arXiv:2303.10515v1 [cs.PL] 18 Mar 2023

所有权引导的 C 到 Rust 翻译

Hanliang Zhang[1], Cristina David[1],Yijun Yu[2],和 Meng Wang[1]

[1] 布里斯托大学{
pd21541,cristina.david,meng.wang}@bristol.ac.uk[2]
开放大学 yijun.yu@open.ac.uk

摘要。Rust 被称为更安全的 C，是一种现代编程语言，兼具内存安全性和底层控制能力。这种引人注目的结合使 Rust 深受开发者欢迎，将遗留代码库（通常是用 C 编写的）迁移到 Rust 的趋势也日益明显。本文提出了一种以静态所有权分析为核心的 C 到 Rust 转换方法。我们设计了一套分析方法，用于推断 C 指针的所有权模型，并自动将这些指针转换为安全的 Rust 等价形式。由此构建的工具 Crown 可扩展至真实世界的代码库（不到 10 秒即可处理 50 万行代码），并实现了较高的转换率。

1 引言

Rust [28] 是一种现代编程语言，兼具内存安全性和底层控制能力，这种组合令人瞩目。具体而言，Rust 借鉴了所有权类型和次结构类型（主要是仿射类型和线性类型），并限制某些内存访问只能由其所有者执行。这意味着，尽管多个指针可以同时指向同一资源（通过一种称为借用的机制），但某些操作只能由所有者执行。Rust 编译器能够静态验证所有权约束，从而保证内存安全和线程安全。可证明安全这一独特优势使 Rust 成为一种广受欢迎的语言，而将 C 语言编写的遗留代码库迁移到 Rust 的前景也极具吸引力。

为满足这一需求，工业界和学术界都涌现出将 C 代码翻译为 Rust 的自动化工具 [14, 21, 26]。其中，工业级翻译器 C2Rust [21] 会将 C 代码改写为 Rust 语法，同时保留原始语义。这种翻译不会合成所有权模型，因此除了复制 C 语言中对指针的不安全用法之外，无法做到更多。因此，Rust 代码必须标记 `unsafe` 关键字，以允许执行编译器不会检查的某些操作。较新的研究则专注于减少这种不安全标记。具体而言，工具 Laertes [14] 旨在通过根据 Rust 编译器的类型错误消息搜索解空间，改写 C2Rust 生成的（不安全）代码。这是一项开创性工作，因为它首次实现了合成恰当的 Rust 代码，而不再局限于逐行直接转换原始 C 源代码。

另一方面，试错方法的局限性也很明显：该系统既不支持对生成过程进行推理，也无法对目标代码形成新的理解（显而易见的“代码已成功编译”这一事实除外）。

在本文中，我们采用一种更有原则性的方法，开发了一种新颖的指针所有权分析。该分析高效（可在不到 10 秒内扩展至大型程序，规模达 50 万代码行）、功能强大（能够处理嵌套指针和归纳定义的数据结构），且精确（具备字段敏感和流敏感性）。得益于我们对 Rust 所有权模型所作的一项强化假设，该所有权分析既可扩展又精确，并且无需进行别名分析。

这项分析的首要目标当然是促进 C 语言到 Rust 的翻译。实际上，正如我们将在本文后续部分看到的，我们构建了一个自动化转换系统，将所有权模型编码到生成的 Rust 代码中，随后由 Rust 编译器证明其安全性。但其意义不止于此。与现有方法通常尝试使用 Rust 编译器不同 [14, 26], ，这种分析方法能够从代码中提取新的所有权知识，从而有望实现多种用途，包括防止内存泄漏、识别本质上不安全的代码片段、检测内存错误等。具体而言，本文

– 设计一种可扩展且精确的所有权分析，能够处理复杂的归纳定义的数据结构和嵌套指针。（第 5 节）– 开发一种利用所有权分析来增强代码安全性的 Rust 重构机制。（第 6 节）– 实现原型工具 Crown（意为“由所有权引导的 C 语言到 Rust 翻译”），将 C 代码翻译为安全性更高的 Rust。（第 7 节）– 使用基准测试套件评估 Crown ，该套件涵盖常用的数据结构库和实际项目（规模从 150 到 50 万代码行不等），并将结果与最先进水平进行比较。（第 8 节）

2 背景

我们首先简要介绍 Rust，特别是其所有权系统和指针的使用，因为它们对内存安全至关重要。

2.1 Rust 所有权模型

Rust 中的所有权是一组规则，用于规定 Rust 程序如何管理内存 [28]。其理念是为每个值指定一个唯一所有者。这一特性有助于内存管理。例如，当所有者离开作用域时，为该值分配的内存便可自动回收。

```
1可变变量 v  = ...2
可变变量 u  = v;            // ownership is transferred to u
```

在上面的代码片段中，将 v 赋值给 u 也会转移所有权；之后，v 会离开作用域，无法再次使用。

这种永久的所有权转移提供了强有力的保证，但在编程中可能难以管理。对于暂时的所有权转移（称为 借用），可以使用 引用 （以与号标记）。例如，在 `f(&mut x)` 中，x 的值的所有权会暂时借给函数调用，直到函数返回。

时间这一概念为所有权管理引入了另一个维度，称为 生命周期。对于可变引用（如上例中由 `mut` 标记的引用），规则相对简单：任意时刻只允许存在一个可变引用。而对于不可变引用（没有 `mut` 标记的引用），只要同一时刻不存在可变引用，就可以有多个不可变引用共存。可以想见，可变引用和不可变引用之间的交互及其生命周期极其复杂。本文重点分析可变引用，因为它们比不可变引用更常见。

2.2 Rust 中的指针类型

Rust 的指针系统比 C 语言更丰富。原始的 C 风格指针（写作 `*const T` 或 `*mut T`）称为 原始指针，Rust 编译器不会对它们进行所有权和生命周期检查。原始指针是不安全 Rust 的主要来源之一（下文将详细介绍）。符合 Rust 惯用风格的代码则提倡使用Box 指针（写作 `Box<T>`）作为所有权指针，独占堆分配；也提倡使用引用（写作 `&mut T` 或 `& T` ，如上一小节所述）作为非所有权指针，用于访问由其他对象所有的值。

C 风格数组指针在 Rust 中表示为数组引用和切片引用，数组边界分别在编译时和运行时确定。创建数组边界等元数据不属于所有权分析的范畴。在本工作中，我们在转换后的代码中将数组指针保留为原始指针。

2.3 不安全 Rust

作为一种务实的设计，Rust 允许程序包含无法由编译器验证为内存安全的特性。这包括解引用原始指针、调用低级函数等。此类用法必须标记为 不安全关键字，并构成 不安全 Rust的一部分。值得注意的是，不安全不会关闭编译器检查；安全指针仍会受到检查。

不安全 Rust 的主要目的是支持底层系统编程，但它也可用于其他用途。例如，c2rust [21] 会将 C 指针直接转换为原始指针。没有不安全 Rust，生成的代码将无法编译。

3 概述

本节通过两个示例概述 CROWN 。第一个示例详细介绍了单向链表的推入方法，第二个示例则展示了一个来自实际基准测试的代码片段。

```c
struct Node {
    int data;
    struct Node * next;
};

struct List {
    Node * head;
};

void push(struct List* list, int
        new_data) {
    struct Node* new_node = (struct
        Node*) malloc(sizeof(struct
        Node));
    new_node->data  = new_data;
    new_node->next = list->head;
    list->head    = new_node;
}

```

(a) C 代码

```rust
#[repr(C)]
#[derive(Copy, Clone)]
pub struct Node {
    pub data: i32,
    pub next: *mut Node,
}

#[repr(C)]
#[derive(Copy, Clone)]
pub struct List {
    pub head: *mut Node,
}

pub unsafe extern "C" fn push(mut
        list: *mut List, mut
        new_data: i32) {
    let mut new_node = malloc(::std
        ::mem::size_of::<Node>() as
        libc::c_ulong) as *mut Node;
    (*new_node).data = new_data;
    (*new_node).next = (*list).head;
    (*list).head = new_node;
}

```

(b) c2rust 转换结果

```rust
#[repr(C)]
pub struct Node {
    pub data: i32,
    pub next: Option<Box<Node>>,
}

#[repr(C)]
pub struct List {
    pub head: Option<Box<Node>>,
}

pub unsafe extern "C" fn push(mut
        list: Option<&mut List>, mut
        new_data: i32) {
    let mut new_node = Some(Box::new
        (<Node as Default>::default
        ()));
    (*new_node.as_deref_mut().unwrap
        ()).data = new_data;
    (*new_node.as_deref_mut().unwrap
        ()).next = (*list.
        as_deref_mut().unwrap()).
        head.take();
    (*list.as_deref_mut().unwrap()).
        head = new_node;
}

