박 사 학 위 논 문
Ph.D. Dissertation

정적 분석을 통한 C-러스트 번역의 개선

Improving C-to-Rust Translation with Static Analysis

2025

홍 재 민 (洪 濟 旻 Hong, Jaemin)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

박 사 학 위 논 문

정적 분석을 통한 C-러스트 번역의 개선

2025

홍 재 민

한 국 과 학 기 술 원

전산학부

정적 분석을 통한 C-러스트 번역의 개선

홍 재 민

위 논문은 한국과학기술원 박사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2024년 11월 19일

심사위원장	류 석 영	(인)
심 사 위 원	강 지 훈	(인)
심 사 위 원	유 신	(인)
심 사 위 원	오 학 주	(인)
심 사 위 원	Ben Hardekopf	(인)

Improving C-to-Rust Translation with Static Analysis

Jaemin Hong

Advisor: Sukyoung Ryu

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Daejeon, Korea
November 19, 2024

Approved by

Sukyoung Ryu
Professor of School of Computing

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

DCS 홍재민. 정적 분석을 통한 C-러스트 번역의 개선. 전산학부 . 2025년. 161+viii 쪽. 지도교수: 류석영. (영문 논문)
Jaemin Hong. Improving C-to-Rust Translation with Static Analysis. School of Computing . 2025. 161+viii pages. Advisor: Sukyoung Ryu. (Text in English)

초 록

러스트는 시스템 프로그래밍에서 C를 대체하도록 설계된 최신 언어로, 메모리 버그의 부재를 보장하는 강력한 타입 체계를 제공한다. 따라서 기존 시스템 프로그램을 C에서 러스트로 번역하는 것은 프로그램의 신뢰도를 높이는 유망한 방법이다. 이때 사람이 직접 코드를 번역하면 큰 비용이 들기 때문에 C-러스트 번역을 자동화하는 것이 바람직하다. 그러나 현존하는 번역기 중 가장 잘 알려진 C2Rust는 문법적으로 코드를 번역하여 모든 C 기능을 생성한 러스트 코드에 그대로 유지한다. C 기능이 러스트 코드에 사용되면, 컴파일러가 안전성을 보장할 수 없고 코드 패턴이 러스트의 이디엄을 따르지 않아 번역의 목적에 위배된다. 기존 연구는 C2Rust가 생성한 코드에서 C 기능을 대응되는 러스트 기능으로 대체하는 기술을 탐구했지만, 전체 기능 중 극히 일부만 다루었다. 본 학위 논문은 정적 분석을 통해 C-러스트 번역을 개선하는 기술을 제시하며, 이전에 다루지 않은 다양한 기능을 대상으로 한다. 제안한 기술은 각 C 기능에 특화된 정적 분석을 통해 해당 기능이 코드에서 어떻게 사용되는지 알아내고 그 정보를 코드 변형에 활용한다. 구체적으로는 락, 태그 붙은 유니언, 출력 매개변수라는 세 가지 중요한 C 기능을 다룬다. 또한, 이 기법들을 보완하고자 대규모 언어 모델을 사용한 번역 방법도 제시한다. 더 나아가, 대응되는 러스트 기능이 아직 개발되지 않은 C 기능을 위해 모듈화된 요약을 제시하여, 해당 기능을 대상으로 하는 정적 분석의 추후 개발을 가능케 한다.

핵 심 낱 말 러스트, C 언어, 자동 번역, 정적 분석, 대규모 언어 모델, 모듈화된 요약

Abstract

Rust, a modern language designed to replace C in system programming, provides a strong type system that ensures the absence of memory bugs. This makes translating legacy system programs from C to Rust a promising approach for enhancing their reliability. Due to the high cost of manual code translation, it is desirable to automate C-to-Rust translation. However, C2Rust, the most well-known existing translator, only performs syntactic translation, retaining all the C features in the resulting Rust code. When C features are used in Rust code, their safety is not ensured by the compiler, and code patterns employing them do not follow Rust idioms, contradicting the goals of translation. Although researchers have studied techniques for replacing C features in C2Rust-generated code with their Rust counterparts, only a small subset of these features has been addressed. In this dissertation, we propose techniques to improve C-to-Rust translation using static analysis, addressing various language features not covered in previous studies. We perform static analysis tailored to each C feature to gather information on how this feature is used in the code, utilizing the results for code transformation. Specifically, we target three important C features: locks, tagged unions, and output parameters. In addition, to complement these approaches, we also propose a translation method using a large language model. Furthermore, we propose modular abstractions for C features whose Rust counterparts have not yet been developed, to facilitate the future development of static analysis targeting these features.

