✓	✓	910	7.365	168.44	✗
29		✓	✓	980	4.864	206.59	✗
30		✓	✓	790	4.191	257.31	✗
31	sniffnet (7,304)	✓	✓	1040	7.288	121.06	✗
32		✗	✓	760	5.674	195.22	✓
33		✓	✓	1010	4.439	172.80	✗
34		✓	✓	980	3.602	234.13	✗
35		✓	✓	1060	3.265	141.95	✗
36		✓	✓	1000	3.507	260.42	✗
37		✓	✓	1060	4.147	209.61	✗
38		✓	✓	1080	4.284	187.32	✗
39		✓	✓	1060	4.273	149.40	✗
40	比鲁斯 (302)	✓	✓	1070	5.158	224.75	✗

表 4：在相同案例上比较 REM2.0 与原始 REM 工具链。原始时间仅指提取所用时间，用户时间则表示 VSCode 报告的系统总体延迟。

表4 比较了原始 REM 原型与 REM2.0 在第5.2.2小节所述完整基准测试套件上的表现。除去缺少选择标记、因此无法评估的案例 #16，新提取器在其余 39 个案例中均成功。相比之下，原始 REM 原型在其中三个案例（案例 #7、#20 和 #32）上失败，而 REM2.0 成功提取了这三项。至关重要的是，没有出现退步——我们未发现 REM 成功而 REM2.0 失败的案例。

时间列简洁地展示了新架构对性能的影响。原始 REM 流水线每次提取需要数百到数千毫秒（平均约 1000 毫秒）。相比之下，在守护进程内部测得的 REM2.0「原始」时间——从收到提取请求到应用编辑的耗时——始终处于个位数毫秒范围（所有案例约为 2 毫秒至 9 毫秒）。「用户」列计入了修复器（在调用该修复器的情况下）和 VSCode 的开销，但用户感知到的延迟仍在 120 毫秒至 260 毫秒之间。考虑到人类对视觉刺激的响应时间约为 200 毫秒至 250 毫秒， [31], 这一表现完全可以接受。附录E中的直方图展示了更详细的延迟分布。

最后一列报告了需要调用 REM-Repairer 来恢复程序编译的频率。在原有的 40 个 REM 案例中，该工具在 7 个案例中被调用，且恰好与原始 REM 评估中相同的案例一致（这些案例的生命周期约束更严格）。这表明，尽管新的提取器往往能独立生成可编译的代码，修复器仍是工具链中不可或缺的组件，而非纯粹可有可无的附加功能。

5.4.2 新语言特性覆盖

现在我们来考察第5.2.3小节中引入的新语料库。该语料库旨在测试原始 REM 原型无法处理的高级 Rust 特性。每项提取结果会被归入一个或多个特性类别：泛型（GEN）、异步/等待（ASYNC）、常量求值（CONST）、非局部控制流（NLCF）、高阶 trait 约束（HRTB）

，

以及动态 trait 对象（DTO）。这些类别直接对应于真实 Rust 代码中通常会给类型推断、生命周期推理或控制流重建带来挑战的情形。表5 中任一特性列里的标记，都表示相应示例在所选区域内包含该构造。附录G中列出了各类别的代表性代码，并简要说明我们期望提取器保留哪些内容。

在新增的四十个提取案例中，REM2.0 成功提取了三十三个（并生成了可编译的程序），总体成功率约为 83%。纵观已有类别可以发现，大多数现代构造都能得到稳健处理：除一个以外，所有 ASYNC 案例均成功（7/8）； CONST 几乎所有示例中的提取结果都成功（10/11）；而 NLCF 在大多数情况下（11/13）都被正确还原为`std::ops::ControlFlow` 。高阶 trait 约束的处理也相当不错（6/8），失败主要是因为复杂。泛型函数本身在大多数示例中也能成功处理（11/14），其余失败则源于泛型约束重建不完整。

当前提取器的主要弱点在于动态分派。四种 DTO情况中只有一种成功；其余三种情况下，推断出的签名会退化为约束不足的形式（例如使用过于宽松或不足的 trait 约束），进而导致代码无法编译。一个特别能说明问题的例子是失败 #4：其中，类型为 `AsRef<[u8]>` 的参数被错误地泛化为 `?Sized`，从而丢失了 trait 与具体切片类型之间的关联。我们将在中更详细地讨论这些失败模3。

#	项目（代码行数）	代码特性 GEN	ASYNC	CONST	NLCF	HRTB	动态	Res	Time (ms)
1	deno (359996)			✓				✓	5.882
2					✓		✓	✗(1)	N/A
3					✓			✓	4.346
4		✓			✓			✓	9.685
5			✓					✓	2.910
6	fuel-core (78921)				✓			✓	6.736
7			✓					✓	2.999
8	Meilisearch (186553)	✓						✓	4.363
9		✓				✓		✓	12.071
10		✓				✓		✗(2)	N/A
11				✓				✓	3.071
12				✓				✓	3.852
13	ripgrep (40453)	✓			✓			✓	3.117
14					✓			✓	2.089
15	starship (37225)				✓			✓	1.955
16		✓						✓	1.866
17	sway (108725)			✓				✓	4.802
18				✓				✓	3.073
19		✓				✓		✓	2.365
20	tauri (83582)	✓				✓		✓	1.183
21		✓			✓			✓	8.565
22					✓			✓	3.054
23					✓			✓	5.346
24	union (514914)		✓			✓		✓	3.175
25				✓				✓	2.864
26			✓		✓	✓		✗(3)	N/A
27							✓	✗(4)	N/A
28	uv (321584)			✓				✓	16.703
29		✓			✓	✓		✓	6.451
30				✓				✓	2.873
31		✓						✗(5)	N/A
32	vaultwarden (29937)			✓	✓			✓	5.134
33			✓					✓	3.057
34		✓		✓				✗(6)	NA
35	zed (888269)						✓	✗(7)	N/A
36			✓				✓	✓	7.383
37				✓				✓	2.062
38		✓	✓			✓		✓	3.667
39		✓						✓	1.959
40			✓					✓	1.853

表 5：提取先前不可用或不兼容的语言特性的结果。失败原因的更多详情见表 6

#	原因
1	无法推断（部分）函数签名
2	无法推断（完整的）返回类型
3	无法确定正确的提取边界（未尝试提取）
4	`AsRef<[u8]>` 在调用方中被错误地映射为 `?Si` 用点中
5	返回签名 `_` 而不是 结 `, S::Error>`
6	无法将任何泛型约束复制到新签名
7	无法推断函数签名（或其中一部分）

表 6：表中失败案例提取失败的原因5

5.4.3 失败案例与限制

表5 中的七个失败案例可归纳为少数几种反复出现的模式，汇总见表6。总的来说，这些问题源于提取引擎在生成

根据局部信息推导函数签名，而不是将选定的代码移入新的函数体。

第一类失败源于返回类型或参数列表的类型推断不完整（失败示例 #1、#2 和 #7）。在这些情况下，RA 能够对原始代码进行类型检查，但提取边界处可用的信息不足以合成具体的函数签名。由此产生的占位类型（例如返回类型）不被编译器接受，因此该提取被记录为失败。不过，在所有这些情况下，编译器都能建议合适的返回签名，因此 REM-Repairer 未来的一项潜在改进是扩展其功能以处理这些情况。

第二类失败涉及泛型约束和 trait 信息（示例 #4、#5 和 #6）。在这里，提取器能够识别签名的正确总体形态，但要么遗漏必要的约束（例如缺少 `where` 子句或类型参数上的 trait 约束），要么对具体的 trait 应用进行了过度泛化。`&Ref<[u8]>` to `?Sized` 在失败 #4 中的误译就是一个典型例子：新签名不再体现参数与切片类型之间预期的关联，从而引发多处下游错误。在这种情况下，Rust 编译器无法建议合适的修复方案。

这些模式在 DTO 案例中很常见，因为重构精确的 dyn Trait （或类似的 `dyn` 签名所需的类型信息通常位于当前文件之外，依赖复杂的 `where` 子句。由于 REM2.0 目前采用程序的（主要是）单文件视图，并为了追求原始速度而避免构建完整的 crate 图及其他高开销依赖项，因此它并不总是拥有足够的全局信息，以 `rustc` 的方式解析这些特征。弥合这一差距可能需要从周围的 crate 导入更多上下文，例如跨模块跟踪特征定义和关联类型，使提取器能够以与编译器相当的保真度重建签名。**5.5 验证评估**

第 5.4 小节中的提取实验表明，REM2.0 能够快速执行重构，并广泛支持各种语言特性。本小节将评估构建在该流水线之上的验证阶段。我们的目标是了解，对于当前 `CHARON/AENEAS` 片段中的代码，验证器能够多么频繁地证明原始函数与提取出的函数之间的等价性，这会带来多大的性能开销，以及实践中会出现哪些失败或限制。我们首先概述验证目标和基准测试设置，然后展示定量结果与性能测量，最后简要考察具有代表性的案例研究和失败模式。

5.5.1 验证目标与设置

这项验证实验旨在回答**RQ4**：对于支持子集中的代码，我们能否切实可行地验证原函数与提取出的函数之间的等价性？在 REM2.0 中，验证器是一种可选的高保证模式：开发者可以使用快速的提取 + 修复流水线进行重构，然后针对 AENEAS 能够支持的 crate 中的关键函数，有选择地调用验证。因此，我们的评估将考察：(i) 对于实际示例，该流水线能够成功生成等价性证明的频率；以及 (ii) 这会带来多少延迟开销

为确保不超出AENEAS当前的能力范围，我们整理了一个包含 20 个案例的基准测试套件（见第5.2.4小节）。前十个案例刻意选取了较简单的示例，来自内部针对提取方法的 RA 测试（表7中的 EX）。其余十个是“简单的真实世界”函数，取自与提取基准测试相同或相关的代码仓库，并经过人工筛选，以排除验证工具目前尚无法处理的特征（表7中的 RW）。对于每项基准测试，REM2.0 都会运行提取，然后将两个版本都交给验证工具链。CHARON 首先将相关 crate 转换为 LLBC，AENEAS随后生成 Coq 定义，最后由 REM2.0 调用证明生成器，在 Coq 中生成并检查一个等价引理。我们记录验证是否已尝试以及是否成功，并将总延迟分解为 LLBC 转换、Coq 翻译和证明构造与检查所用的时间，如表7所示。所有实验使用相同的机器和工具链版本，并为每个案例设置固定的 10 秒超时。我们将超时和 AENEAS 转换失败计为已尝试但未成功的证明，而无法转换为 LLBC 的情况则既不计为已尝试，也不计为成功。