```

(c) CROWN 结果

图 1：向单向链表中 push

3.1 向单向链表中 push

函数 push在图 1a 中的 C 代码会分配一个新节点，并将作为参数接收的数据存储其中。随后，新节点会成为链表的头节点。这段代码由 c2rust翻译成图 1b 中的 Rust 代码。值得注意的是，c2rust的翻译基于语法，只是将所有 C 指针改为*mut 裸指针。由于在 Rust 中，解引用裸指针被视为不安全操作（例如，图 1b 第 16 行对 新_节点的解引用），方法 push 必须使用 不安全关键字进行注解。此外，c2rust还为两个结构体定义引入了两条指令： #[repr(C)] 和 #[derive(Copy, Clone)]。前者使数据布局与 C 中相同，以便进行互操作；后者则指示相应类型只能通过复制来生成副本。

虽然 c2rust 在翻译中使用了原始指针，但图 1b 中的所有权方案遵循 Rust 的所有权模型，这意味着原始指针可以翻译为安全指针。在第 15 行，新分配的节点的指针被赋给 new_node。据此，我们可以推断，新分配节点的所有权属于 new_node。

随后，在第 18 行，所有权从 new_node 转移到 (*list).head。此外，如果 (*list).head 在第 17 行之前拥有任何内存对象，那么其所有权会在第 17 行转移给 (*new_node).next。这种所有权方案对应于安全指针的使用方式：(i) 每个内存对象都关联着一个唯一所有者；(ii) 当其所有者离开作用域时，该对象会被释放。就 (i) 而言，例如，当新分配内存的所有权在第 18 行从 new_node 转移到 (*list).head 时，(*list).head 成为唯一所有者，而 new_node 则失效，不再使用。就 (ii) 而言，鉴于 push 的参数列表中的参数是一个输出参数（即可以从函数外部访问的参数），我们假定该参数在方法退出时必须具有所有权。因此，push 方法不会释放任何内存对象，而是将其返回给调用者。

Crown 推断由 c2rust 翻译的代码的所有权信息，并在图 1c 中利用这些信息将代码转换为更安全的 Rust。如下文所述，Crown 首先根据所有权信息将原始指针重新定型为安全指针，然后改写它们的用法。

在 Crown 中重新定型指针。如果一个指针在其作用域内的任意时刻拥有 Crown 一个内存对象，就将其重新定型为 Box 指针。例如，在图 1c 中，局部变量 new_node被重新定型为 Option<Box<Node>> （为处理空指针值，安全指针类型会包装在 Option 中）。变量 new_node在函数入口处不拥有对象，在第 13 行变为拥有对象，并在第 16 行再次转移所有权。

对于结构体字段， Crown 会考察结构体声明作用域内的所有代码。如果某个结构体字段在其结构体声明作用域内的任意时刻拥有内存对象，那么就会将其重新定型为 Box。在图 1b 中，字段 next 和head通过访问路径 (*new_node).next 和 (*list).head进行访问，并分别在第 17 行和第 18 行获得所有权。因此，在图 1c 中，它们分别在第 4 行和第 9 行被重新定型为 Box 。

一种特殊情况是输出参数，例如 示例中的 list。对于这类参数，即使它们可能拥有所有权， Crown 也会将其重新定型为 &mut，以便进行借用。在 push中，输入参数 list 被重新定型为Option<&mut List>。

在 Crown 中重写指针用法。指针重新定型后， Crown 会重写这些指针的用法。重写过程会同时考虑它们的新类型以及使用它们的上下文。由于 Rust 语义的影响，重写规则略显复杂（见第 6 节）。例如，第 14 行对new_node的解引用会被重写为 (*new_node).as_deref_mut().unwrap() ，因为它需要被修改，而且必须对 Box的可选部分调用 unwrap()。类似地，在第 15 行，(*list).head 会被重写为 ((*list.as_deref_mut()).unwrap()).head.take() ，因为赋值的左值需要一个 Box 指针。

3.2 释放 bzip2 中的参数列表

接下来，我们展示一个包含循环结构的真实世界代码片段的转换：bzip2 中一段释放参数列表的代码。bzip2 定义了一个单向

```c
typedef
struct zzzz {
    Char        *name;
    struct zzzz *link;
}
Cell;
[...]
Cell* aa = argList;
while (aa != NULL) {
    Cell* aa2 = aa->link;
    if (aa->name)
        free(aa->name);
    free(aa);
    aa = aa2;
}
```

（a）C 定义

```c
#[derive(Copy, Clone)]
#[repr(C)]
pub struct zzzz {
    pub name: *mut Char,
    pub link: *mut zzzz,
}
pub type Cell = zzzz;
[...]
let mut aa = argList;
while !aa.is_null() {
    let mut aa2 = (*aa).link;
    if !(*aa).name.is_null() {
        free((*aa).name as *mut libc
        ::c_void);
    }
    free(aa as *mut libc::c_void);
    aa = aa2;
}
```

（b）c2rust 结果

```rust
#[repr(C)]
pub struct zzzz {
    pub name: *mut /* owning */ Char
        ,
    pub link: Option<Box<zzzz>>,
}
pub type Cell = zzzz;
[...]
let mut aa = argList;
while !aa.as_deref().is_none() {
    let mut aa2 = (*aa.as_deref_mut
        ().unwrap()).link.take();
    if !(*aa.as_deref().unwrap()).
        name.is_null {
        free((*aa.as_deref().unwrap
        ()).name as *mut libc::
        c_void);
    }
    aa = aa2;
}
```

(c) CROWN 结果

图 2：释放参数列表

链表式结构 `Cell`，用于存储参数名称列表。在图 2 中，我们从源代码中提取了一段用于释放参数列表的代码。这里，局部变量 `argList`是一个已构建好的参数列表，而 `Char` 是 C 风格字符的类型别名。

CROWN 准确推断出此代码片段的所有权方案。首先，argList 的所有权转移给 aa，后者将在随后的循环中被释放。在循环内部，通过 aa 访问的 link 的所有权首先转移给 aa2，随后通过 aa 访问的 name 的所有权在调用 free 时被释放。循环结束后，aa 的所有权也被释放。最后，aa 从 aa2 重新获得所有权。

循环的处理。对于循环， CROWN 只分析其循环体一次，因为这样就足以揭示所有权信息。对于 `Cell` 这样的归纳定义的数据结构，虽然进一步展开循环体可以更深入地遍历数据结构，但不会揭示任何新的结构体字段：指针变量和指针结构体字段在循环迭代之间不会改变所有权。此外， CROWN 会生成约束，使循环入口和出口处所有局部指针的所有权相等。例如，循环入口处 `aa`和 `aa2` 的所有权状态被设为与循环出口处相等，并分别推断为拥有所有权和不拥有所有权。

空指针的处理。在 C 语言中，指针在调用 malloc 后或 free 前与空值进行检查，是一种常见的 C 惯用法：`if !p.is_null() { free(p);}`。这可能会造成问题，因为 then 分支和 else 分支中 `p` 的所有权状态会相互冲突。我们采用与 [19]类似的解决方案：在空值分支中插入显式的空值赋值：`if !p.is_null() {free(p); } else { p = ptr::null_mut(); }`。由于我们将空指针视为既拥有所有权又不拥有所有权，`p` 的所有权将由非空分支决定，从而使 CROWN 能够推断出正确的所有权方案。

翻译。使用上述所有权方案，CROWN 会执行图 2c 中所示的重写。请注意，我们不会尝试重写 name，因为它是一个数组指针（有关局限性，请参见第 7 节）。

4 架构

本节简要介绍 CROWN 的架构。CROWN 以 c2rust 自动翻译的 Rust 程序为输入。这些程序与原始 C 语言程序非常相似，只是将 C 语言语法替换成了 Rust 语法。CROWN 会对 Rust 的 MIR 执行若干静态分析，以推断指针的属性：

– 所有权分析：计算代码中指针的所有权信息，也就是说，推断每个指针在特定程序位置是否拥有所有权。– 可变性分析：推断哪些指针被用于修改其指向的对象（受 [18, 20] 启发）。– 胖指针分析：区分数组指针和非数组指针（受 [27] 启发）。

这些分析的结果会归纳为类型限定符 [17]。类型限定符是一种原子属性（即所有权、可变性和胖指针属性），用于限定标准指针类型。随后，这些限定符会用于指针重定型。例如，拥有所有权的非数组指针会被重定型为`Box`。指针重定型后，CROWN 会相应地改写指针的用法。

5 所有权分析

我们的所有权分析旨在为给定程序计算一种符合 Rust 所有权模型的所有权方案（如果存在这样的方案）。该所有权方案包含程序中指针在特定程序位置上是拥有所有权还是不拥有所有权的信息。总体而言，我们的分析通过生成一组所有权约束（第 5.2 节）来完成，随后由 SAT 求解器求解这些约束（第 5.3 节）。所有权约束的一个满足赋值就是一种符合 Rust 语义的所有权方案。