Keywords Rust, C, automatic translation, static analysis, large language model, modular abstraction

Contents

List of Tables

List of Figures

List of Listings

List of Algorithms

Chapter 1. Introduction

C, despite its widespread use in system programming, is notorious for its poor language-level safety mechanisms. C programs can have memory bugs even after passing the type checking of the compiler. As a result, legacy system software developed in C often suffers from memory bugs leading to severe security vulnerabilities. For example, memory bugs account for two-thirds of the vulnerabilities in Linux [69] and approximately 70% of CVE-assigned vulnerabilities in Microsoft's codebase [189]. Recognizing this, the White House recently recommended discouraging the use of C [104].

Various approaches have been proposed to reduce memory bugs in system programs. Fuzzing, which generates random inputs to a program, has been successful in finding bugs [111, 194, 179, 120]. Static analysis approximates program behavior without execution, and several tools focus on analyzing system programs [142, 59]. Formal verification techniques use proof assistants [62, 163] to verify program correctness. Recent advances in these techniques have enabled the verification of complex system programs, even including operating systems (OSs) [124, 50, 94, 158].

However, each of these approaches has drawbacks. Fuzzing cannot detect bugs in unexplored paths, making it insufficient to prove the absence of bugs. Static analysis often generates too many false alarms, hindering its scalability for large, real-world system programs. Lastly, formal verification requires manual proof writing, which is time-consuming even for experts using state-of-the-art proof assistants.

Rust [145, 16], a modern language designed to replace C in system programming, provides a strong type system that ensures the absence of memory bugs for programs that pass type checking [115]. For this reason, writing software in Rust enables the creation of safe-by-construction system programs. The type checker does not miss bugs unlike fuzzing, and its type error messages are easier for developers to understand and fix compared to the alarms from static analysis using complex abstract domains and sensitivities. Additionally, it does not require the manual proof writing of formal verification. Due to these advantages, Rust has been widely adopted in system programming, as shown by the development of critical software such as garbage collectors [136] and OSs [131, 127, 128, 67, 156].

While developing new systems in Rust is beneficial, many legacy programs were written in C before Rust's emergence, leaving them without its safety benefits. Translating them to Rust is a promising way to improve their reliability. After the translation, developers can detect previously unknown bugs through Rust's type checking. Notably, more than half of cURL's security vulnerabilities could have been prevented if it had been written in Rust [105]. In addition, once software is ported to Rust, the risk of introducing new bugs when adding features significantly decreases.

Noticing this potential, the industry has already begun transitioning from C to Rust [110, 107, 45]. In particular, Mozilla developed an alternative web browser, Servo, written in Rust, and has gradually replaced Firefox's C modules with semantically equivalent ones from Servo [53]. According to Mozilla developers, Rust's safety guarantee was crucial for building Servo's complex, parallel engine for CSS styling and GPU rendering with low cost and high confidence [92]. Furthermore, Rust has become one of the implementation languages for Linux [183], Android [187], and Fuchsia [5], Google's OS for smart home devices [68].

However, the current approach of manually translating legacy C code to Rust is labor-intensive and error-prone, limiting wider adoption. Developers must write Rust code that is not only syntactically valid but also semantically equivalent to the original C code. Additionally, the resulting Rust code should