5.5.2 验证结果与性能

表7 汇总了我们二十项基准测试的验证结果和耗时。所有十个基本案例（EX）都成功生成了等价性证明，端到端耗时约为 1.7–2.2 秒。大部分时间都花在证明检查上（底层无论如何都会调用 coqc 处理 3 个独立文件），而 LLBC 转换每次运行只耗费几百毫秒。如果此前运行过 `charon cargo`，我们会发现，大多数中间结果都已缓存在target 目录中，此时生成 LLBC 所需时间约为原来的 66%。

对于十项真实场景基准测试（RW），情况类似，但规模有所扩大。成功的案例耗时约为 3.8–4.4 秒，不过总耗时的分配要均衡得多。我们还可以看到，缓存的重要性要高得多：全新进行 LLBC 转换需要约 9–15 秒，而复用缓存的 LLBC 可将耗时缩短至约 1.6–1.9 秒。其中很大一部分时间花在依赖项上，这些依赖项只需编译一次（通过 charon cargo 或 cargo）。这意味着，只要某个 crate 已经转换过一次（甚至只是编译过一次），之后对同一 crate 进行验证通常只需几秒钟。总体而言，结果表明，对于支持的语言子集中的代码，REM2.0 能够常规地获得机器验证的等价证明，但其代价比单独进行提取高出一到两个数量级。

为确保稳健性，我们还在刻意进行错误 重构的情况下重新运行了验证流水线。对于每个基准测试，我们手动修改提取出的函数或其调用点，以引入轻微的行为变化，同时保持 Rust 代码类型正确。在原始验证尝试成功的所有情况下（即不包括本来就无法通过验证的案例 #20），修改后的版本都会导致等价性证明被拒绝：Coq 无法证明生成的引理，验证器也报告失败。这进一步支持了我们的主张：在其支持的语言子集中，该流水线对真实的语义差异十分敏感。

5.5.3 案例研究

为了进一步说明等价性检查器的结果意味着什么，我们简要考察两个具有代表性的案例：一个合成示例和一个现实世界函数。

#	Type	大小（代码行数）		时间（毫秒）				
		CLR	CLE	LLBC FRESH	LLBC CACHE	Coq 转换	证明构造与验证	总计
1	EX	4	4	293.04	198.4	308.25	1361.46	1868.11
2	EX	5	3	318.51	211.99	324.04	1638.93	2174.96
3	EX	5	3	275	175.96	295.95	1493.59	1965.5
4	EX	7	3	333.63	236.36	331.67	1314.47	1882.5
5	EX	5	3	289.29	176.14	326.38	1274.78	1777.3
6	EX	7	4	289.88	194.97	316.42	1268.78	1780.17
7	EX	3	4	272.51	170.92	294.54	1267.48	1732.94
8	EX	7	7	308.61	211.39	321.55	1216.28	1749.22
9	EX	8	6	269.91	192.37	325.98	1298.64	1816.99
10	EX	4	3	349.53	242.97	327.96	1363.15	1934.08
11	RW	4	10	15154.38	1666.14	1260.24	1383.22	4309.60
12	RW	11	4	9406.05	1671.70	1194.65	1340.72	4207.08
13	RW	14	5	9288.67	1763.12	1088.14	1168.97	4020.23
14	RW	5	44	9202.97	1746.28	1228.37	1313.29	4287.94
15	RW	24	12	9506.05	1618.23	1042.66	1202.97	3863.86
16	RW	6	7	9535.96	1748.76	1050.50	1185.17	3984.43
17	RW	13	22	9836.45	1650.95	1194.40	1514.83	4360.18
18	RW	29	7	10579.37	1710.06	1086.26	1200.34	3996.67
19	RW	26	11	9318.42	1680.40	1316.54	1328.94	4325.88
20	RW	17	5	9337.03	1880.03	1166.90	0.00	N/A

表 7：等价性结果与耗时 总时间取 FRESH 和 CACHE 中较短者

这些 EX 基准测试规模虽小，但在结构上并不简单：其中一些包含循环、条件语句以及对可变累加器的更新，反映了开发者通常会提取为辅助函数的代码类型。对于这些案例，REM2.0 成功生成了 Coq 开发，其中原始函数和提取函数通过自动生成的等价引理建立了关联。这类引理及其证明示例可见附录 32 中的清单 I）。

这些 RW 基准测试表明，同样的方法也适用于从实际项目中选取的短函数，而不仅仅是手工编写的示例。提取后，这些函数有时会增加额外参数（这些参数来自移出的局部变量），但除此之外仍保留原有结构；转换完成后，Coq 仍能在几百到一千毫秒内检查生成的等价性证明。证明的形式完全相同，但如代码清单 33 （附录 I）所示，Coq 中底层的数据类型要复杂得多。

总的来说，这些示例表明，等价性检查流程能够为非平凡的重构生成证明；相同的底层证明结构可以轻松适用于从极其简单（甚至不带单元类型）的函数定义，到复杂得多的数据类型和模块。

5.5.4 验证器的失败模式

尽管总体结果令人满意，但少数基准测试凸显了当前验证流水线以及 CHARON/AENEAS 的一些限制。

最明显的失败案例是表 7中的案例 #20，这是一个计算总退避延迟的“现实世界”示例。原始（Rust）函数维护了一个计数器 `attempts`、一个可变的 `delay`,

以及一个运行中 计，位于 w _ 循环中。然而，转换之后，AENEAS生成了一个递归的 Coq Fixpoint，其 通过一个 attempts 参数来表达。coqc随后报 推断 *fix* 的递减参数”。直观上，我们知道 Rust 代码会终止，因为循环计数器会递增，并与固定上限进行比较；但 AENEAS 为该函数生成递归定义的方式，使得这种递减关系在语法上并未清晰地呈现给 Coq 的终止性检查器。因此，这个基准测试对应的整个 Coq 开发都无法通过编译。

更广泛地说，在流水线的早期阶段，我们不得不排除若干潜在的基准测试，因为它们依赖于AENEAS尚不支持的语言特性。这些特性包括某些核心内建函数、派生的 相等比较/ PartialEq 实现， 及 es! 之类的宏展开。出现此类结构时，转换步骤会 Coq 代码之前失败，因此无法尝试证明。在当前的实验设计中，我们设计真实场景基准测试时避开了这些结构，但这种筛选必然掩盖了开发者在不受限制的代码中可能遇到的一些阻碍。

短期内，最切实可行的缓解措施是更好地指导开发者，让他们了解在什么情况下可以期待验证成功。作为这项工作的后续计划，我们打算为 REM2.0 增加一个轻量级的“AENEAS 兼容性”检查器，以便快速测试给定函数是否属于当前已验证的代码片段，并报告明显的阻碍因素（不受支持的内建函数、特征或控制流模式）。撰写本文时，该检查器尚不可用，但计划在本项目完成后不久推出。

5.6 讨论与有效性威胁5.6.1 讨论

提取实验表明，将 REM2.0 集成到 RA 的基础设施中，实现了第 3 节提出的主要设计目标。在原有的 REM 基准测试中，它严格优于原型——涵盖了全部 39 个可用案例，且延迟低到开发者几乎无法察觉。在新增案例集中，它能够处理涉及异步/ 等待 、常量函数 、NLCF、泛型和高阶 trait 约束的大多数示例，但也暴露出一系列与 DTO 及某些更复杂的泛型约束相关的失败。 Q1 和 RQ2：REM2.0 至少具备与原工具相当的能力，并且能够重构的 Rust 特性更加丰富。

在性能方面（RQ3），两者的差异尤为显著。原有的 REM 流水线提取时间为数百至数千毫秒，这很可能还未计入 IntelliJ 执行的工作。通过在持久化守护进程中复用 RA 的分析结果，REM2.0 将原始提取耗时缩短至几毫秒，并将用户感知到的延迟降至远低于四分之一秒。

验证结果提供了一个新的视角（RQ4）。对于属于 CHARON 和 AENEAS 支持子集的代码，验证器能够生成原始函数与提取出的函数之间的端到端等价性证明。这一过程比提取慢几个数量级，并且仅限于 Rust 的一个子集，但能够提供强得多的保证。实际上，这表明将会采用双层工作流：开发者默认使用 REM2.0 的快速提取，

并针对安全关键代码、棘手的生命周期模式，或需要遵守严格行为契约的应用程序接口，有选择地调用修复器或验证流水线。

最后，REM-Repairer 的使用情况证实了它作为工具链中一个针对性组件的作用。对于生命周期约束严格的情况，它仍然不可或缺，但大多数提取结果无需它也能成功。不过，如上所示，VSCode 带来的开销所引入的大量延迟远远超过了调用修复器的耗时，因此我们已将 Extract +Repair 流程设为 VSCode 扩展中的默认选项。

5.6.2 有效性威胁

与任何实证研究一样，我们的评估也面临若干有效性威胁。 y:

内部效度。 RQ3 和 RQ4 的时间测量会受到系统噪声、缓存以及 RA 和操作系统预热行为的影响。尽管我们尽力使用受控环境和一致的配置，但各次运行仍可能略有差异，因此这些数值应被视为相对趋势，而非精确的约束。

外部效度。 尽管我们的基准测试取自实际项目，但其范围必然有限。原始 REM 测试套件记录了 Rust 代码的一个特定快照，而新语料库则聚焦于大型且热门的 GitHub 代码仓库，并包含经过人工筛选的提取位置。这使评估偏向更“主流”的代码，因此无法涵盖嵌入式领域无 std crate 或高度专业化的库。