我们的所有权分析是流敏感且字段敏感的，后者使我们能够推断指针结构体字段的所有权信息。为了满足字段敏感性，我们会跟踪访问路径 [8, 12, 24] 的所有权信息。访问路径通过从初始基变量访问内存位置的方式来表示该位置，由基变量和一系列字段选择运算符组成。对于图 1b 中的程序，一些访问路径示例包括 new_node（仅由基变量组成）、(*new_node).next 和 (*list).head。我们的分析为每条访问路径关联一个所有权变量，例如，p 关联的所有权变量为 $\mathbb{O}_p$，而 (*p).next 关联的所有权变量为 $\mathbb{O}_{(*p).next}$。如果相应的访问路径拥有所有权，则每个所有权变量可取值 1；

如果访问路径不拥有所有权，则取值 0。访问路径的所有权指的是通过该访问路径最后访问的字段（或更一般地说，指针）的所有权。例如，(*new_node).next 的所有权指的是字段 next 的所有权。

5.1 所有权与别名

设计所有权分析的一大挑战，是所有权与别名之间的相互作用。为了理解这个问题，我们来看下面代码清单第 3 行的指针赋值。我们假设，赋值之前的各行足以推断出 q、(*q).next 和 r 是拥有所有权的，而 p 和 (*p).next 不拥有所有权。此外，我们假设赋值之后的各行要求 (*p).next 拥有所有权（例如，显式释放 (*p).next）。据此，所有权分析可以合理地得出结论：第 3 行发生了所有权转移（使 (*p).next 获得所有权），并且推断出的所有权方案遵循 Rust 语义。

```
1 令 p, r, q: *可变 节点;2   // p and (*p).next non-owning; q,
(*q).next and r owning3 (*p).next = r;
4   // (*p).next must have ownership
```

现在我们也来考虑别名。一种可能的假设是，在第 3 行之前，p 和 q 是别名，这意味着 (*p).next 和 (*q).next 也是别名。那么，在第 3 行之后，(*p).next 和 (*q).next 仍然会是别名（指向同一个内存对象）。然而，根据上述所有权方案，(*p).next 和 (*q).next 都是所有者，而这在 Rust 中是不允许的，因为一个内存对象必须有唯一所有者。上面模拟的所有权分析并未检测到这种不一致。问题在于所有权分析忽略了别名。事实上，如果存在一个拥有所有权的别名，并且在所有权转移后仍指向与 (*p).next 相同的内存对象，那么就不应将所有权转移给 (*p).next。

精确计算别名信息非常困难，尤其是在存在归纳定义的数据结构时。在本文中，我们通过对 Rust 所有权模型作出更强的假设，免去了检查别名的必要：我们限制指针沿访问路径获取所有权的方式，从而限制所有权与别名之间的相互作用。具体而言，我们提出了一个全新的概念：所有权单调性。该性质指出，沿着访问路径，指针的所有权值只能递减（见定义 1，其中 $is\_prefix(p, q)$ 在访问路径 p 是 q 的前缀时返回 true，否则返回 false——例如，$is\_prefix(p, (*p).next) = true$）。回到我们的例子，所有权单调性意味着，对于访问路径 (*p).next，我们有 $\mathbb{O}_p \geq \mathbb{O}_{(*p).next}$；对于访问路径 (*q).next，我们有 $\mathbb{O}_q \geq \mathbb{O}_{(*q).next}$。这意味着，如果允许 (*p).next 获取所有权，那么 p 本身必须已经拥有所有权。因此，p 的所有别名都必须是

非拥有的，这意味着 (*p).next 的所有别名（包括 (*q).next）都是非拥有的。

定义 1（所有权单调性）。给定两个访问路径 p 和 q，如果 $is\_prefix(p, q)$，那么 $\mathbb{O}_p \geq \mathbb{O}_q$。
所有权单调性比 Rust 语义更严格，因此我们的分析可能会拒绝某些原本会被 Rust 编译器接受的所有权方案。我们将在本节后文回到这一点。

5.2 所有权约束的生成

在约束生成过程中，我们假设给定一个 k，表示代码中使用的最长访问路径的长度。这样我们就能够捕获代码中出现的所有访问路径的所有权。本节稍后将讨论如何处理循环，因为循环可能会产生更长的访问路径。

接下来，我们用 $\mathcal{P}$ 表示程序中所有访问路径的集合，$base\_var(p)$ 返回访问路径 p的基变量，而 $|p|$ 根据从基变量应用的字段选择运算符的数量，计算访问路径 p 的长度。举例来说，$base\_var((*p).next) = p$，$base\_var(p) = p$，$|p| = 1$ 且$|(*p).next| = 2$。然后，我们定义 $ap(v, lb, ub)$，使其返回基变量为 v 且长度介于下界 lb 和上界 ub之间的访问路径集合：$ap(v, lb, ub) = \{p \in \mathcal{P} | base\_var(p) = v \wedge lb \leq |p| \leq ub\}$。举例来说，我们有$ap(new_node, 1, 2) = \{new_node, (*new_node).next\}$。

$$\text{ASSIGN}$$
$$\frac{\begin{array}{c} a = base\_var(v), \quad b = base\_var(w), \\ p \in ap(a, |v|, k), \quad q \in ap(b, |w|, k), \quad r \in ap(a, 1, |v|{-}1), \quad s \in ap(b, 1, |w|{-}1) \\ |p| - |v| = |q| - |w|, \quad |r| = |s| \\ C' = C \cup \{\mathbb{O}_p = 0 \wedge \mathbb{O}_{p'} + \mathbb{O}_{q'} = \mathbb{O}_q \ \wedge \mathbb{O}_{r'} = \mathbb{O}_r \wedge \mathbb{O}_{s'} = \mathbb{O}_s\} \end{array}}{C \vdash \mathrm{v} = \mathrm{w}; \ \Rightarrow C'}$$

图 3：赋值的所有权约束生成

所有权转移。可能发生所有权转移的程序指令包括（指针）赋值和函数调用。这里我们讨论赋值；受篇幅限制，跨过程所有权分析的规则请参见附录 A。赋值位置的所有权转移规则遵循 Rust 的Box语义：移动一个Box 指针时，它所指向的对象也会随之移动。例如，以下 Rust 伪代码片段：

```
1令p,q: Box<Box<i32>>; 2 令p = q;
// ownership transfer occurs
3 // the use of q and *q is disallowed
```

当所有权从 q 转移到 p时，*q 也会失去所有权。除重新赋值外，不允许使用失去所有权后的 Box 指针，因此第 3 行禁止使用 q 或 *q 。

因此，我们强制执行以下所有权转移规则：如果某个指针变量（例如示例中的 p 和 q）发生所有权转移，那么从该指针可达的所有指针（例如 *p 和 *q）也必须发生所有权转移。能够到达所讨论指针的指针变量，其所有权保持不变（例如，如果代码中的某次赋值 *p = *q 发生了所有权转移，那么 q 和 p 保留各自先前的所有权值）。

指针赋值时可能发生的所有权转移：图 3 中的 ASSIGN 规则描述了指针赋值位置处的所有权转移规则。判断式 $C \vdash$ v =w; $\Rightarrow C'$ 表示在约束集 C 下分析该赋值，并生成 C'。我们使用撇号记法表示赋值后的变量。对于指针赋值 $v = w$， p 和 q 分别表示从 v 和 w 开始的所有访问路径，而 r 和 s 表示从 v 和 w 的基变量出发、分别到达 v 和 w 的访问路径。随后，等式 $\mathbb{O}_{p'} + \mathbb{O}_{q'} = \mathbb{O}_q$ 刻画了从 v 和 w 出发的所有访问路径发生所有权转移的可能性：(i) 若发生转移，则 q 的所有权转移给 p'（$\mathbb{O}_{p'} = \mathbb{O}_q$ 和 $\mathbb{O}_{q'} = 0$）。(ii) 否则，所有权值保持不变（$\mathbb{O}_{p'} = \mathbb{O}_p$ 和 $\mathbb{O}_{q'} = \mathbb{O}_q$）。最后两个等式 $\mathbb{O}_{r'} = \mathbb{O}_r \wedge \mathbb{O}_{s'} = \mathbb{O}_s$ 表示，在 (i) 和 (ii) 两种情况下，访问路径 r 和 s 上的指针都保留其先前的所有权。

C 内存泄漏：在 ASSIGN 规则中，我们添加约束 $\mathbb{O}_p = 0$ 到 C'，以强制在赋值之前使 p 不拥有所有权。反过来，在通过所分析的赋值重新赋值之前，若 p 拥有所有权，则表明原始 C 程序存在内存泄漏。鉴于 Rust 会自动回收内存，允许进行这种翻译会通过修复内存泄漏而改变原始程序的语义。因此，我们选择禁止所有权分析生成此类解。