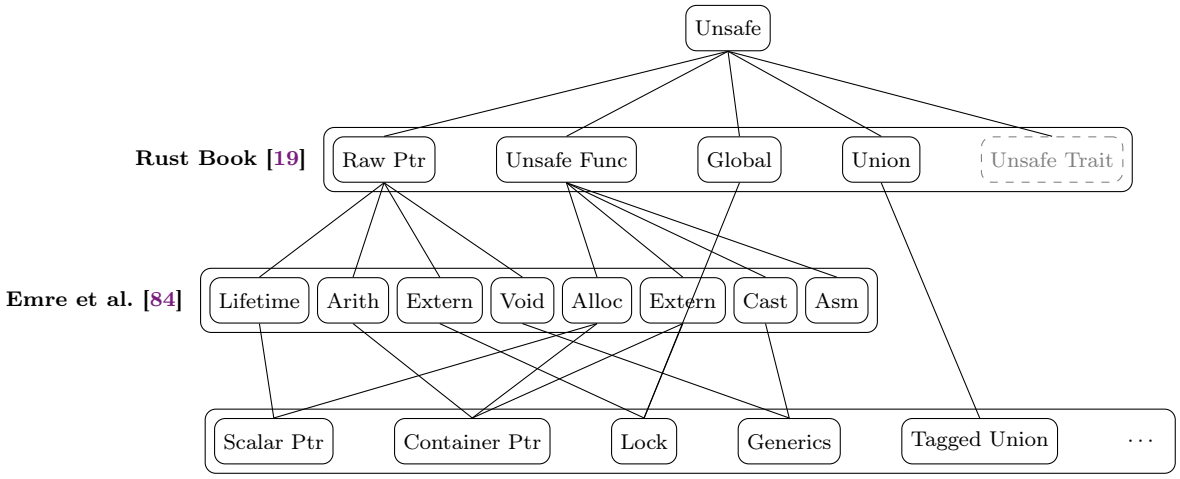


Figure 1.1: Classification of Rust's unsafe features

follow Rust's programming idioms, fully leveraging its features. Rust has adopted many useful features not only from C but also from other modern languages. For instance, its support for algebraic data types, parametric polymorphism, and traits (which resemble Haskell's type classes) is influenced by functional languages. These features help express logic more concisely and clearly, so properly incorporating them significantly impacts translation quality. Therefore, developers must understand how C features are used in the original code and restructure the program to express the intended logic using Rust's features.

To address this problem, developing an automatic C-to-Rust translator is crucial. Such a tool can translate legacy C code to Rust much faster and with far fewer errors than human developers. Recognizing the importance of this effort, the Defense Advanced Research Projects Agency (DARPA) recently announced a program aimed at automating C-to-Rust translation [195].

Unfortunately, existing techniques for automatic C-to-Rust translation remain unsatisfactory. C2Rust [1, 198], the most well-known tool, performs a straightforward syntactic translation, retaining all the C features in the resulting Rust code. However, using C features in Rust is problematic becuase they are either *unsafe* or *unidiomatic*. First, many of C features fall under Unsafe Rust [19], which allows the use of *unsafe features* like dereferencing raw pointers and calling external functions. Since the Rust compiler does not guarantee the safety of unsafe features, their use contradicts the goal of translation. Second, C features often do not align with Rust's idioms, making the code harder to comprehend and maintain. For example, while Rust uses types like `Option` and `Result` to express partial functions and iterators to traverse collections, C relies solely on pointers for these purposes, failing to explicitly convey the intended logic. Therefore, it is essential to replace C features in C2Rust-generated code with their Rust equivalents to improve the quality of automatic translation. Given the variety of C features, it is infeasible to replace all at once. Instead, techniques must be developed for each feature based on a deep understanding of its role.

To better understand C features, we categorize them based on their characteristics. As mentioned earlier, C features can be broadly divided into unsafe features and unidiomatic features. Since most C features belong to unsafe features, we further classify them as shown in Figure 1.1. As illustrated in the second row, Rust has five kinds of unsafe features: raw pointer dereferencing, unsafe function calls, mutable global variable access, union field access, and implementation of unsafe traits [19]. Of these, as unsafe traits do not appear in C2Rust-generated Rust code, they do not correspond to any C features. Additionally, as seen in the third row, Emre et al. [84] further classify some unsafe features. They

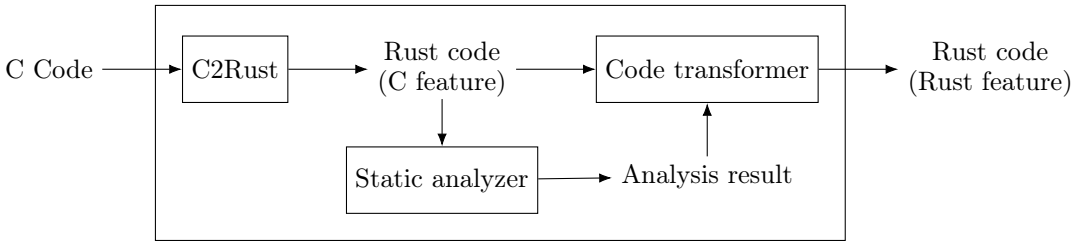


Figure 1.2: The workflow of the proposed approach

categorize raw pointers into lifetime pointers, pointer arithmetic, external pointers, and void pointers and unsafe functions into allocation functions, external functions, casts, and inline assembly. This classification is exhaustive by their category definitions: lifetime pointers include all raw pointers not in other categories, and external functions cover all functions not classified elsewhere.