_

构念效度。 我们将编译成功，以及在适用情况下成功的 CHARON/AENEAS 证明，作为重构正确性（RQ1、RQ2、RQ4）的代理指标。实际上，我们只能对每个示例有粗略的了解，主要依靠 Rust 编译器和（在可能时）验证流水线来发现问题。虽然类型系统和借用检查器能够捕获许多错误，但细微的行为变化（例如性能、日志记录和并发效应）仍可能被忽略。

工具生态系统稳定性。 REM2.0 依赖 RA 的内部 API 以及外部验证工具（`CHARON`、`AENEAS`），而这些工具仍在积极开发中[16]。未来的变更可能会影响其行为、性能或支持的特征，因此我们的结果反映的是该工具栈演进过程中的一个特定阶段。

5.7 总结

本节围绕开篇提出的四个研究问题，对 REM2.0 进行了评估。提取实验表明，REM2.0 达到了原始 REM 原型的能力，并在此基础上有所扩展（RQ1、RQ2）；新架构则通过在持久化守护进程中复用 RA 的增量分析，实现了交互式 IDE 级延迟（RQ3）。对于目前支持的 CHARON/AENEAS 片段中的代码，验证流水线还能够进一步证明原始函数与提取函数之间的行为等价性；不过，这会带来显著更高的性能开销，同时也提供相应更强的保证（RQ4）。

[16]具体而言，RA 的内部 API 完全是为 RA 内部使用而设计的，而且经常发生变动。我们的工具链基于 0.0.262 版本构建，但随着每周发布，API 已升至 0.0.3XX 版本。尽管我们所依赖的主要组件尚未更新，但我们曾短暂尝试迁移，随后便很快放弃了。

总体而言，结果表明，REM2.0 为现代 Rust 提供了一种实用的提取函数重构方案，具备快速的默认路径，并针对自动修复以及可选的形式化验证提供了定向支持，以满足更高保证需求。

6 结论与未来工作

6.1 重新审视目标与研究问题

本项目旨在设计、实现并评估一套实用的 Rust 提取函数重构工具链，使其能够处理现代语言特性，并在可能的情况下提供强于“仍能编译”的正确性保证。具体而言，本研究致力于构建 REM 的新版本，使其能够与 `rust-analyzer` 集成，支持现代 Rust 语言特性，例如 `async` / `await` 和 `const fn`，并连接到用于等价性检查的验证流水线。

第 5 节围绕四个研究问题阐述了这一目标：

• **RQ1** – 新工具链在现有基准测试套件上能否达到或超越原始 REM 的表现?

• **RQ2** – 新的提取流水线在多大程度上能够处理现代 Rust 特性（`async`、`const fn`、泛高束、动态 trait 对象和非局部控制流）？

• **RQ3**– 当我们复用 rust-analyzer 作为持久运行的分析守护进程时，可以实现怎样的提取延迟？这对用户体验有何影响?

• **RQ4** – 对于受支持子集中的代码，我们能否借助 CHARON 和 AENEAS，切实验证原函数与提取出的函数之间的等价性?

评估结果给出了明确答案。对于 **RQ1**，REM2.0 在原有基准测试套件上与原版 REM 严格持平，并且还成功处理了此前失败的若干案例。对于**RQ2**，它能够处理大多数涉及异步、常量、NLCF、泛型和高阶 trait 约束的特性示例；剩余的薄弱之处主要集中在 DTO 和复杂的泛型约对**Q3**，新架构将提取延迟从数百或数千毫秒缩短至内部的几毫秒，在编辑器中观察到的延迟也远低于四分之一秒。对于 **RQ4**，验证流水线能够为 Rust 中一个不可忽略的代码片段构造端到端等价性证明，但性能开销显著更高，且支持的语言子集有限。

6.2 贡献总结

在此背景下，本文做出了若干技术和实证贡献。

6.2.1 基于 RA 的提取架构。

首要贡献是设计并实现了 REM2.0：一个构建在 RA 之上、可长期运行的守护进程。REM2.0 不再反复对整个 crate 调用 Rust 编译器，而是复用 RA 的分析功能，并提供 JSON-RPC 接口，可通过 VSCode 扩展或命令行客户端驱动。该扩展目前已在 VSCode Marketplace 上架。

6.2.2 功能丰富的提取引擎。

其次，项目开发了一种提取引擎，支持的 Rust 特性远多于原始 REM 原型。它能够正确处理异步/等待 、常量函数 、高阶 trait 约束、带有多个 trait 约束的泛型，以及 NLCF，并使用 [illegible] 统一处理返回、跳出 和继续。它在 DTO 密集型案例中仍表现不佳；[illegible] 功能类别，也是 [illegible] 明确方 [illegible]

6.2.3 生命周期与类型签名修复器。

第三，这项工作改进并重新利用 REM-Repairer，将其作为后处理步骤：当原始提取导致借用检查器错误时，它会自动调整提取出的函数签名和生命周期注解。实证结果表明，修复器在与原始评估相同类型的情形下必不可少——也就是生命周期约束严格的情形；因此，它仍是整个工具链的关键组成部分，而非单纯的便利功能。

6.2.4 与验证工具集成。

第四，我们制作了一个验证流水线原型，将 REM2.0 接入AENEAS。对于支持的语言子集中的函数，该流水线能够将原始代码和提取的代码转换为 Coq 定义，并构造证明，以证实两个版本在行为上等价。这为特定情形提供了一条从常规重构迈向机器检验的正确性保证的路径，同时也延续了构造即正确编程的宏大愿景 [32]。

6.2.5 对真实 Rust 项目的实证评估。

最后，该项目基于三个基准测试语料库开展了一项大规模实证研究。这些实验表明，该架构能够扩展到真实工作负载，并量化了延迟方面的改进，同时刻画了 REM2.0 的优势、直接改进之处和失败模式。

6.3 限制与开放挑战

尽管取得了这些成果，REM2.0 仍有明显的限制，划定了当前可以依赖它的范围，也指出了仍需进一步开展工作的领域。

在语言特性方面，DTO 和复杂的泛型约束是当前最大的薄弱环节。提取器无法还原精确函数签名时，就会出现若干失败情况，导致回退到未指定类型或丢弃约束。这与第二个限制密切相关：REM2.0 主要基于 RA 的 crate 图，使用经过修改的单文件视图进行操作，因此难以发现定义在未被我们明确跟踪和导入的模块中的约束。

验证流水线同样受到 AENEAS 所支持子集的限制。许多现实世界中的函数会使用超出该子集范围的特征，即使可以进行验证，其成本也显著高于提取。因此，验证器目前还不能不加区别地应用于任意重构，最好将其视为一种可选的高保证模式。

最后，评估关注的是正确性和性能，而不是用户体验。我们尚未开展研究来评估开发者实际采用该工具的方式、调用验证的频率，或他们对结果的信任程度。理解这些人为因素是一项重要但不在本研究范围内的挑战。

6.4 未来方向

未来工作的几个方向直接源于现有限制，以及我们对可证明正确的重构的更广阔愿景。

第一个方向是与 *RA* 进行更深入的集成。借助更多全局 crate 图信息，REM2.0 应能提取更丰富的 trait 和 where 子句信息，从而弥补我们在 DTO 和复杂泛型约束方面发现的许多不足。这可能需要扩展内部 API，或向上游贡献改动。尽管由于依赖 cargo check，这不太可能实现，但将修复器集成到 RA 中仍是本项目的长期工程目标。

第二个方向是扩展等价片段。随着 CHARON 和 AENEAS 不断改进，我们有望支持 Rust 的更大子集——内部可变性是一项值得期待的重要特性。同时，更深入地理解 LLBC（以及可能与 Polonius 借用检查器集成），有助于大幅提升我们无需进行函数式转换、即可直接推理原始 Rust 代码的范围。

第三，更智能的修复与分析启发式方法仍有改进空间。REM-Repairer 可以利用已收集的编译器反馈推断缺失的约束或函数注解，也可以通过学习以往成功修复的模式来升级。能够预测某个提取结果是否可能通过验证的静态检查，可以帮助开发者判断是否值得付出验证成本。

最后，复杂的所有权情况仍是一个重要的未解难题。开发者经常通过有选择地引入克隆、传递单个字段而非整个结构体，或使用内部可变性，来绕过借用检[illegible]拒绝。这些借用检查问题正是 Polonius 项目旨在解决的。目前，REM2.0 将这类情况视为无法处理的失败，因为我们依赖的 Rust 编译器无法为我们提供帮助。未来工作的一个自然方向是识别并应用一小套保持语义不变的“借用修复”模式，使更多提取结果能够自动调整为满足借用检查器要求的形式，同时不改变预期行为。此后，一个可能的改进方向是使用 Polonius 借用检查器，或深入利用 LLBC 语义，进一步改进原始提取结果。

6.5 更广阔的展望

退一步来看，本文位于重构工具、现代语言服务器和形式化验证的交汇处。对于安全的自动化重构而言，Rust 是一个要求极高的测试平台：其所有权和生命周期系统使整类错误都无法表达，但正如我们和许多其他人

所发现的，这些保证使天真的提取变得不可能，也让出于好意的提取变得异常困难。

REM2.0 表明，基于 RA 的设计能够实现提取函数重构，既足够快速，适合日常使用，又足够灵活，能够处理范围广泛的现代 Rust 语言特性。新增修复阶段和验证流水线表明，正确性的依据不必局限于编译；我们还可以有选择地获取经过机器检查的证据，证明重构能够保留程序行为。

如果这类工具变得普及，它们可能会改变开发者对 Rust 代码大规模演进的看法：许多重构仍将保持轻量、交互式，并由集成开发环境驱动；而关键的特性变更则可以升级为经过形式化论证的变换，同时对开发者工作流程的干扰降至最低。更广泛地说，这项工作展示了一条打造重构工具的路径：将正确性视为首要关注点，而非事后补救。在这个意义上，REM2.0 不只是一个更快的 Rust 提取函数命令，也是迈向新一代重构工具的早期一步；这类工具将验证融入日常软件工程实践。

A 附录 1：按值和按引用传递

本附录提供三个简短且自成一体的 Rust“朴素”函数提升示例。在每个示例中，我们都会定义一个正在重构的函数，并展示将参数按值传递和按引用传递的情况。为此，我们设想有一个自动化重构工具，它最初会将一段内联代码块移入一个新函数。编译错误（如果有）将为后续的所有权分析提供依据，由此推断每个变量应如何传递：按值传递（`T`）、按共享引用传递（`&T`）或按可变引用传递（`&mut T`）。