沿访问路径同时转移所有权：可以发现，ASSIGN 生成的约束并未完整体现所述的所有权转移规则。具体而言，这些约束无法确保：每当所有权从 w 转移到 v时，所有访问路径 p 和q上的所有指针也都会发生所有权转移。相反，根据定理 1 所述，这一点由所有权单调性规则隐式保证。

定理 1（所有权转移）。如果所有权从 w 转移到 v，那么根据 ASSIGN 规则和所有权单调性，所有访问路径 p 和 q上对应指针之间的所有权也会转移：$\mathbb{O}_{p'} = \mathbb{O}_q$ 和 $\mathbb{O}_{q'}$ =0。（证明见附录 C）

所有权与别名：我们在第 5.1 节中看到，别名可能导致这样的情况：所有权转移后，同一个内存对象拥有多个所有者。定理 2 指出，在所有权单调性下，这种情况不可能发生。

定理 2（所有权单调性下 ASSIGN 的可靠性）。在所有权单调性下，如果所有已分配的内存对象在应用 ASSIGN 之前都具有唯一

所有者，那么应用 ASSIGN 后它们也将具有唯一所有者。（证明见附录 B）

直观而言，定理 2 使指针无需考虑别名即可获得所有权：所有权转移后，该指针将成为唯一所有者。这一想法与强更新的思路相似 [25]。

其他访问路径：补充说明，程序中的其他基变量也可能可以访问 v 和 w。在这种情况下，鉴于所有权转移位置没有明确提及这些访问路径，我们不会为它们生成新的所有权变量。因此，默认情况下，它们当前的所有权变量保持不变。

所有权转移示例。为说明 ASSIGN 规则，我们使用下面的单向链表示例，并假设p、q 的类型均为*mut 节点。因此，我们需要考虑以下四条访问路径：p，q，(*p).next，(*q).next。在 SSA 风格下，示例中的每一行都会为该行提及的访问路径生成新的所有权变量（通过递增下标）。对于第一个赋值，所有权可以分别在 p 与 q之间，以及（$*p$）$.next$ 与（$*q$）$.next$之间转移。对于第二个赋值，所有权可以在（$*p$）$.next$ 与（$*q$）$.next$之间转移，而 p 与 q 必须保留其先前的所有权。

```
1 p = q;      // 𝕆_{p1} = 0 ∧ 𝕆_{p2} + 𝕆_{q2} = 𝕆_{q1}  ∧
2             // 𝕆_{(*p1).next} = 0 ∧ 𝕆_{(*p2).next} + 𝕆_{(*q2).next} = 𝕆_{(*q1).next}
3 (*p).next = (*q).next;
4             // 𝕆_{p3} = 𝕆_{p2} ∧ 𝕆_{q3} = 𝕆_{q2}  ∧
5             // 𝕆_{(*p2).next} = 0 ∧ 𝕆_{(*p3).next} + 𝕆_{(*q3).next} = 𝕆_{(*q2).next}
```

除了为赋值生成所有权约束外，我们还必须对常用 C 标准函数（如 malloc、calloc、realloc、free、strcmp、memset 等）的所有权信息进行建模。由于篇幅所限，关于这些函数的更多细节，以及所有权单调性和过程间所有权分析的规则，详见附录 A。

处理条件语句和循环。如第 3.2 节所述，我们只分析一次循环体，因为这样就足以揭示所有必需的所有权变量。对于归纳定义的数据结构，虽然进一步展开循环体会增加访问路径的长度，但不会揭示任何新的结构体字段（结构体字段的所有权不会在循环迭代之间发生变化）。

为处理控制路径的汇合点，我们采用一种变体的 SSA 构造算法 [4], ，其中不同路径通过 ϕ 节点合并。每个所有权变量的值必须在所有合并路径上都相同，否则分析将失败。

5.3 求解所有权约束

所有权约束系统由一组形式为 $O_v = O_w + O_u$ 的 3 变量线性约束，以及形式为 $O_v = 0$ 和 $O_v = 1$ 的 1 变量等式约束组成。

定义 2（所有权约束系统）。一个所有权约束系统 $(P, \Delta, \Sigma, \Sigma_\neg)$ 由一组所有权变量 P、一组 3 变量等式约束 $\Delta \subseteq P \times P \times P$，以及两组 1 变量等式约束 Σ、$\Sigma_\neg \subseteq P$ 组成，其中所有权变量可以取值 0 或 1。Σ 中的等式形式为 $x = 1$，而 $\Sigma_\neg$ 中的等式形式为 $x = 0$。

定理3（所有权约束求解的复杂性）。所有权约束求解是 NP完全问题。（证明见附录D）

我们通过调用 SAT 求解器来求解所有权约束。所有权约束可能无解。这种情况会在不存在符合 Rust 所有权模型和所有权单调性（在某些情况下比 Rust 模型更严格）的所有权方案时发生，也可能是因为原始 C 程序存在内存泄漏。

由于 Rust 语义复杂，我们不对满足赋值是否符合 Rust 所有权模型进行形式化证明。相反，我们会在翻译后运行 Rust 编译器来执行这项检查。

5.4 关于所有权单调性的讨论

如前文第 5.4 节所述，所有权单调性比 Rust 语义更严格，因此我们的分析可能会拒绝一些原本会被 Rust 编译器接受的所有权方案。我们发现，最显著的此类情况是引用输出参数。这指的是以引用形式传递的函数参数，它可以从函数外部访问，因此可充当输出（例如图 1a 中的 链表 ）。对于这类参数，基变量是非拥有的（因为它是引用）且可变，而从它可达的指针则可能是拥有的（见图 1c 中的示例，其中 `(*node).head` 被赋值为一个指向新分配节点的指针）。我们会检测此类情况并显式启用它们。具体而言，在翻译阶段，我们会将拥有指针 `p` 显式转换为 `&mut(*p)`。

6 C 语言到 Rust 的翻译

Crown 利用所有权、可变性和胖指针属性分析的结果来执行实际翻译，包括指针重新定型（第 6.1 节）和重写指针用法（第 6.2 节）。

6.1 指针重新定型

如第 2.2 节所述，我们不会尝试将数组指针转换为安全指针。在本节余下部分，我们将重点讨论可变的非数组指针。

翻译需要从全局角度考察指针的所有权，而所有权分析推断出的信息则针对各个程序位置。为了进行翻译，鉴于我们会将拥有所有权的指针重构为 Box

指针，如果某个指针在其作用域内的任一程序位置拥有一个内存对象，则从全局来看，该指针被视为拥有所有权；否则，它从全局来看不拥有所有权。在重新确定结构体指针字段的类型时，我们必须考虑结构体声明的作用域，而该作用域通常涵盖整个程序。在此作用域内，每个字段通常会通过多个基变量进行访问，因此必须将它们全部纳入考虑。例如，给定图 1b 中的 `List` 声明，以及两个类型为`l1` 和 `l2` 的变量`*mut List`。那么，为了确定字段 `next`的所有权状态，我们必须考虑从基变量 `next`l1和 l2出发的所有访问路径。

下表展示了可变非数组指针的重定型规则。为处理空指针值，我们将安全指针类型封装在 `Option`中：

	非数组指针
拥有所有权	`Option<Box<T>>`
非拥有所有权	`*mut`T 或 `Option<&mut T>`

保留为原始指针的非所有权指针 `*mut` T 对应可变局部借用。要使 Crown 能够可靠地将其翻译为可变局部借用，就必须分析生命周期信息。鉴于 Rust 中一个值不能有两个可变引用， Crown 就必须检查指向同一对象的不同可变引用的生命周期是否存在重叠。在本研究中，我们选择不做这项工作，而是将其留待未来处理（第 7 节的局限性部分也提到了这一点）。值得注意的是，这项限制不适用于输出参数，我们会将其翻译为可变引用。这已经涵盖了大多数可变引用——Das 观察到，在 C 代码中，大多数引用都是因为将变量的地址作为参数传递而产生的 [11]。值得注意的是，由于缺少生命周期分析，我们也无法处理不可变局部借用，因此我们的翻译侧重于可变指针。

6.2 重写指针用法

指针表达式的重写方式取决于其新类型以及使用它的上下文。例如，重写`q` in `p = q`时，上下文取决于 `p`的新类型。根据这种新类型，我们有四种上下文：BoxCtxt ，要求使用 `Box` 指针；MutCtxt ，要求使用 `&mut`引用；ConstCtxt ，要求使用 `&` 引用；以及 RawCtxt ，要求使用原始指针。例如，如果上文中的 `p` 是一个 `Box` 指针，那么我们就在 BoxCtxt上下文中重写 q。

接下来，重写将按照下表进行：列对应要重写的指针的新类型，行表示可能的上下文 [3]。

	`Option<Box<T>>`	`Option<&可变 T>`	`*可变 T`
BoxCtxt	p.take()	⊥	`Some(Box::from_raw(p))`
MutCtxt	p.as_deref_mut()	p.as_deref_mut()	p.as_mut()
ConstCtxt	p.as_deref()	p.as_deref()	p.as_ref()
RawCtxt	至_raw(&mut p)	到_raw(&mut p)	p