While this classification provides a useful starting point, we propose reclassifying unsafe features by combining some into broader categories and subdividing others, as shown in the fourth row. This is motivated by (1) the need to replace related features, such as lifetime pointers and allocation functions, with safe counterparts simultaneously, and (2) the observation that some features remain too coarse-grained; for example, external functions serve diverse purposes like concurrency, file systems, and subprocess management, each of which requires distinct techniques. Naturally, this reclassification is subjective and challenging to make exhaustive. While we leave the task of further expanding this list to future studies, we describe some of the important features here:

- *Scalar pointers*: This category involves the use of raw pointers to objects other than collections. It includes lifetime pointers and certain allocation functions. Such pointers can be replaced by references and `Box`, which are pointers to heap-allocated objects.

- *Container pointers*: This category involves the use of raw pointers to collections. It includes pointer arithmetic as well as certain allocation and external functions. Such pointers can be replaced by collection types like `Vec` and `String`, and by slices, which are pointers to (possibly parts of) collections.

- *Locks*: This category involves the use of locks to protect shared data in concurrent programs. It includes certain external pointers, external functions, and global variables. C locks can be replaced by Rust locks, which are containers that combine protected data with C locks.

- *Tagged unions*: This category involves the use of unions to mimic tagged unions. It includes certain unions, which are accompanied by tag values indicating the active fields. Such unions can be replaced by tagged unions, or *enums* in Rust's terminology.

- *Generics*: This category involves functions that mimic generic functions. It includes certain void pointers and casts between function pointer types. Such functions can be replaced by generic functions.

Recognizing the importance of replacing C features in C2Rust-generated code, researchers have explored this area, but only a very small subset of features has been addressed. The only existing studies, Laertes [84, 83] and Crown [203], focus on scalar pointers. However, all other features remain unaddressed, limiting the applicability of automatic translation to real-world C code.

In this dissertation, we propose techniques to improve C-to-Rust translation using *static analysis* while addressing various language features not covered in previous studies. Figure 1.2 illustrates the overall workflow of our approach. First, we translate C code to Rust code that retains C features using C2Rust. Next, we perform static analysis tailored to a specific C feature to automatically gather information on how this feature is used in the code. Finally, we transform the Rust code by replacing the C feature with the corresponding Rust feature based on the analysis results. Since each language feature has distinct characteristics, dedicated static analysis is required for each feature. Specifically, we target three important C features: locks, tagged unions, and output parameters. The first two are unsafe features, while the last is an unidiomatic feature.

In addition, to complement the approaches using static analysis, we also propose translation using a *large language model* (LLM) like ChatGPT [165, 46]. While static analysis is a desirable method for producing correct code, designing proper analysis for each feature requires significant effort. For this reason, this work does not cover all C features through static analysis alone. As a complementary approach, we introduce LLM-based translation, which, though it may not always produce fully correct code, can replace various C features with Rust features. Specifically, we focus on type migration, i.e., replacing C types with Rust types, by leveraging LLMs' intuitive understanding of program semantics and programming idioms, obtained from training on vast collections of human-written code.

Furthermore, we propose *modular abstractions* for C features whose Rust counterparts have not yet been developed, to facilitate the future development of static analysis targeting these features. Rust programmers have constructed modular abstractions for code patterns expressed with unsafe features to enable modular reasoning of the code [55]. Therefore, proper abstractions for C features are essential for enabling their automatic replacement through static analysis. While many patterns in general system software already have modular abstractions, those found specifically in OSs have not been thoroughly studied. In this work, we identify six widely used patterns in OSs and propose modular abstractions for them.

Overall, our contributions are as follows:

- Chapter 2: To translate locks, we propose static analysis that identifies where locks are held and which data they protect, along with code transformation that replaces C locks with Rust locks. We implement this approach in a tool named Concrat (**Con**current-**C** to **R**ust **A**utomatic **T**ranslator) and evaluate it using 46 real-world C programs with locks. The implementation and benchmark programs are publicly available [100].