A.1 示例 1：按值传递

原始代码（内联）

```
fn main() {
    let x = 42;
    let y = x * 2;
    println!("y is {}", y);
}
```

代码清单 6：按值传递：原始代码

朴素提升后的代码（修复前）

假设某个自动化重构工具决定将乘法逻辑“提升”到一个新函数中：

```
函数提升_块(x: 32 位整数) {
```

```
2 设 y = x * 2;3       println!
("yis{}",y);4  }56
fnmain(){7 设 x = 42;8       提
升的_代码块(x);9  }
```

列表 7：按值传递：Naïve 提取

在此示例中，不会发生错误，因为调用 提升的代码块(x _ 在 x 中使用它 主函数，因此（传递所有权）是安全的。自动修复分析会检测到 x 在 主函数 中不再需要，因此判断可以按值传递。

A.2 示例 2：按共享引用传递

原始代码（内联）

```
1 fnmain(){ 2 let 文本 =String::from("Hello world!"); 3 let
length =文本.len(); 4 5 //仅从‘文本’读取的某个内联代码块： 6
println!("First word is: {}", get_first_word(&文本)); 7 println!("总长
度为： {}", length); 8 } 9 10 fn get_first_word(s: &str) ->&str { 11
s.split_whitespace().next().unwrap_or("") 12 }
```

列表 8：按共享引用传递：原始代码

朴素的提升代码（修复前）

假设重构工具决定将 `get first word` （或其周围的一些逻 _ _ 新的辅助函数中：

```
1 fnmain(){ 2 let text =String::from("你好，世界！"); 3 let
length =text.len(); 4 5 //只读取‘text’的内联代码块： 6 println!("首个
单词是： {}", get_first_word(&text)); 7 println!("总长度是： {}",
length); 8 } 9 10 fn get_first_word(s: &str) ->&str {
```

```
    s.split_whitespace().next().unwrap_or("")
}
```

代码清单 9：按共享引用传递：天真提取

如果函数 打印首个单词块 按值传递文本，那么这个版本 文本 被 到 打印首个单词 我们之后就无法 而此处是在 主函数中。 器会报错，指 本 已被移动，因此下面这一行将无 println!("总长度是：", length); 如果我们还需要文本 来完成其他操作。

自动修复结果

```
1 fn print_first_word_block(text: &String) { 2 // 仅从‘text’读取，因此只需要
共享引用。 3 println!("第一个单词是：{}", get_first_word(text)); 4 } 5 6 fn
main() { 7 let text =String::from("Hello world!"); 8 let length =text.len(); 9 10
print_first_word_block(&text); // 传递共享引用 11 println!("总长度为：{}",
length); 12 }
```

代码清单 10：按共享引用传递：自动修复

发生了什么?

- **只读用法**：函数 打印首个单词块 只 首个单词）。
- **共享引用**：由于该字符串之后必须在 主函数 中保持有效，所有权分析得 按共享引用传递（`&String`）才是正确的。

A.3 示例 3：按可变引用传递

原始代码（内联）

```
fnmain(){    letmut值 =vec![1, 2, 3];  //会修
改‘值’的内联代码块：     values.push(4);
println!("值现在是：{:?}",values);  }
```

清单 11：按可变引用传递：原始代码

天真的提升代码（修复前）

将 push 操作提升到一个新函数中，可以写成这样：

```
1 fn push_block(values: Vec<i32>) { 2 let mut v =values; 3 v.push(4); 4 println!("
Values are now: {:?}", v); 5 } 6 7 fn main() { 8 let mut values =vec![1, 2, 3]; 9
push_block(values); 10 println!("Final values: {:?}", values); // 错误：‘values’已被移
动 11 }
```

代码清单 12：通过可变引用传递：天真的提取

这里出现了一个与示例 2 类似的编译错误：一旦 `values` 按值传递[illegible]权就会转移，我们便无法在 `values` 中再次使用它。可是在这种情[illegible]我们确实需[illegible]改 主函数中的 `values`，并且希望 主函数 能够看[illegible]的向量。[illegible]

自动修复结果

```
1 fn push_block(values: &mut Vec<i32>) { 2 values.push(4); 3 println!("Values are
now: {:?}", values); 4 } 5 6 fn main() { 7 let mut values =vec![1, 2, 3]; 8 push_block(&
mut values); 9 println!("Final values: {:?}", values); // 现在 values 仍在作用域内，且已更新
10 }
```

列表 13：通过可变引用传递：自动修复

这是怎么回事?

- **需要可变性**：代码需要修改原始向量。
- **可变引用**：通过传递 `&mut Vec<i32>`[illegible]修改 值，而且 主函数 仍[illegible]的所有权，供之[illegible]用。[illegible]

B 附录 2：每个代码仓库的详细信息

排名	项目代码仓库	星标	复刻	未解决问题
2	denoland/deno	105077	5768	2384
3	rustdesk/rustdesk	102158	14993	61
4	tauri-apps/tauri	98717	3155	1161
5	unionlabs/union	74500	3835	155
6	astral-sh/uv	72353	2212	2136
7	zed-industries/zed	69398	5885	3054
8	FuelLabs/sway	62106	5428	862
9	Alacritty 终端模拟器	60977	3227	323
10	Rustlings	60569	10978	46
11	Fuel 核心	57463	2856	149
12	ripgrep	57159	2304	85
13	Magisk	56856	15853	36
14	bat	55755	1409	318
15	Meilisearch	54438	2254	221
16	ChatGPT	54214	6192	856
17	Awesome Rust	53670	3064	10
18	Starship	52113	2284	762
19	Vaultwarden	50694	2374	17
20	OpenAI Codex	50131	6215	1164
21	Typst	47876	1305	1007

表8：搜索 20 个代码仓库以查找真实案例的简要总结。#1 是 Rust 项目本身。

#	名称	URL
1	denoland/deno	https://github.com/denoland/deno/commit/104514b06dfbbb899235fb1147aa1fa62477a6d5
2	FuelLabs/fuel-core	https://github.com/FuelLabs/fuel-core/commit/ed92738280e068475682e220b93921bbbfebfdd7
3	Meilisearch/Meilisearch	https://github.com/meilisearch/meilisearch/commit/31142b36635ba062184b7d42f2e86021b11e911c
4	BurntSushi/ripgrep	https://github.com/BurntSushi/ripgrep/commit/7534d5144f5c5e89d67c791e663cdc0bee069b0e
5	starship/starship	https://github.com/starship/starship/commit/9093891acbe2c86b1615c37386dadbb0cc632199
6	FuelLabs/sway	https://github.com/FuelLabs/sway/commit/cdfc56c700ab32eefb84f75ed3048bd2a0ea22d2
7	tauri-apps/tauri	https://github.com/tauri-apps/tauri/commit/658e5f5d1dc1bd970ae572a42447448d064a7fee
8	unionlabs/union	https://github.com/unionlabs/union/commit/9b01ce6d75b84f049c1ede75750d33ce26ae73d9
9	astral-sh/uv	https://github.com/astral-sh/uv/commit/9af989e30c86391767830aa588c32fcb7c9a76b5
10	dani-garcia/vaultwarden	https://github.com/dani-garcia/vaultwarden/commit/952992c85bdf3147ddb8e663538e29c748b71f94
11	zed-industries/zed	https://github.com/zed-industries/zed/commit/d00ad02b05981c79942b3bdff957e2384e68b89f

表9：评估中各代码仓库的详细信息，包括我们找到示例或进行评估所依据的具体提交哈希

C 附录 3：GitOxide 示例详解

以下差异展示了如何将“if add path”代码块重构为新的辅助方法“提取包含路径”。

```
1 - ifadd_path{ 2 - ifletSome(body) =target_config.sections.get(&id){ 3 - letpaths =
body.values(&Key::from("path")); 4 - letpaths =paths.iter().map(|path|
values::Path::from(path.clone())) ;5- include_paths.extend(paths);6- }7- } 8 //为简洁起见，省略了部分
细节9 + ifadd_path{10 + extract_include_path(target_config,&mutinclude_paths,id)
11 + }
```

下面是完整的新方法：1 fn extract_include_path<’a>(2 target_config: &mut File<’a>, 3 include_paths: &mut Vec<values::Path<’a>>, 4 id: 节标识符 5) { 6 if let Some(body) = target_config.sections.get(&id) { 7 let paths =函数体.values(&Key::from("path")); 8 let paths =paths.iter().map(|path|values::Path::from(克隆(path))); 9 include_paths.extend(paths); 10 } 11 }

清单 14：完整的 gitoxide 方法

D 附录 4：来自H.3

```
// 此输出格式旨在用于人
// 并且可能在不另行通知的情况下发生变化。Knock
    3 fn 主函数()->(){ 4 let mut _0:(); 5 let _1: std::vec::Vec<
i32>; 6 let mut _2:std::boxed::Box<[i32]>; 7 let mut _3:无
符号整数类型; 8 let mut _4:无符号整数类型; 9 let mut _5:
*mutu8; 10 let mut _6:std::boxed::Box<[32 位整数; 3]>; 11
let _8:&32 位整数; 12 let mut _9:&std::vec::Vec<i32>; 13 l
et _10:(); 14 let mut _11:std::fmt::Arguments<'_>; 15 let _12:
&[&str; 2]; 16 let _13:&[core::fmt::rt::Argument<'_>; 1];
17 let _14: [core::fmt::rt::Argument<'_>; 1]; 18 let mut _15:
core::fmt::rt::Argument<'_>; 19 let _16:&&32 位整数; 20 let
mut _17:*const [32 位整数; 3]; 21 let mut _18:*const(); 22
let mut _19:无符号整数类型; 23 let mut _20:无符号整数类型;
24 let mut _21:无符号整数类型; 25 let mut _22:无符号整数类
型; 26 let mut _23:布尔值; 27 作用域 1{ 28 debugv => _1;
29 let _7:&32 位整数; 30 作用域2{ 31 debugx => _7; 32 } 33 }
34 35 bb0: {36         _3 = 大小查询 ([32 位整数 ; 3]);
37         _4 =对齐查询([32 位整数; 3]);38         _5 =
alloc::alloc::exchange_malloc(move _3
```