[3] 由于我们对输出参数的处理，标记为 ⊥ 的单元格不适用。

我们的翻译使用 Rust 标准库中的函数，如下所示：

1. 当 `Option<Box<T>>` 被传递给 BoxCtxt 时，我们预期会发生移动，因此使用 `take` 将选项中的值替换为 None；2. 我们使用 `as_deref` 和 `as_deref_mut`，以避免消耗原始选项，并创建对原选项的引用组成的新选项；3. `as_mut`和 `as_ref` 将原始指针转换为引用；4. `Box::from_raw`将原始指针转换为 `Box` 指针。

我们还定义了辅助函数 to_raw，用于将安全指针转换为原始指针：

```
fn 到_原始<T>(b: &可变 Option<Box<T>>) ->*可变 T {b.as_解引用_可
变().map(|b| b作为 *可变 T).unwrap_或(null_可变()) }
```

这里，我们解释 `to_raw` 对于 `Box` 参数的含义（由于 `&mut` 的多态特性，其解释相同，`as_deref_mut`）：

1. 要转换 Option<Box<T>>，我们首先可变借用整个 Option，这由辅助函数的可变借用参数表示。这样做是必要的，因为 Option 不可复制，否则会被消耗；2. as_deref_mut 会将 &mut Option<Box<T>> 转换为 Option<&mut T>；3. map 会将引用中的可选部分转换为裸指针选项；4. 最后，unwrap_or 会返回该选项中的 Some 值；如果值为 None，则返回空指针 std::ptr::null_mut()。

解引用：当指针 p 在较大的表达式中被解引用时（例如 (*p).next），我们需要额外调用一次 unwrap()。

Box 指针检查： Rust 不允许在 `Box` 指针失去所有权后继续使用它们。由于所有权分析无法捕获此规则，因此会在翻译阶段检测这类情况，并将有问题的 `Box` 指针恢复为裸指针。

为简洁起见，我们省略了对非指针类型的结构体字段稍有不同的处理方式。

7 处理真实世界代码的挑战

我们设计 Crown，使其能够分析和翻译真实世界代码，而这带来了重大挑战。在本节中，我们将讨论 Crown 的一些工程挑战及其当前局限性。

7.1 预处理

在转译 C 库时，c2rust 会将每个文件视为独立的编译单元，并将其翻译成独立的 Rust 模块。因此，结构体定义会重复，而可用的函数定义则放入 extern 块中 [14]。我们会执行一个类似于 Laertes 的 resolve-imports 工具 [14] 的预处理步骤，以链接不同文件中的这些定义。

7.2 所有权分析的局限性

我们的实现无法完全支持一些 C 语言结构和惯用写法，因此 Crown 只能为其生成部分所有权约束。只要存在涉及某个变量的约束，Crown 的翻译就会尝试重写该变量。因此，从理论上说，这种翻译既不可靠，也不完备：它可能生成无法编译的代码（不过我们在实践中尚未观察到这种情况，见第 8 节），也可能让某些指针保留为原始指针，导致翻译结果不够理想。下面列出可能出现这种情况的情形。

某些不安全的 C 构造。对于类型转换，我们只为头指针生成所有权转移约束；对于联合体，我们假设其中不包含指针字段，因此不生成任何约束；类似地，我们也不为可变参数生成任何约束。我们注意到，联合体和可变参数可能导致我们的工具崩溃（例如，[14], 中提到的第 8 节中的三个基准测试）。当分析包含联合体字段解引用的访问路径（我们假设其中不包含指针字段）时，以及分析调用带有可变参数的函数且传入指针作为参数的调用时，就会发生这些崩溃。

函数指针。Crown 不会为它们生成任何约束。

C 库中的非标准内存管理。某些 C 库会封装 malloc 和 free，通常使用静态函数指针（指向分配器/释放器的指针存储在静态变量中），或使用结构体中的函数指针。Crown 不会在此类情形下生成任何约束。在 C 语言中，也可以使用 malloc 分配一大块内存，然后将其拆分为若干子区域，分别赋给不同的指针。在我们的所有权分析中，只有一个指针能够获得通过调用 malloc 分配的内存的所有权。另一种我们尚未完全支持的 C 惯用法是：某些指针可能指向堆上分配的对象，也可能指向静态分配的栈数组。Crown 只为堆生成所有权约束，因此这些变量的约束将不够充分。

7.3 Crown 的其他局限性

数组指针。对于数组指针，尽管 Crown 推断出了正确的所有权信息，但它不会生成合成 Rust 代码所需的元数据。

可变局部借用。如第 6.1 节最后一段所述，Crown 不会将可变的非所有权指针转换为局部可变引用，因为这需要对生命周期进行专门分析。请注意，Crown 确实会为输出参数生成可变引用。

破坏所有权单调性的访问路径。如第 5.4 节所述，在某些情况下，所有权单调性的要求可能比 Rust 语义更严格。

8 实验评估

我们基于 Rust 编译器（版本 nightly-2023-01-26）实现了 Crown。我们使用 0.16.0 版本的 c2rust。对于 SAT 求解器，我们使用 z3 [16] 的 Rust 绑定，版本为 0.11.2。所有实验均在配备 Apple M1 芯片的 MacBook Pro 上运行，该电脑有 8 个核心（4 个性能核心和 4 个能效核心）。电脑配有 16GB 运行内存，运行 macOS Monterey 12.5.1。

基准测试选择。为评估 Crown 的效用，我们收集了一个包含 20 个程序的基准测试套件（表 1）。其中包括 Laertes [14] 随附制品 [13] 中的基准测试（表 1 中以 * 标记）[4]，，以及另外 8 个实际项目（binn、brotli、buffer、heman、json.h、libtree、lodepng、rgba）和 4 个常用数据结构库（avl、bst、ht、四叉树）。

表 1：基准测试信息

基准测试	文件	结构体	函数	代码行	基准测试	文件	结构体	函数	代码行
avl	1	2	11	229	libcsv*	1	6	23	976
binn	1	5	165	4426	libtree	1	18	32	2610
Brotli	30	237	867	537723	libzahl*	49	65	108	4655
bst	1	1	6	154	lil*	2	9	136	5670
buffer	2	3	42	1207	lodepng	1	19	236	14153
bzip2*	9	39	126	14829	四叉树	5	14	31	1216
genann*	6	10	27	2410	RGBA	2	3	19	1855
heman	24	52	302	13762	robotfindskitten*	1	8	18	1508
ht	1	3	10	264	tulipindicators*	111	18	229	22363
json.h	1	13	53	3860	URL 解析器*	1	1	21	1379

8.1 研究问题

我们旨在回答以下研究问题。

RQ1. Crown 能将多少原始指针/指针用法转换为安全指针/指针用法? RQ2. Crown的结果与最先进水平 [14]相比如何? RQ3. Crown的运行时性能如何?

研究问题 1：不安全指针减少。为了评估 Crown的效能，我们测量原始指针声明和使用的减少率。这可以直接反映安全性的提升，因为安全指针始终会经过 Rust 编译器的检查（即使位于不安全区域内）。如前所述，我们关注可变非数组指针。结果见表 2，其中 #ptrs 表示给定基准测试中的原始指针声明数量，#uses 表示原始指针的使用次数，而 Laertes 和 Crown 列标题表示原始指针数量的减少率

[4]我们排除了 json-c、optipng 和 tinycc，因为如第 7 节所述， Crown 会因使用联合体和可变参数而崩溃。论文 [14] 还提到了其他程序（qsort、grabc、xzoom、snudown、tmux、libxml2），但这些程序在制品 [13]中缺失或不完整。

以及这两种工具分别实现的原始指针使用减少率。例如，对于基准测试 avl，100% 表示所有原始指针声明及其使用都被转换为安全形式。请注意，robotfindskitten 对应行中的“-”符号是因为该基准测试中原始指针使用次数为 0。

Crown 将原始指针和原始指针使用的中位数减少率分别提升至 37.3% 和 62.1%。 Crown 在许多非平凡数据结构（ avl 、 bst 、 buffer 、 ht ）以及 RGBA 上都实现了 100% 的减少率。对于由谷歌开发的无损数据压缩算法 Brotli（也是我们规模最大的基准测试）， Crown 实现的减少率分别为 21.4 % 和 20.9%。Brotli 以及其他少数几个基准测试（tulipindicators、lodepng、bzip2、genann、libzahl）的减少率相对较低，这是因为它们采用了非标准的内存管理策略（详见第 7 节）。