- Chapter 3: To translate tagged unions, we propose static analysis that identifies where tag values for unions are stored and code transformation that replaces unions with tagged unions. This approach is realized in a tool named Urcrat (**U**nion-**R**emoving **C**-to-**R**ust **A**utomatic **T**ranslator), which we evaluate with 36 real-world C programs using unions. The implementation and benchmark programs are publicly available [102].

- Chapter 4: To translate output parameters, we propose static analysis that identifies output parameters and code transformation that replaces output parameters with tuples and `Option`/`Result` types. This approach is implemented in a tool named Nopcrat (**N**o-**O**utput-**P**arameter **C**-to-**R**ust **A**utomatic **T**ranslator) and evaluated with 55 real-world C programs. The implementation and benchmark programs are publicly available [101].

- Chapter 5: To complement the approaches using static analysis, we propose type-migrating C-to-Rust translation using an LLM. We explicitly prompt the LLM to generate candidate type sig-

natures to facilitate type migration and augment functions with their callees' migrated signatures to reduce type errors. This approach is implemented in a tool named Tymcrat (**Ty**pe-**M**igrating **C**-to-**R**ust **A**utomatic **T**ranslator) and evaluated with 39 GNU programs written in C. The implementation is publicly available [99].

- Chapter 6: To facilitate the automatic translation of complex code patterns found in OSs, we develop modular abstractions for such patterns. Specifically, we identify six patterns in the xv6 OS [74] and design their modular abstractions while rewriting xv6 entirely in Rust, naming it xv6_{Rust}. The implementation of xv6_{Rust} is publicly available [103].

We also discuss related work (Chapter 7) and conclude the dissertation (Chapter 8).

Chapter 2. Translation of Locks

In system programming, concurrency is important yet notoriously difficult to get right. System software reduces execution time by spawning multiple threads and splitting tasks. As a drawback, it suffers from various bugs not existing in the sequential setting: data races, deadlock, starvation, etc [48].

Data races are the most common category of concurrency bugs [48]. It happens when multiple threads read and write the same memory address simultaneously. Data races lead system programs to exhibit not only unpleasant malfunctions but also critical security vulnerabilities [69].

Among synchronization mechanisms to avoid data races, locks are the most widely-used one. Each thread acquires and releases a lock before and after accessing shared data. This simplicity has facilitated the adoption of locks in diverse system software. Unfortunately, locks prevent data races only when they are used correctly. Programmers may acquire wrong locks, acquire locks too late, or release locks too early, thereby failing to eliminate data races.

C burdens programmers with the validation of correct lock use. The most popular lock API of C, pthreads [106], does not automatically check whether programs use locks correctly. Developers often fail to recognize incorrectly used locks in their programs, and C programs thus have suffered from data races.

On the other hand, Rust provides a lock API guaranteeing *thread safety*, i.e., the absence of data races, in `std::sync` of its standard library [30]. The combination of the ownership type system of Rust and the carefully designed API allows the type checker to validate the correct use of locks at compile time [115]. The API is different from the C lock API not only syntactically, e.g., in names of functions, but also semantically. For instance, the Rust lock API requires programs to explicitly describe which lock protects which data, while the C lock API does not.

In this chapter, we propose techniques to automatically replace the C lock API with the Rust lock API. Figure 2.1 shows the workflow of the proposed approach. We first translates C code using C2Rust. Then, we statically analyze the Rust code produced by C2Rust to construct a lock summary. Finally, we convert the C lock API to the Rust lock API using the lock summary.

Overall, we make the following contributions:

- We propose code transformation that replaces the C lock API in C2Rust-generated code with the Rust lock API based on the given lock summary (Section 2.2).
- We propose static analysis that efficiently generates a precise lock summary by combining bottom-up dataflow analysis and top-down data fact propagation (Section 2.3).
- We realize the proposed approach in a tool named Concrat. Our evaluation shows that the transformer efficiently and correctly transforms real-world programs and the analyzer outperforms the state-of-the-art static analyzer in terms of both speed and precision. Specifically, they transform and analyze 66 KLOC in 2.6 seconds and 4.3 seconds, respectively, and translate 74% of real-world programs to compilable code (Section 2.4).