仅消费者
把自己排除在外。

```
, 移动 _4)-> [返回:bb1,unwind继续];
39     }4041     bb1: {42         _6 =浅初始化
Box(move _5, [32 位整数; 3]);43         _17 =复制(((_6.
0:标准库唯一指针<[32 位整数;
3]>).0: 标准库非空指针<[32 位整数; 3]>).0: *
常量 [32 位整数; 3]);44         _18 =复制 _17为
*const ()（指针到指针转换）; 45         _19 =复制 _18
为无符号整数类型（类型重解释）; 46         _20 =对
齐查询([32 位整数; 3]);47         _21 =减法(copy _20,
常量 1_无符号整数类型);48         _22 =按位与(copy
_19, copy _21);49         _23 =相等比较(copy _22, 常
量 0_无符号整数类型);
```

50 断言(copy _23, "指针解引用未对齐：地址必须是 {} 的倍数，但实际为 {}", copy _20, copy _19) -> [成功：bb10，unwind 不可达]; 51 } 52 53 bb2: {54 _9 =&_1;55 _8 = <Vec<i32> 作为 Index<无符号整数类型>>::索引(move _9, 常量 0_无符号整数类型) -> [返回：bb3，unwind：bb8]; 56 } 57 58 bb3: {59 _7 =复制 _8;60 _12 =常量 main::promoted[0];61 _16 =&_7;62 _15 =核心库格式化运行时参数::<'_>::new_display::<&i32>(copy _16) -> [返回：bb4，unwind：bb8]; 63 } 64 65 bb4: {66 _14 = [move _15];67 _13 = &_14;68 _11 = Arguments::<'_>::new_v1::<2, 1>(copy _12, copy _13) -> [返回：bb5，unwind：bb8]; 69 } 70 71 bb5: {72 _10 = _打印(move _11) -> [返回：bb6，unwind：bb8]; 73 } 74 75 bb6: { 76 drop(_1) -> [返回：bb7，unwind 继续]; 77 } 78 79 bb7: { 80 返回; 81 } 82 83 bb8（清理）：{ 84 drop(_1) -> [返回：bb9，unwind 终止(cleanup)]; 85 } 86 87 bb9（清理）：{ 88 恢复; 89 } 90 91 bb10: { 92 (*_17) = [常量 1_32 位整数, 常量 2_32 位整数, 常量 3_i32];93 _2 =move _6 转换为 std::boxed::Box<[i32]> (PointerCoercion(Unsize, Implicit));94 _1 =slice::<impl [i32]>::into_vec::<std::alloc::Global>(move _2) -> [返回：bb2, unwind 继续]; 95 } 96 }

```
97 98 CONST 主函数::promoted[0]:&[&str;
 2] ={ 99 let mut _0:&[&str; 2]; 100 let mut _1: [
&str; 2]; 101 102 bb0: {103 _1 = [
const"",const"\n"];104 _0 =&_1; 105
return; 106 } 107 }
```

清单 15：完整的 MIR，摘自 H.3

E 附录 5：绘制的提取时间数据

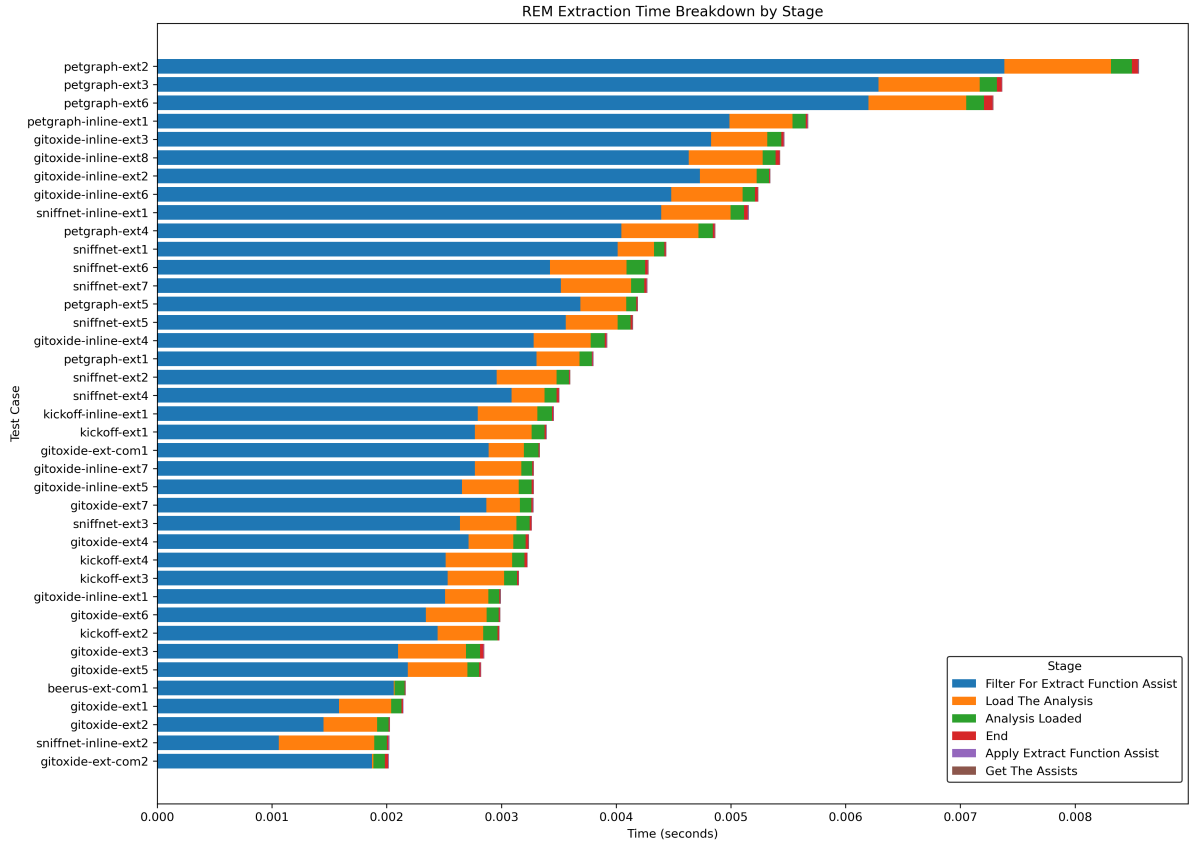

图 11：每个测试用例按流水线阶段划分的执行时间明细。时间以秒为单位（根据纳秒计时数据得出）。

Case	加载 分析	分析 已加载	获取 辅助操作	筛选条件 EF Assist	应用 EF Assist	剩余	总计
beerus-ext-com1	9285	86226	2120	2060529	2476	5371	2166007
gitoxide-ext-com1	307613	128739	2785	2887546	2574	7746	3337003
gitoxide-ext-com2	10238	100897	2468	1872872	2692	28872	2018039
gitoxide-ext1	454708	90926	2210	1583716	3171	13176	2147907
gitoxide-ext2	468576	99266	2948	1449459	3634	6355	2030238
gitoxide-ext3	593338	121557	2593	2098630	4041	30428	2850587
gitoxide-ext4	387921	109024	2798	2715010	3668	21512	3239933
gitoxide-ext5	520525	101651	2508	2183887	2948	11685	2823204
gitoxide-ext6	529156	104946	2568	2342050	2513	10251	2991484
gitoxide-ext7	290983	101383	2307	2869735	5587	11670	3281665
gitoxide-inline-ext1	376787	96027	2954	2508828	3532	6906	2995034
gitoxide-inline-ext2	496737	105101	2427	4729361	3486	8189	5345301
gitoxide-inline-ext3	488481	122647	3248	4827718	4000	21855	5467949
gitoxide-inline-ext4	495756	121965	3684	3281795	5483	15325	3924008
gitoxide-inline-ext5	495051	112329	2474	2657133	2991	12808	3282786
gitoxide-inline-ext6	622550	107609	2873	4480736	3848	22912	5240528
gitoxide-inline-ext7	403207	96122	2356	2769977	2617	8734	3283013
kickoff-ext1	494546	112377	3619	2768148	2671	14533	3395894
kickoff-ext2	398752	124576	2850	2443569	2733	11091	2983571
kickoff-ext3	492629	110374	2586	2531962	2069	13516	3153136
kickoff-ext4	578567	108555	2736	2514134	3164	22551	3229707
kickoff-inline-ext1	520277	126225	2583	2793580	2920	11691	3457276
petgraph-ext1	374755	105993	2342	3306952	3347	9064	3802453
petgraph-ext2	929776	181024	3519	7384406	4936	53650	8557311
petgraph-ext3	881791	149205	2483	6287406	2447	42136	7365468
petgraph-ext4	671999	123294	3069	4046105	3188	16542	4864197
petgraph-ext5	400859	86750	2658	3688382	3107	9421	4191177
petgraph-ext6	853792	154358	3156	6198179	2902	75377	7287764
petgraph-inline-ext1	550082	114773	3004	4988262	3287	14740	5674148
sniffnet-ext1	318576	89782	3191	4012997	2925	11313	4438784
sniffnet-ext2	521024	108808	2324	2959298	2719	7431	3601604
sniffnet-ext3	491132	114474	2420	2639553	2501	14870	3264950
sniffnet-ext4	289206	104857	2458	3087829	3184	19270	3506804
sniffnet-ext5	451829	112413	2904	3560519	2571	16898	4147134
sniffnet-ext6	665962	163596	5027	3423886	2428	22809	4283708
sniffnet-ext7	610395	115393	3062	3518957	6090	19601	4273498
sniffnet-inline-ext1	606027	117642	2553	4392070	8793	30967	5158052
sniffnet-inline-ext2	830359	110345	3269	1060569	7246	13375	2025163