值得注意的是，所有翻译后的基准测试都能在上述版本的 Rust 编译器下编译通过。为检验语义保持情况，对于提供测试套件的基准测试（libtree、RGBA、四叉树、URL 解析器、genann、buffer），翻译后的基准测试均通过了所有提供的测试。

表 2：可变、非数组原始指针声明及其使用的减少情况

基准测试	#指针	Laertes	Crown	#uses	Laertes	Crown
avl	8	0.0%	100.0%	41	0.0%	100.0%
binn	103	46.6%	65.0%	247	62.3%	71.3%
Brotli	846	0.0%	21.4%	3686	0.0%	20.9%
bst	5	0.0%	100.0%	22	0.0%	100.0%
buffer	38	0.0%	100.0%	56	0.0%	100.0%
bzip2*	126	14.3%	26.2%	2946	2.2%	**3.7%**
genann*	28	0.0%	**7.1%**	160	0.0%	15.0%
heman	360	30.3%	35.0%	926	50.2%	60.2%
ht	6	33.3%	100.0%	28	42.9%	100.0%
json.h	128	2.3%	23.4%	647	1.2%	62.1%
libcsv*	20	65.0%	70.0%	141	97.9%	97.9%
libtree	48	29.2%	39.6%	227	33.0%	62.1%
libzahl*	87	2.2%	16.1%	279	4.1%	16.8%
lil*	202	9.2%	18.8%	1018	51.4%	69.4%
lodepng	227	46.3%	44.9%	1232	40.4%	37.7%
四叉树	33	0.0%	42.4%	117	0.0%	48.7%
rgba	6	83.3%	83.3%	12	100.0%	100.0%
robotfindskitten*	1	0.0%	0.0%	0	-	-
tulipindicators*	134	0.0%	**0.7%**	625	0.0%	0.0%
URL 解析器*	9	0.0%	11.1%	40	0.0%	45.0%

研究问题 2：与最先进水平进行比较。 Crown与 Laertes [14] 的比较结果也列于表 2 中，粗体标出了较好的结果。Laertesis 的数据要么直接取自制品 [13]，要么已通过与作者的私人通信得到确认。可以看到， Crown 在大多数情况下都优于最先进水平（通常优势显著），唯一的例外是 lodepng，我们怀疑

原因也在于前文提到的非标准内存管理策略。Laertes 受此影响较小，因为它不依赖所有权分析。

RQ 3：运行时性能。尽管我们的分析依赖于求解一个已被证明为 NP完全的约束满足问题，但在实际应用中， Crown 的运行时性能始终很高。对于整个基准测试套件，分析和重写的执行时间均在 60 秒以内。

9 相关工作

所有权分析。所有权这一概念本身并不新鲜。在面向对象编程中，所有权类型系统通过限制面向对象程序 [9, 10] 运行时堆底层的对象图，实现受控别名。人们已尝试自动推断所有权类型 [1, 2, 32], ，并将所有权类型应用于内存管理 [3,33]。类似地，所有权的概念也被用于分析 C 语言/C++ 程序。Heine 等人 [19] 推断指针所有权类型，以检测内存泄漏。他们对所有权转移约束的编码对我们影响很大。Ravitch 等人 [30] 运用静态分析推断所有权，以自动生成库绑定。我们的输出参数概念受这项工作启发。由于应用领域不同，这些工作各自采用了不同的假设。Heine 等人 [19] 假设无法通过间接方式访问的指针不能获得所有权，而 Ravitch 等人 [30] 假设所有结构体字段均拥有所有权，除非另有明确注释。我们的分析不受这些假设限制，因此能够进行更精确的分析，从而合成正确的 Rust 代码。最后，所有权的思想也广泛应用于并发分离逻辑 [5–7, 15, 31]。Iris 框架 [23] 也被用于形式化 Rust 类型系统 [22] ，并验证 Rust 程序 [29]。

类型限定符。类型限定符是一种轻量且实用的机制，用于指定和检查传统类型系统无法涵盖的属性。研究者提出了一种通用的流不敏感类型限定符框架 [17], ，随后将其用于分析 Java 引用可变性 [18, 20] 和 C 数组边界 [27]。我们对 Crown 的可变性和胖指针属性所做的分析深受这些工作的影响。

C 到 Rust 的转换。我们已经介绍过 c2rust [21], ，这是一款工业级工具，可将 C 语言转换为 Rust 语法。c2rust 不会尝试修复原始指针等不安全特性，而且它生成的程序始终会标记为不安全。尽管如此，它仍是其他转换工作的基础。CRustS [26] 通过基于 AST 的代码转换，移除 c2rust 生成的多余不安全标记。但它也不会修复不安全特性。Laertes [14] 是首个真正能够自动减少不安全代码的工具。它将 Rust 编译器作为黑盒预言机，并搜索能够移除原始指针的代码变更；这种方法与 Crown 截然不同（实验比较见第 8 节）。

10 结论

我们针对 c2rust 转换的 Rust 程序设计了一种所有权分析。通过强化 Rust 所有权模型（我们称之为所有权单调性），该分析既具备可扩展性（不到 10 秒即可处理 50 万行代码），又具有精确性（能够处理归纳数据结构）。基于这一新分析，我们开发了一个将 C 程序转换为 Rust 程序的原型重构工具。实验评估表明，所提出的方法能够处理真实世界的基准测试，并优于最先进水平。

引用

1. Aldrich, J., Kostadinov, V., Chambers, C.：《用于程序理解的别名注解》。载于：第17届 ACM SIGPLAN 面向对象编程、系统、语言与应用会议论文集，第311–330页。OOPSLA '02，计算机协会，纽约，美国（2002年）。https://doi.org/10.1145/582419.582448, https://doi.org/10 582448 2 1145 582419. / . . Boyapati, C., Liskov, B., Shrira, L.：《用于对象封装的所有权类型》。载于：第 30 届 ACM SIGPLAN-SIGACT编程语言原理研讨会论文集，第213–223页。POPL '03，计算机协会，纽约，美国（2003年）。https://doi.org/10.1145/604131.604156, https://doi.org/10 604156 3 1145 604131. / . . Boyapati, C., Salcianu, A., Beebee, W., Rinard, M.：《用于实时 Java 安全基于区域的内存管理的所有权类型》。载于：ACM SIGPLAN 2003编程语言设计与实现会议论文集，第324–337页。PLDI '03，计算机协会，纽约，美国（2003年）。https://doi.org/10.1145/781131.781168, https://doi.org/10 781168 4 1145 781131. / . . Briggs, P., Cooper, K.D., Harvey, T.J., Simpson, L.T.：《静态单赋值形式构造与销毁的实用改进》。软件实践与经验。28(8), 859–881（1998年7月）5. Brookes, S.：《共享内存程序中的变量作为资源：语义与可靠性》。理论计算机科学电子札记。158，123–150（2006年5月）。https://doi.org/10.1016/j.entcs.2006.04.008, https://doi.org/10 008 6 1016 2006 04 . / j.entcs. . . . Brookes, S.：《并发分离逻辑的语义》。理论计算机科学。375(1–3)，227–270（2007年4月）。https://doi.org/10.1016/j.tcs.2006.12.034, https://doi.org/10 034 7 1016 2006 12 . /j.tcs. . . . Calcagno, C., O'Hearn, P.W., Yang, H.：《局部动作与抽象分离逻辑》。载于：第22届 IEEE 逻辑与计算机科学年度研讨会论文集，第366–378页。LICS '07，IEEE计算机学会，美国（2007年）。https://doi.org/10 30 8 1109 2007 30, 10 1109 2007 . /LICS. . https://doi.org/ . /LICS. . . Cheng, B.C., Hwu, W.M.W.：《使用访问路径的模块化过程间指针分析：设计、实现与评估》。载于：ACM SIGPLAN 2000编程语言设计与实现会议论文集，第57–69页。PLDI '00，计算机协会，纽约，美国（2000年）。https://doi.org/10.1145/349299.349311, https://doi.org/10.1145/349299.349311