2.1 Background

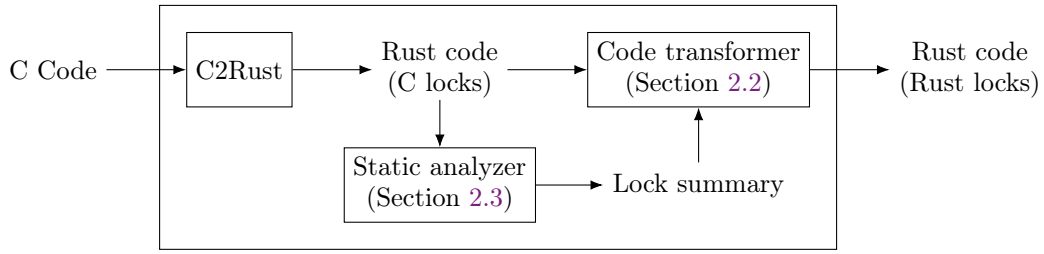


Figure 2.1: The workflow of Concrat

2.1.1 Lock API of C

The most widely used C lock API, pthreads [106], provides three types of lock: mutexes, read-write locks, and spin locks. While all of them are within the scope of this work, we mainly discuss mutexes. The others are similar to mutexes and briefly discussed in Section 2.4.1.

The API provides the `pthread_mutex_lock` and `pthread_mutex_unlock` functions, each of which takes a pointer to a lock; the former acquires the lock, and the latter releases the lock. Locks are used together with shared data. C programs have two common patterns to organize locks and shared data: *global* and *struct* [192].

Global Both data and lock are global variables.

```
int n = ...;
pthread_mutex_t m = ...;
void inc() {
    pthread_mutex_lock(&m);
    n += 1;
    pthread_mutex_unlock(&m);
}
```

The global variable `n` is a shared integer and `m` is a lock. Each thread must hold `m` when accessing `n`, i.e., `m` protects `n`. Thus, `inc` acquires and releases `m` before and after increasing `n`.

Struct Both data and lock are fields of the same struct.

```
struct s {
    int n;
    pthread_mutex_t m;
};
void inc(struct s *x) {
    pthread_mutex_lock(&x->m);
    x->n += 1;
    pthread_mutex_unlock(&x->m);
}
```

The lock stored in the field `m` of a struct `s` value protects the integer stored in the field `n` of the same struct value. Each thread must hold `x->m` when accessing `x->n`.

2.1.2 Data Races in C

The C lock API does not guarantee whether programs use locks correctly. Data races may occur by mistake despite the use of locks. There are two major reasons for data races: *data-lock mismatches* and *flow-lock mismatches*.

Data-Lock Mismatch A data-lock mismatch is an acquisition of an incorrect lock when accessing shared data. See the following where `m1` protects `n1` and `m2` protects `n2`:

```
pthread_mutex_lock(&m2);
n1 += 1;
pthread_mutex_unlock(&m2);
```

All the other parts of the program acquire `m1` when accessing `n1`. However, the above code has a bug: it acquires `m2`, instead of `m1`, when accessing `n1`. This allows multiple threads to access `n1` simultaneously, thereby incurring a data race.

Flow-Lock Mismatch A flow-lock mismatch is an acquisition of a lock at an incorrect program point. Consider the following program, where `m` protects `n`:

```
void f1() {
    n += 1;
    pthread_mutex_lock(&m);
    ...
}
void f2() {
    ...
    pthread_mutex_unlock(&m);
    n += 1;
}
```

The function `f1` accesses `n` before acquiring `m`, and `f2` accesses `n` after releasing `m`. Both functions are buggy as they allow accesses to the shared data when the lock is not held.

2.1.3 Lock API of Rust

The Rust lock API guarantees the correct use of locks [115]. The API makes two kinds of relation explicit: *data-lock relations*, i.e., which lock protects which data; *flow-lock relations*, i.e., which lock is held at which program point. It naturally prevents both data-lock mismatches and flow-lock mismatches.

Rust makes the data-lock relation explicit by coupling a lock with shared data. A Rust lock is a C lock plus shared data; it can be considered a protected container for shared data. The type of a lock is `Mutex<T>`, where `T` is the type of the protected data [39]. A program can create a lock as follows:

```
static m: Mutex<i32> = Mutex::new(0);
```

making `m` a lock initially containing `0`. The coupling of data and a lock prevents data-lock mismatches. When accessing shared data, threads acquire the lock coupled with the data.