表10：各种提取结果的原始计时数据。时间单位均为纳秒

F 附录 6：（几乎）完整的 Coq 翻译

这里删去了一些 Coq 样板代码，使这些内容更易于理解。即便如此，也不难看出，哪怕是一个很小的示例，转换后的内容也可能变得多么庞大。

```
Definition bump
 {N : usize} (coremarkerCopyArrayI32NInst : core_marker_Copy (array i32 N))
 (arr : array i32 N) (k : usize) (mark : bool) :
 result ((array i32 N) * bool)    :=
 if k s< N then bump_loop0 k arr mark 0%usize else bump_loop1 arr mark 0%usize
.  (* *)  (* *)  (* *)  (* *)  (* *)  (* *)  (* *)  (* *)  (* *)
(* *)  (* *)  (* *)  (* *)  (* *)  (* *)  (* *)  (* *)  (* *)
(* *)  (* *)
```

代码清单 16：原始 Coq 翻译（删去部分样板代码）

```
Definition bump
{N : usize} (coremarkerCopyArrayI32NInst : core_marker_Copy (array i32 N))
(arr : array i32 N) (k : usize) (mark : bool) :
result ((array i32 N) * bool) :=
  if k s< N then (
  _ <- bump_loop0 coremarkerCopyArrayI32NInst arr k mark 0%usize;
  Ok (arr, mark)) else (
   _ <- bump_loop1 coremarkerCopyArrayI32NInst arr mark 0%usize;
   Ok (arr, mark)) 。

Definition bump_extracted
{N : usize} (coremarkerCopyArrayI32NInst : core_marker_Copy (array i32 N))
(arr : array i32 N) (i : usize) (mark : bool) : result unit :=
i1 <- array_index_usize arr i; if i1 s= 0%i32
then (_ <- i32_add i1 1%i32; _ <- array_index_mut_usize arr i; Ok tt)
else (_ <- i32_add i1 1%i32; _ <- array_index_mut_usize arr i; Ok tt) 。
```

清单 17：错误提取的 Coq 翻译，省略部分样板代码

G 附录 7：新增提取功能类别详述

本附录为第5.4.2节介绍的每种功能类别提供具体示例（尽管规模较小）。这些示例取自第5.2.3节列出的代码仓库，或是对我们在其中观察到的模式略作简化后的变体，旨在隔离考察提取器遇到特定结构时的行为。除异步函数、常量求值、泛型、动态 trait 对象和高阶 trait 约束外，我们还列举了若干非局部控制流的实例。尽管原有 REM 工具链已经能够处理某些非局部控制流重构，但新的提取引擎通过（现已成为标准的）`std::ops::ControlFlow` 变体而非自定义枚举来实现这些重构，因此我们会单独评估它们。总的来说，这些示例展示了原有 REM 无法支持、而新工具链专门为处理这些情况而设计的高级语言机制，使我们能够逐一检查它是否保留了各个案例中的语义、类型和编译有效性。

G.0.1 泛型（GEN）

此案例针对一个具有多个 trait 约束的泛型函数，验证这些约束是否会被原样复制到提取出的函数中。提取器不得将泛型参数单态化或省略，否则会限制函数的多态性，并很可能导致类型检查失败。此外，如果调用方函数的 trait 约束中只有一部分适用于被调用方（例如，被调用方只接收多个泛型参数中的一个），那么我们期望只复制适用的 trait 约束。

```
fn min_of<T: Ord + Copy>(a: T, b: T) -> T {
    // EXTRACT START
    if a < b { a } else { b }
    // EXTRACT END
}
```

清单 18：泛型示例：之前

```
fn min_of<T: Ord + Copy>(a: T, b: T) -> T {
    choose_min(a, b)
}

fn choose_min<T: Ord + Copy>(a: T, b: T) -> T
    {
    if a < b { a } else { b }
}
```

清单 19：泛型示例：之后

G.0.2 异步与等待（ASYNC）

此案例针对从异步函数内部提取代码，此时插必须保留在编译器生成的状态机[17]内。提取器必须正确保留异步上下文以及由结果类型返回类型所隐含的?，同时确保生命周期和借用不会跨越 await 挂起点而被果提取引擎生成的生命期不正确，我们期望 REM-Repairer 能够修正它们。

[17] `rustc` 会自动生成一个充当状态机的结构体，其中包含一个内部状态变量和一个 poll 方法。每个 await 点都会成为状态机可以暂停、返回 `Poll::Pending`，并由运行时稍后恢复执行的位置，同时不会丢

```
async fn fetch_and_tag(client: &Client, id: u64) -> Result<String, Error> {
    // EXTRACT START
    let body = client.get(id).await?;     // await inside block
    let tagged = format!("[#{}] {}", id, body);
    // EXTRACT END
    Ok(tagged)
}
```

清单 20：异步示例：之前

```
async fn fetch_and_tag(client: &Client, id: u64) -> Result<String, Error> {
    fetch_and_tag_inner(client, id).await
}

async fn fetch_and_tag_inner(client: &Client, id: u64) -> Result<String, Error> {
    let body = client.get(id).await?;
    let tagged = format!("[#{}] {}", id, body);
    Ok(tagged)
}
```

清单 21：异步示例：之后

G.0.3 常量声明（CONST）

此案例涵盖在常量函数内部提取代码 st 的常量检查规则，这类函数必须在编译时仍可求值。因此，提取引擎必须识别常量上下文，并在可能的情况下，将常量限定符传递给新函数，以维持这些保证。

```
// A const function that can be evaluated at compile time
const PREFIX: u32 = 10;
const fn compute_total(a: u32, b: u32) -> u32 {
    // EXTRACT START
    PREFIX + a + b
    // EXTRACT END
}
```

清单 22：CONST 示例：之前

```
// A const function that can be evaluated at compile time
const PREFIX: u32 = 10;
const fn compute_total(a: u32, b: u32) -> u32 {
    compute_total_inner(a, b)
}

// Extracted function still marked as const
const fn compute_total_inner(a: u32, b: u32) -> u32 {
    PREFIX + a + b
}
```

清单 23：CONST 示例：之后

G.0.4 非局部控制流（NLCF）

此案例涵盖了 NLCF 的所有方面，包括 retu 出 在 Ru 法。NLCF 对象无法跨越函数边界，因此提取引擎必须将控制流具体化为返回值，并使用 `std::ops::ControlFlow` 变体 调用方。如前所述，这与 REM 最初通过插入自定义枚举来概述控制流处理的做法不同，因为我们依赖提取引擎而非 REM-Controller 来提供这些信息。

```
fn sum_until_negative(xs: &[i32]) -> i32 {
    let mut sum = 0;
    for &x in xs {
        // EXTRACT START
        if x < 0 { break; }   // non-local ‘break‘
        sum += x;
        // EXTRACT END
    }
    sum
}
```

清单 24：NLCF 示例：之前

```
use std::ops::ControlFlow;
use std::ops::ControlFlow::{Break, Continue};
fn sum_until_negative(xs: &[i32]) -> i32 {
    let mut sum = 0;
    for &x in xs {
        match step_sum_cf(x) {
            Break(()) => break,
            Continue(delta) => sum += delta,
        } }
    sum
}
fn step_sum_cf(x: i32) -> ControlFlow<(), i32> {
    if x < 0 { Break(()) }
    else { Continue(x) }
}
```

清单 25：NLCF 示例：之后

G.0.5 高阶特征约束 (HRTB)

本案例考察从带有高阶 trait 约束（例如 `for<’a> Fn(&’a T) -> &’a T`）的函数中进行提取。这些约束必须被正确地复制到新的函数签名中，以确保生命周期正确性。同样，此处预计会调用 REM- Repairer，以便 a) 验证生命周期是否正确，和/或 b) 修复不正确的生命周期约束。

```
fn apply_str<F>(f: F) -> &’static str
where
    F: for<’a> Fn(&’a str) -> &’a str,
{
    // EXTRACT START
    f(s)
    // EXTRACT END
}
```

清单 26：HRTB 示例：之前

```
fn apply_str<F>(f: F) -> &’static str
where
    F: for<’a> Fn(&’a str) -> &’a str,
{
    call_higher(f)
}
fn call_higher<F>(f: F) -> &’static str
where
    F: for<’a> Fn(&’a str) -> &’a str,
{
    let s = "hello";
    f(s)
}
```

清单 27：HRTB 示例：之后

G.0.6 动态特征对象（DTO）

这里隔离出涉及 `&dyn Trait` 参数的提取引擎对动态分派和对象安全的保留情况。提取器必须保留特征对象类型，避免将其替换为泛型约束，否则会改变运行时行为。此类别还包括函数的参数实现某个 trait 的情况，例如 `fn ident(s: impl Into<字符串> -> ...`。

```
use std::fmt::Display;

fn show_twice(d: &dyn Display) -> String {
    // EXTRACT START
    format!("{} | {}", d, d)
    // EXTRACT END
}
```

清单 28：DTO 示例：之前

```
use std::fmt::Display;

fn show_twice(d: &dyn Display) -> String {
    show_twice_inner(d)
}
fn show_twice_inner(d: &dyn Display) ->
String {
        format!("{} | {}", d, d)
}
```

清单 29：DTO 示例：之后

H 附录 8：验证流水线的实现

本附录将深入探讨等价性证明器背后的所有实现细节，内容更加详尽。本文旨在配合源代码阅读，其中部分内容会与第4.5节重复。

REM2.0 的等价性检查设计遵循两项原则：(1) 验证不得要求开发者提供任何额外的注解或输入；(2) 验证必须足够快速，能够融入交互式 IDE 工作流。为此，流水线将原始程序 P 和重构后的程序 P' 转换为结构逐步增强的语义表示，最终得到一项观察等价的机器验证证明。

从总体上看，流水线由五个阶段组成：**提取**、**REM 修复**、**CHARON 转换**、**AENEAS 函数化**和**Coq 验证**。每个阶段都以前一阶段为基础，逐步揭示重构后代码的语义，最终得到 Coq 中纯粹的函数式模型。图12概述了整个流程。本节仅简要介绍提取和生命周期修复；完整内容请参见第3 节。