9. Clarke, D., ¨Ostlund, J., Sergey, I., Wrigstad, T.：《所有权类型：综述》。第7850卷，第15–58页。施普林格（2013年）。https://doi.org/https://doi.org/10.1007/978-3-642-36946-9-3, https://lirias.kuleuven.be/1630873 10. Clarke, D.G., Potter, J.M., Noble, J.：《用于灵活别名保护的所有权类型》。载于：第13届 ACM SIGPLAN 面向对象编程、系统、语言与应用会议论文集，第48–64页。OOPSLA '98，计算机协会，纽约，美国（1998年）。https://doi.org/10.1145/286936.286947, https://doi.org/10 286947 11 1145 286936. / . . Das, M.：《具有定向赋值的基于统一的指针分析》。载于：ACM SIGPLAN 2000编程语言设计与实现会议论文集，第35–46页。PLDI '00，计算机协会，纽约，美国（2000年）。https://doi.org/10.1145/349299.349309, https://doi.org/10 349309 12 1145 349299. / . . De, A., D'Souza, D.：《带强更新的 Java 可扩展流敏感指针分析》。载于：Noble, J.（编）ECOOP 2012——面向对象编程。第665–687页。施普林格柏林海德堡，柏林，海德堡（2012年）13. Emre, M., Schroeder, R.：《"将 C 语言转换为更安全的 Rust"的制品》（2021年9月）。https://doi.org/10.5281/zenodo.5442253, https://doi.org/10 5442253 14 5281. /zenodo. . Emre, M., Schroeder, R., Dewey, K., Hardekopf, B.：《将 C 语言转换为更安全的 Rust》。ACM 编程语言论文集。5(OOPSLA)，1–29（2021年）。https://doi.org/10 3485498 15 1145 3485498, 10 1145 . / https://doi.org/ . / . Feng, X.：《局部依赖-保证推理》。载于：第36届 ACM SIGPLAN-SIGACT编程语言原理研讨会论文集，第315–327页。POPL '09，计算机协会，纽约，美国（2009年）。https://doi.org/10.1145/1480881.1480922, https://doi.org/10 1480922 16 1145 1480881 . / . . Fitzgerald, N., Hoare, G., Mitchener, B., Puri, S.：《 Z3 SMT 求解器的 Rust 绑定》。https://crates.io/crates/z3 17. Foster, J.S., Johnson, R., Kodumal, J., Aiken, A.：《流不敏感类型限定符》。ACM 编程语言与系统汇刊。28(6)，1035–1087（2006年11月）。https://doi.org/10.1145/1186632.1186635, https://doi.org/10 1186635 18 1145 1186632 . / . . Greenfieldboyce, D., Foster, J.S.：《 Java 类型限定符推断》。载于：第 22 届 ACM SIGPLAN 面向对象编程、系统、语言与应用会议论文集，第321–336页。OOPSLA '07，计算机协会，纽约，美国（2007年）。https://doi.org/10.1145/1297027.1297051, https://doi.org/10 1297051 19 1145 1297027 . / . . Heine, D.L., Lam, M.S.：《实用的流敏感、上下文敏感 C 与 C++ 内存泄漏检测器》。载于：Cytron, R., Gupta, R.（编）ACM SIGPLAN 2003编程语言设计与实现会议论文集，2003年6月9–11日，美国加利福尼亚州圣迭戈，第168–181页。ACM（2003年）。https://doi.org/10.1145/781131.781150, https://doi.org/10 781150 20 1145 781131. / . . Huang, W., Milanova, A., Dietl, W., Ernst, M.D.：《Reim 与 reiminfer：引用不可变性和方法纯度的检查与推断》。载于：ACM 面向对象编程、系统、语言与应用国际会议论文集，第879–896页。OOPSLA '12，计算机协会

计算机协会，纽约，纽约州，美国（2012年）。
https://doi.org/10.1145/2384616.2384680, https://doi.org/10 2384680 21 1145
2384616 2 . / . . inc., I.: c rust. https://github.com/immunant/c2rust 22. Jung, R.,
Jourdan, J.H., Krebbers, R., Dreyer, D.: Rustbelt：确保 Rust 编程语言基础的安全。
Proc. ACM Program. Lang. 2(POPL)（2017年12月 3158154 23 10 1145 3158154, 10
1145）。https://doi.org/ . / https://doi.org/ . / . Jung, R., Krebbers, R., Jourdan, J.H.,
Bizjak, A., Birkedal, L., Dreyer, D.: Iris 从基础构建：高阶并发分离逻辑的模块化基础。
《函数式编程杂志》28(e20)（2018年）。
https://doi.org/10.1017/S0956796818000151, https://hal.science/hal-
01945446
24 ¨ . Lerch, J., Spath, J., Bodden, E., Mezini,
M.: 访问路径抽象：通过无界访问路径扩展字段敏感数据流分析（t）。载于：2015年
IEEE/ACM 自动化软件工程国际会议（ASE）第30届会议。第
619 629 2015 10 1109 2015 9 – 页（）。
25 ´
https://doi.org/ . /ASE. . . Lhotak, O., Chung, K.C.A.: 通过高效强更新进行指向分析。载
于：第38届 ACM SIGPLAN-SIGACT 编程语言原理年度研讨会论文集。第3–16页。
POPL '11，计算机协会，纽约，纽约州，美国（2011年）。
https://doi.org/10.1145/1926385.1926389, https://doi.org/10 1926389 26 1145
1926385 . / . . Ling, M., Yu, Y., Wu, H., Wang, Y., Cordy, J.R., Hassan, A.E.: 我们信任
Rust——一种将不安全的 C 转换为更安全 Rust 的源到源编译器。载于：第44届
IEEE/ACM 国际软件工程会议：会议伴随论文集，ICSE Companion 2022，24，2022
22 354 355 2022 10 1145 3510454 3528640，美国宾夕法尼亚州匹兹堡，5月 - 。第 – 页。
ACM/IEEE（）。https://doi.org/ . / . https://doi.org/10 3528640 27 1145 3510454 . / . .
Machiry, A., Kastner, J., McCutchen, M., Eline, A., Headley, K., Hicks, M.: 通过 3c 将
C 转换为经检查的 C。Proc. ACM Program. Lang. 6(OOPSLA1)（2022年4月）。
https://doi.org/10 3527322 28 1145 3527322, 10 1145 . / https://doi.org/ . / .
Matsakis, N.D., Klock, F.S.: Rust 语言。载于：2014年 ACM SIGAda 高完整性语言技术
年度会议论文集。第103–104页。HILT '14，计算机协会，纽约，纽约州，美国（2014年）。
https://doi.org/10.1145/2663171.2663188, https://doi.org/10 2663188 29 1145
2663171 . / . . Matsushita, Y., Denis, X., Jourdan, J.H., Dreyer, D.: Rusthornbelt：为包
含不安全代码的 Rust 程序函数式验证提供语义基础。载于：第43届 ACM SIGPLAN 编程
语言设计与实现国际会议论文集。第841–856页。PLDI 2022，计算机协会，纽约，纽约
州，美国（2022年）。https://doi.org/10.1145/3519939.3523704, https://doi.org/10
3523704 30 1145 3519939 . / . . Ravitch, T., Jackson, S., Aderhold, E., Liblit, B.: 使用静
态分析自动生成库绑定。载于：Hind, M., Diwan, A.（编）2009年 ACM SIGPLAN 编程
语言设计与实现会议论文集，PLDI 2009，爱尔兰都柏林，2009年6月15日至21日。第
352–362页。ACM（2009年）。https://doi.org/10.1145/1542476.1542516,
https://doi.org/10 1542516 31 1145 1542476 . / . . Vafeiadis, V., Herlihy, M., Hoare,
T., Shapiro, M.: 证明高度并发线性化对象的正确性。载于：第十一届 ACM SIGPLAN 并
行编程原理与实践研讨会论文集，

第129–136页。PPoPP '06，计算机协会，纽约，纽约州，美国（2006年）。
https://doi.org/10.1145/1122971.1122992, https://doi.org/10.1145/1122971.1122992 32.
Wolff, F., B´ıl´y, A., Matheja, C., M¨uller, P., Summers, A.J.: Rust 中闭包的模块化规范与验证。Proc. ACM Program. Lang. 5(OOPSLA)（2021年10月 3485522 33 10 1145 3485522, 10 1145）。https://doi.org/ . / https://doi.org/ . / . Zhao, T., Baker, J., Hunt, J., Noble, J., Vitek, J.: 用于内存管理的隐式所有权类型。《计算机程序科学》71(3)，213–241（2008年）。
https://doi.org/https://doi.org/10.1016/j.scico.2008.04.001,
https://www.sciencedirect.com/science/article/pii/S0167642308000300

A 所有权约束

所有权单调性、函数体和函数调用的规则，以及用于建模 C 库函数的部分规则，见图 4。

单调规则。当生成新的所有权变量时， Crown 会生成一组约束，以强制满足所有权单调性，如图 4 中的 MONOTONE 规则所示。在该规则中， V 表示约束变量集， C 表示约束集， Σ 表示以相关所有权变量表示的函数签名。根据该规则，如果 p 和 q 都是基变量为 v 的访问路径，且 p 是 q 的前缀，那么 p 的所有权大于或等于 q 的所有权。例如，Crown 针对图 1b 生成如下内容： $\mathbb{O}_{\texttt{new\_node}} \geq \mathbb{O}_{\texttt{(*new\_node).next}}$。