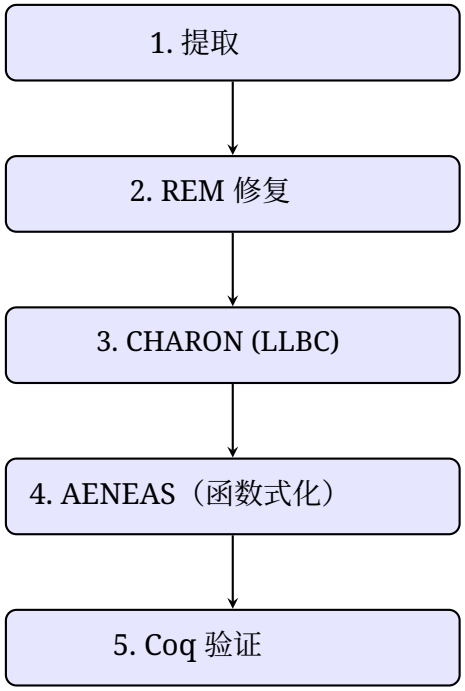

图 12：验证流水线概览：从源代码提取到 Coq 中观察等价性的证明。

H.1 提取

提取会将用户选定区域提取为一个新函数，并在原位置将选中内容替换为一次调用。正如第3.5节所述，此步骤由 Rust-Analyzer（RA）支持的引擎驱动，通过单文件增量分析完成处理。它会在内存中解析选中内容的抽象语法树和符号信息，并生成函数骨架以及调用点编辑。

提取结果以确定性编辑集（补丁）和供下游阶段使用的机器可读摘要形式返回。按照设计，提取器不会尝试最终确定生命周期或高级所有权模式——这些工作交由 REM 的修复阶段负责。这种分离使提取过程只需数毫秒且可预测，并且完全不受 crate 大小影响，同时确保后续验证拥有稳定的接口。

H.2 REM

REM 执行提取后的语义修复，尝试使提取的代码重新符合 Rust 的所有权和生命周期规则。尽管原始 REM 工具链的大部分工作已集成到提取过程中，但我们仍然高度依赖 REM 分析中最复杂的部分：生命周期具体化。我们引入、传播并

约束生命周期参数，以满足借用检查器的要求，并在必要时进行泛化。如前文所述，REM 将编译器视为预言机：它不断迭代修复，直到程序能够编译，然后尝试应用生命周期省略，以生成便于人类阅读的签名（在实际可行的情况下）以及尽可能贴近惯用 Rust 的函数体。因此，REM 的输出是修复后的程序 P'，随后将其传递给 CHARON，以开始消解等价性义务的过程。

H.3 CHARON

验证流水线的第一个（新）阶段，是将 Rust 转换为更低层、更适合验证的形式。这项工作由**CHARON**完成，它是一个将 Rust 转换为 LLBC（低级借用演算）的工具。CHARON 接受 Rust 源代码（更准确地说，是其编译器中间表示 MIR）[18]。CHARON 的主要任务是从 `rustc` 中提取复杂的语义信息，并生成包含这些信息的机器可读输出。此前，我们一直只处理／分析单个文件。虽然这让提取速度极快，但 CHARON 对单文件转换的支持充其量也十分有限[19]。

因此，在这一阶段，我们会在用户的临时目录（`std::env::temp()`）中构建两个“虚拟”crate。这[illegible]：

1. 在我们执行后续分析阶段期间，用户对源代码所做的任何更改都不会传递给我们（否则可能引入只写了一半的单词或其他会立即导致失败的问题）。2. 我们不需要锁定用户的代码文件，因此用户可以继续工作。3. 我们可以安全地封装虚拟 crate 的创建和删除，并使用自定义的`Drop` trait，使其生命周期仅与等价性引擎的运行时间相同。4. 我们可以将通过 JSON-RPC 桥接传递的数据限制为用户正在处理的文件的原始代码和提取的代码。随后，CHARON 会执行若干非平凡的转换：

[illegible]

- **结构化控制流**。 MIR 是一个控制流图，其中包含任意 `goto`；CHARON [illegible]其重构为结构化的 `if`、`match`和 `loop` 结构，也[illegible]以[illegible]`-ullb`[illegible]保留原始形式。

- **trait 和类型解析**。 显式记录 trait 约束的证明方式，规范化默认方法，并可将关联类型转换为显式参数（--remove-associated-types）。

- **生命周期处理**。 隐藏早绑定变量与晚绑定变量之间的区别，并将隐含约束显式化，从而简化对泛型函数的推理。
- **闭包和虚函数表**。 将它们表示为实现相关 Fn* trait 的普通结构体，使其可供后续验证步骤使用。
- **减少噪声**。例如，使用`--hide-marker-traits` 可移除内置 trait（`Sized`、`Send` 等），否则[illegible]容[illegible]输出显得杂乱。

[18]更准确地说，该工具本身接受 Rust 代码，但其初始步骤依赖于与 rustc 的复杂交互来获取 MIR[19]CHARON 可以只转换单个文件，但这需要依赖 rustc，因此该单个文件必须是自包含的，而我们无法保证这一点

最终得到的是一个经过清理、语义忠实的 LLBC 程序。与面向编译器的原始 MIR 不同，LLBC 专为验证而设计：它直接表示所有权和借用，明确呈现 trait 解析，并采用结构化的控制流。下方展示了一个示例程序及其部分 MIR。不过，CHARON 的输出并非为方便人类阅读而设计，即使是图13 中的微型程序也会生成超过 140,000 个字符的 LLBC。这种（极其冗长的）显式表示，使得后续能够将其转换为纯函数式代码以进行验证。

```
fn 主函数() {
    令 v = vec![1 , 2, 3];
    令 x = &v[0];
    println!("{}", x);
}
```

清单 30：Rust 源代码

```
bb2: {
    _9 =&_1;
    _8 = <Vec<i32>as Index<无符号整数类型>>::
         index(move _9, const 0_无符号整数类型);
}

bb3: {
    _7 =copy _8;
    _16 =&_7;
    _15 = 核心库格式化运行时参数::<'_>::
         新建_显示::<&i32>(复制 _16);
}
```

清单 31：MIR 摘录（rustc）

图 13：CHARON 作用示意图。这里仅展示了 Rust 和 MIR，而 CHARON 会将 MIR 转换为 LLBC。LLBC 的设计目标仅是便于机器读取，因此未予展示

H.4 AENEAS

CHARON 将程序转换为 LLBC 后，流水线的下一阶段由 **AENEAS** 负责。AENEAS 接收显式包含借用和生命周期信息的 LLBC，并将其转换为纯函数式形式，剥离底层内存操作，同时保留程序的语义。这一函数式化步骤使程序适合在 Coq 中进行后续推理：我们不再需要针对引用和借用进行推理，而可以针对值和函数进行推理。目前，AENEAS 支持转换为*Coq*、*Lean*、*HOL4* 和 F^*。

在实践中，AENEAS 充当了编译器级细节与证明级抽象之间的桥梁。CHARON 确保生命周期、加载操作和 trait 解析的所有复杂性都被明确呈现；随后，AENEAS 将这些内容封装为一种形式，使我们无需接触借用检查器或栈/堆，便可以陈述并证明正确性性质。在我们的流水线中，这正是 Rust 代码“停止作为 Rust”并转变为函数式程序的阶段，而其等价性可以在 Coq 中进行检查。

H.5 Coq

AENEAS 将 LLBC 转换为纯函数式形式，目标语言为 Coq。随后，流水线的最后阶段会将生成的两个 Coq 文件（P_{coq} 和 P'_{coq}）与所有辅助信息结合起来，以自动验证原始 Rust 程序与重构后的 Rust 程序之间的等价性。这一阶段的一个主要挑战是生成的 Coq 代码过于冗长

代码（如清单 10b）所示），因此我们实现了一个定制的 Coq 词法分析器和解析器，以适配 AENEAS 输出的格式。
与这些转换同步进行的是，等价性证明器的主要任务是生成`EquivCheck.v` 文件。该模块实例化原始定义和重构后的定义，并生成一个等价定理；该定理是对第 4.4 节所述义务的 Coq 编码。实际上，这意味着 Coq 项目的设置会自动检查这两个版本是否对所有 Input 产生相同的结果。证明义务的具体细节将在该节中介绍；这里的关键在于，生成的 Coq 项目是自包含的：编译完成后，无需用户进一步干预即可验证等价性。

什么是 Coq?

Coq 是一种基于依赖类型函数式语言构建的交互式定理证明器和证明助手。它允许开发者定义数学对象、程序和逻辑命题，并构造关于这些对象的机器验证的证明。每个证明都由 Coq 的小型可信内核进行验证，从而为正确性提供极高的保证。Coq 长期以来一直用于大规模验证项目，例如 CompCert（经过验证的 C 编译器）和形式化验证的操作系统，因此非常适合用于推理 Rust 程序的安全性和正确性。

为什么选择 Coq?