为简单起见，只有在规则中使用时，我们才显式写出 V 或 Σ 。此外，我们假设每当使用指针时，都会生成新的变量并应用 MONOTONE 规则。

FREE。当指针传递给 free 时，我们会生成约束，断言该指针在调用前拥有所有权，而在调用后不拥有所有权。图 4。

FN-DECL。函数声明的约束生成规则如 FN-DECL 所示。我们考虑一个函数 f，它有输出参数 ˜x、普通参数 ˜y 和函数返回值 z。该规则指出，给定当前的所有权变量 V、约束 C 和函数签名 Σ，函数声明的推断会为函数体中的语句生成所有权约束，并相应更新 V'、 C'、 Σ' 。对于输出参数，我们需要生成两组所有权变量，一组用于函数入口，另一组用于函数出口，以表示其输入/输出所有权状态，然后将它们约束为相同。

函数调用规则。 函数调用规则如函数调用规则所示。正如第 5.4 小节所述，我们会在输出参数位置显式地将指针 p 转换为 `&mut (*p)`。此处假设所有输出参数的调用实参均采用 `&mut p` 的形式。该规则指出，对于普通参数，实参的所有权可以选择转移 给参数，如三变量约束所示；对于输出参数，实参的所有权

会借用给参数：参数的进入/退出状态分别与实参的前置/后置状态相等。

$$\text{MONOTONE}\quad \frac{a = base\_var(v) \qquad p, q \in ap(v, 1, k) \qquad is\_prefix(p, q) \qquad C' = C \cup \{\mathbb{O}_p \geq \mathbb{O}_q\} \qquad V' = V \cup \{\mathbb{O}_p \text{新建} | p \in ap(a, 1, k)\}}{V, C \vdash \text{v 单调} \Rightarrow V' \qquad , C'}$$

$$\text{FREE}\quad \frac{C' = C \cup \{\mathbb{O}_v = 1 \wedge \mathbb{O}_{v'} = 0\}}{C \vdash \text{free(v)} \Rightarrow C'}$$

$$\text{FN-DECL}\quad \frac{\begin{array}{c}\vec{v} \in \text{局部变量}(\ \vec{}\text{语句}\ ) \\ V' = V \cup \{\mathbb{O}_l | l \in ap(\vec{x}_{\text{entry}}, 1, k) \cup ap(\vec{x}_{\text{exit}}, 1, k) \cup ap(\vec{y}, 1, k) \cup ap(z, 1, k)\} \\ C' = C \cup \left\{\mathbb{O}_{\vec{x}}^{\text{entry}} = 1, \mathbb{O}_{\vec{x}}^{\text{exit}} = 1\right\} \cup \{\mathbb{O}_r = 0 | r \in ap(\vec{v}_{\text{exit}}, 1, k)\}\end{array} \qquad \Sigma' = \Sigma \cup \{\text{f}(\ \vec{}\text{x};\ \vec{}\text{y}) : \text{z}\} \qquad V', C', \Sigma' \vdash \text{语句} \Rightarrow V'', C''}{V, C, \Sigma \vdash \text{f}(\ \vec{}\text{x};\ \vec{}\text{y}) : \text{z}\ \{\ \vec{}\text{语句}\} \Rightarrow V'', C'', \Sigma'}$$

$$\text{FN-CALL} f(\vec{x}; \vec{y}):\quad \frac{\begin{array}{l} z \in \Sigma C' = C \cup \{\mathbb{O}_{r'} + \mathbb{O}_l = \mathbb{O}_r | r \in ap(q, 1, k) \quad l \in ap(y, 1, k) \quad |r| - |q| = |l| - |y|\} \cup \{\mathbb{O}_s = \mathbb{O}_n, \mathbb{O}_{s'} = \mathbb{O}_{n'} | s \in ap(p, 1, k) \quad n \in ap(x, 2, k) \\ |s| - |p| = |n| - |x| - 1\} \cup \{\mathbb{O}_t = 0 \wedge \mathbb{O}_{t'} = \mathbb{O}_m | t \in ap(r, 1, k) \quad m \in ap(z, 1, k) \\ |t| - |r| = |m| - |z|\} C, \Sigma \vdash \text{let r} \quad = \text{f}(\&\text{mut} \quad \vec{}\text{p}; \vec{}\text{q}) \Rightarrow C'\end{array}}{}$$

图 4：选定的所有权规则

B 定理 2 的证明

考虑 p 和 q之间的一次指针赋值。
(i) 如果没有发生所有权转移，则结论可由假设得出。

(ii) 如果所有权从 q 转移到 p，那么赋值后 p' 拥有一个新对象。若另一个指针 l 在赋值后拥有同一个内存对象，则在赋值前， l 必定是 p 的别名。现在考虑 p 和 l的所有可能访问路径：

– 两者都通过直接访问： $p = q$； l。在这种情况下，赋值后 p' 和 l' 不再互为别名，这意味着它们不能拥有同一个对象。– p 通过间接方式访问， l 通过直接方式访问： $*p = q$； l。如果 $*p$ 是 l的别名，那么，同样地，
$*p'$ 和 l' 在赋值后不再是别名。

– 两者都是间接访问的：$*p = *q$；$*l$。在这种情况下，我们还需要考虑 p 和 l之间可能存在的别名。

- 如果 $*p$ 与 $*l$互为别名，但 p 与 l不是别名，那么赋值后，$*p'$ 和 $*l'$ 也不再是别名。
- 如果 $*p$ 与 $*l$ 互为别名，且 p 与 l互为别名，那么赋值后，$*p'$ 和 $*l'$ 就是别名。现在，为了检查它们是否都可以拥有所有权，我们来看看所有权约束。我们知道 $*p'$ 拥有所有权，这意味着 $\mathbb{O}_{*p'} = 1$。根据所有权单调性，我们有 $\mathbb{O}_{p'}$ =1。鉴于 p' 与 l'互为别名，根据假设，我们知道 l' 不能拥有同一个对象，因此 $\mathbb{O}_{l'} = 0$。根据所有权单调性，我们有 $\mathbb{O}_{*l'} = 0$。因此，$*l'$ 不能拥有所有权

– 更长的访问路径的证明与上文间接访问 p 时相同，因为只需考察访问路径上的最后一次间接访问。

C 定理 1 的证明

证明。令 a =base var(v)，b =base var(w)。假设 $p \in$ ap ($a, |v|, k$)、$q \in$ ap ($b, |w|, k$)，且 $|p| - |v| = |q| - |w|$。根据 ASSIGN 规则，我们有 $\mathbb{O}_p = 0 \wedge \mathbb{O}_{p'} + \mathbb{O}_{q'} = \mathbb{O}_q$。根据单调性规则，我们有 $\mathbb{O}_{w'} \geq \mathbb{O}_{q'}$。根据假设，所有权从 w 转移到 v，因此 $\mathbb{O}_{w'} = 0$，这意味着 $\mathbb{O}_{q'} = 0$，且 $\mathbb{O}_{p'} = \mathbb{O}_q$。这意味着，如果 q 在赋值之前拥有所有权，那么随后它将被转移到 p。

D 定理3的证明

证明。我们的证明分为两部分：

(1) 约束求解属于 NP。我们通过在多项式时间内将其归约为可满足性问题来证明这一点。具体而言，每个等式都可以按如下方式转换：

– $x + y = z$ 被翻译为 $(z \wedge x \wedge \neg y) \vee (z \wedge \neg x \wedge y) \vee (\neg z \wedge \neg x \wedge \neg y)$，其合取范式为 $(z \vee y \vee \neg x) \wedge (x \vee \neg y) \wedge (z \vee x \vee \neg y) \wedge (\neg y \vee \neg z) \wedge (x \vee \neg z) \wedge (x \vee y \vee \neg z)$。– $x = 1$ 被翻译为 x。– $x = 0$ 被翻译为 $\neg x$。– $x \leq y$ 被翻译为 $(\neg x \wedge \neg y) \vee (x \wedge \neg y) \vee (x \wedge y)$

（2）约束求解是NP难的。我们通过在多项式时间内将 EXACT- 1-3-SAT 问题归约为所有权约束求解来证明这一点。EXACT- 1-3-SAT 是这样一个问题：判断是否存在一种解释，使给定的布尔公式成立。该公式由三文字子句的合取构成，并额外要求每个子句中恰好有一个文字为真。

接下来，我们将介绍如何构造所有权约束系统。对于每个子句 $l_1 \vee l_2 \vee l_3$，我们在约束系统中生成三个等式：$l_1 + l_2 = not\_l_3$、$l_3 + not\_l_3 = l_4$ 和 $l_4 = 1$，其中 not_l_3 和 l_4 是新变量。于是，原问题可满足，当且仅当所有权约束系统有解。注意，对于一个子句，唯一可满足的配置

$l_1 \vee l_2 \vee l_3$ 是 (1, 0, 0)、(0, 1, 0) 和 (0, 0, 1)，这与 $l_1 + l_2 = not\_l_3$、$l_3 + not\_l_3 = l_4$ 和 $l_4 = 1$ 的情况完全相同；此外，$not\_l_3 = \neg l_3$ 和 $l_4 = 1$。另外，我们知道每个所有权变量的值只能是 0 或 1。