相较于 HOL4、Lean 或 F* 等替代方案，选择 Coq 主要有三个原因：

- **成熟的生态系统** 数十年的发展带来了强大的工具、丰富的库以及稳定的内核。

- **直接集成** AENEAS 已经以 Coq 为目标，而 HOL4 和 Lean 则需要大量新的后端才能实现兼容。
- **自动化** Coq 的策略语言和 SMT 集成能够自动完成许多常规证明，例如等价性检查。

Lean 和 HOL4 都提供了表达能力很强的语法，但代价是需要投入大量工程工作才能与 AENEAS [20] 集成。虽然 F* 所需的额外基础设施最少，但该平台还不够成熟，而且会重复实现现有功能。因此，Coq 在成熟度、自动化程度以及与我们流水线的直接兼容性方面，提供了最佳组合。

[20] 二者都需要在当前流水线之上增加更多额外的支持架构

I 附录 9：Extraction Evaluation 中的 EquivCheck.v 引理

```
1 (** EquivCheck.v 由 REM 验证自动生成 **) 2 Require
Import Primitives. 3 Import Primitives. 4 Require Import
Input. 5 Require Import Out. 6 Local Open Scope Primitives_作
用域. 7 8 Lemma foo_相等比较_check : 9 Input.foo  =Out.foo. 10
Proof. 11 intros. 12 unfold Input.foo. 13 unfold Out.foo. 14 自反
性. 15 Qed.
```

清单 32：一个简单验证案例中的 EqivCheck.v 文件

```
1 (** EquivCheck.v 由 REM 验证自动生成 **) 2 Require Import Primitives. 3 Import
Primitives. 4 Require Import BusinessLogic. 5 Require Import BusinessLogicRefactored. 6
Local Open Scope Primitives_作用域. 7 8 Lemma example5_config_normalise_config_equiv_
check : forall (raw : example5_config_RawConfig_t), 9 BusinessLogic.example5_config_
normalise_config raw  =BusinessLogicRefactored. example5_config_normalise_config raw.
10 Proof. 11 intros raw. 12 unfold BusinessLogic.example5_config_normalise_config. 13
unfold BusinessLogicRefactored.example5_config_normalise_config. 14 自反性. 15 Qed.
```

清单 33：一个复杂验证案例中的 EqivCheck.v 文件

引用

[1] Nicholas D. Matsakis 和 Felix S. Klock。《Rust 语言》。*Ada Lett.*，34(3):103–104，2014年10月。ISSN 1094-3641。doi: 10.1145/2692956.2663188。URL https://doi.org/10.1145/2692956.2663188。

[2] Sewen Thy、Andreea Costea、Kiran Gopinathan 和 Ilya Sergey。《Adventure of a Lifetime: Extract Method Refactoring for Rust》。*ACM*程序语言，7(OOPSLA2)，2023年10月。doi：10.1145/3622821。URLhttps://doi.org/10.1145/3622821。

[3] Sewen Thy。《Borrowing Without Sorrowing: Implementing Extract Method Refactoring for Rust》。2023年。URLhttps://verse-lab.github.io/papers/Sewen-Thy-Capstone.pdf。

[4] Yaroslav Golubev、Zarina Kurbatova、Eman Abdullah AlOmar、Timofey Bryksin 和 Mohamed Wiem Mkaouer。《一千零一个故事：软件重构的大规模调查》。发表于 第 *29* 届 *ACM* 欧洲软件工程会议与软件工程基础研讨会联合会议论文集，ESEC/FSE 2021，第 1303–1313 页，纽约，纽约州，美国，2021 年。美国计算机协会。ISBN 9781450385626。doi：10.1145/3468264.3473924。网址：https://doi.org/10.1145/3468264.3473924。

[5] Garming Sam、Nick Cameron 和 Alex Potanin。《Rust 程序的自动化重构》。发表于澳大拉西亚计算机科学周多会议论文集，ACSW '17，纽约，纽约州，美国，2017年。美国计算机协会。ISBN 9781450347686。doi：10.1145/3014812.3014826。网址：https://doi.org/10.1145/3014812.3014826。

[6] Alejandra Garrido 和 Jose Meseguer。Java 重构的形式化规约与验证。发表于*2006* 年第六届 *IEEE* 国际源代码分析与操作研讨会，第 165–174 页，2006 年。doi：10.1109/SCAM.2006.16。

[7] RustLang 团队。《Rust By Example》。https://doc.rust-lang.org/rust-by-example，无日期。访问日期：2025-04-15。

[8] Twain Byrnes、Yoshiki Takashima 和 Limin Jia。自动强制执行 Rust trait 属性。发表于《验证、模型检验与抽象解释：第 *25* 届国际会议，*VMCAI 2024*，英国伦敦，*2024* 年 *1* 月 *15—16* 日，论文集，第二部分》，第 210–223 页，柏林、海德堡，2024 年。施普林格出版社。ISBN 978- 3-031-50520-1。doi：10.1007/978-3-031-50521-8 10。网址：https://doi.org/10.1007/978-3-031-50521-8_10。

[9] Alex Potanin、James Noble、Dave Clarke 和 Robert Biddle。泛型 Java 的泛型所有权。载于*ACM* 面向对象编程、系统、语言与应用国际会议论文集（*OOPSLA 2006*），第 311–324 页。ACM，2006 年。doi：10.1145/1167473.1167500。

[10] Yoav Zibin、Alex Potanin、Paley Li、Mahmood Ali 和 Michael D. Ernst。泛型 Java 中的所有权与不可变性。载于*ACM* 面向对象编程、系统、语言与应用国际会议论文集（*OOPSLA 2010*），第 598–617 页。ACM，2010 年。doi：10.1145/1869459.1869509。

[11]Yoav Zibin、Alex Potanin、Mahmood Ali、Shay Artzi、Adam Kie˙zun 和 Michael D. Ernst。使用 Java 泛型实现对象和引用不可变性。载于第六届欧洲软件工程会议与 *ACM SIGSOFT* 软件工程基础研讨会联合会议（*ESEC/FSE 2007*），第 75–84 页。ACM，2007年。doi：10.1145/1287624.1287637。杰出论文奖。

[12] 维基百科贡献者。Rust（编程语言）。https://en.wikipedia.org/ wiki/Rust_ (programming_language)，2025年。访问日期：2025-11-15。[13] Wenzhang Yang、Linhai Song 和 Yinxing Xue。Rust-Lancet：自动修复所有权规则违规，同时保持行为。载于 2024 IEEE/ACM 第46届国际软件工程大会（ICSE），第 1034–1046 页，美国加利福尼亚州洛斯阿拉米托斯，2024年4月。IEEE计算机学会。网址：https://doi.ieeecomputersociety.org/。[14] Ralf Jung、Jacques-Henri Jourdan、Robbert Krebbers 和 Derek Dreyer。RustBelt：保障 Rust 编程语言的基础。ACM程序语言，2(POPL)，2017年12月。doi：10.1145/3158154。网址：https://doi.org/10.1145/ 3158154。[15] Son Ho 和 Jonathan Protzenko。AENEAS：通过函数式转换验证 Rust。ACM程序语言，6(ICFP)，2022年8月。doi：10.1145/3547647。网址：https: //doi.org/10.1145/3547647。[16] Son Ho、Aymeric Fromherz 和 Jonathan Protzenko。通过符号语义实现 Rust 的健全借用检查。ACM程序语言，8(ICFP)，2024年8月。doi：10.1145/ 3674640。网址：https://doi.org/10.1145/3674640。[17] Julian Mackay、Hannes Mehnert、Alex Potanin、Lindsay Groves 和 Nicholas Cameron。使用 Coq 证明助手对带赋值和不可变性的轻量级 Java进行编码。载于类Java程序形式化技术研讨会（FTfJP 2012），第 11–19 页。ACM，2012年。doi：10.1145/2318202.2318205。[18] Stef Gijsberts。《Prusti 实践：使用 Prusti 主动式程序验证器验证 Rust 的案例研究》。博士论文，2023年3月。[19] Vytautas Astrauskas、Aurel B´ıl´y、Jon´aˇs Fiala、Zachary Grannan、Peter M¨uller、Federico Poli 和 Alexander Summers。Prusti项目：Rust形式化验证，第 88–108 页。2022年5月。ISBN 978-3-031-06772-3。doi：10.1007/978-3-031-06773-0 5。[20]Lennard G¨aher、Michael Sammler、Ralf Jung、Robbert Krebbers 和 Derek Dreyer。RefinedRust：用于高保证验证 Rust 程序的类型系统。ACM程序语言论文集，8：1115–1139，2024年。网址：https: //api.semanticscholar.org/CorpusID:270398597。[21]Alexa VanHattum、Daniel Schwartz-Narbonne、Nathan Chong 和 Adrian Sampson。验证 Rust 中的动态 trait 对象。2023年。网址：https://www.amazon.science/publications/verifying-dynamic-trait-objects-in-rust。[22] Isaac Oscar Gariano、Marco Servetto、Alex Potanin 和 Hrshikesh Arora。迭代式组合静态验证的特征。理论计算机科学电子论文集，299：7，2019年。doi：10.4204/EPTCS.299.7。[23] Dorian Schiedt。改进 Rust Analyzer 中的“提取函数”，2022年8月。网址：https:

//dorianlistens.com/2022/07/improving-extract-function-in-rust-analyzer/#:~:文本=根据 Rust-Analyzer 官方网站。

[24] Raul Marticorena、Carlos L´opez、Yania Crespo 和 F. Javier P´erez。《Java 泛型重构：提取方法案例研究》。第 212–221 页，2010 年。doi：10.1109/CSMR.2010.38。

[25] Yang Zhang、Zhaoyang Xie、Yanxia Yue 和 Lin Qi。基于 CompletableFuture 的异步机制自动重构方法。应用科学，14(19)，2024 年。ISSN 2076-3417。doi：10.3390/app14198866。网址https://www.mdpi.com/2076-3417/14/19/8866。

[26] Yu Lin、Semih Okur 和 Danny Dig。Android 异步编程的研究与重构（t）。第 224–235 页，2015 年。doi：10.1109/ASE.2015.50。

[27] Huiqing Li、Simon Thompson 和 Claus Reinke。Haskell 重构工具 HaRe 及其应用程序接口。电子理论计算机科学通讯，141：29–34，2005 年 12 月。doi：10.1016/j.entcs.2005.02.053。

[28] URLhttps://pybowler.io/。

[29] David Young、Ziyi Yang、Ilya Sergey 和 Alex Potanin。用于演绎合成带指针程序的高阶规范。载于第 *38* 届欧洲面向对象编程会议（*ECOOP 2024*），第 313 卷，*LIPIcs*，第 45:1–45:26 页。Schloss Dagstuhl - 莱布尼茨信息学中心，2024 年。doi：10.4230/LIPIcs.ECOOP.2024.45。

[30]Evan Li。《Rust 星标排名：前 100 名》。https://evanli.github.io/GithubRanking/Top100/Rust.html，2025 年 11 月。

[31] Aditya Jain、Ramta Bansal、Avnish Kumar 和 K. D. Singh。基于性别和身体活动水平对医学院一年级学生视觉与听觉反应时间进行的比较研究。《国际应用与基础医学研究杂志》，5（2）：124–127，2015 年 5 月至 8 月。ISSN 2229-516X。doi：10.4103/2229-516X.157168。

[32] Tobias Runge、Tabea Bordis、Alex Potanin、Thomas Th¨um 和 Ina Schaefer。灵活的构造即正确编程。《计算机科学中的逻辑方法》，19（2）：16:1– 16:36，2023年。doi：10.46298/lmcs-19(2:16)2023。