Rust makes the flow-lock relations explicit by introducing the notion of a *guard*. Threads need a guard to access the in-lock data. A guard is a special kind of pointer to the in-lock data. The only way to create a guard is to acquire a lock. The `lock` method of a lock produces a guard as a return value.

Threads can access the protected data by dereferencing the guard. When a thread wants to release a lock, it drops, i.e., deallocates, the guard by calling `drop`. A predefined drop handler attached to the guard automatically releases the lock. The following shows the process from construction to destruction of a guard:

```
let mut g = m.lock().unwrap();
*g += 1;
drop(g);
```

Because `lock` returns a wrapped guard, `unwrap()` is required. The `unwrap` call fails and makes the current thread panic when a thread previously holding the lock has panicked before releasing it. Otherwise, `unwrap` returns the guard.

With the help of Rust's ownership type system [115], guards prevent flow-lock mismatches. In Rust, each function can use only the variables it owns. A function owns a variable after initializing it and loses the ownership after passing the variable to another function as an argument. The type checker detects every flow-lock mismatch at compile time by tracking the ownership of guards. Consider the following buggy code:

```
fn f1() {
    let mut g;
    *g += 1;
    g = m.lock().unwrap();
}
fn f2() {
    let mut g;
    ...
    drop(g);
    *g += 1;
}
```

Because `f1` uses `g` before owning it and `f2` uses `g` after losing the ownership, the type checker rejects both functions.

2.2 Code Transformation

This section proposes an automatic Rust code transformer that takes C2Rust-generated code and its lock summary as inputs and replaces the C lock API with the Rust lock API. We first describe the contents of a lock summary (Section 2.2.1) and then show how the transformer replaces the C lock API with the Rust lock API using the summary (Section 2.2.2).

2.2.1 Lock Summary

A program's lock summary abstracts its data-lock and flow-lock relations. A lock summary is a JSON file containing three maps: `global_lock_map`, `struct_lock_map`, and `function_map`. The first two represent data-lock relations, and the last one represents flow-lock relations. Listing 2.1 is the lock summary for the C2Rust-generated code in Listing 2.2. We describe each component of the summary in detail.

```
{
    "global_lock_map": { "n": "m" },
    "struct_lock_map": { "s": { "n": "m" } },
    "function_map": {
        "unlock": {
            "entry_lock": ["m"],
            "return_lock": [],
            ...
        },
        "lock": {
            "entry_lock": [],
            "return_lock": ["m"],
            ...
        },
        "foo": {
            ...
            "lock_line": { "m": [23, 24] },
        },
        ...
    }
}
```

Listing 2.1: Lock summary

Global Lock Map `global_lock_map` expresses data-lock relations of the global pattern. It is a map from a global variable to the lock variable protecting it. The summary states that `n` is protected by `m`.

Struct Lock Map `struct_lock_map` keeps data-lock relations of the struct pattern. It maps a struct type name to its summary, which maps a field to the lock field protecting it. The summary states that the field `n` of the struct type `s` is protected by the field `m`.

Function Map `function_map` expresses the flow-lock relation. It maps a function name to its summary, which consists of `entry_lock`, `return_lock`, and `lock_line`. Locks are represented by symbolic paths. For example, `m` and `x.m` are locks, where the type of the variable `x` is `s`. `entry_lock` is a list of locks that are always held at the entry of the function. The summary states that `m` is always held at the entry of `unlock`. `return_lock` is a list of locks that are always held at the return of the function. The summary states that `m` is always held at the return of `lock`. `lock_line` is a map from a lock to a list of lines in the function where the lock is held. The summary states that `m` is held in lines 23 and 24.

2.2.2 Transformation

The transformer produces Listing 2.3 by replacing the C lock API in Listing 2.2 with the Rust lock API. Note that each line of Listing 2.3 corresponds to the same line of Listing 2.2. We explain the transformation line-by-line.

- Lines 1–2: We check `global_lock_map` to identify locks and variables they protect. We define a new struct containing variables protected by a certain lock and replace the original C lock with a Rust

```
1 static mut n: i32 = 0;
2 static mut m: pthread_mutex_t = ...;
3 struct s { n: i32, m: pthread_mutex_t }
4 fn f() {
5     pthread_mutex_lock(&mut m);
6     n += 1;
7     pthread_mutex_unlock(&mut m);
8 }
9 fn unlock() {
10    pthread_mutex_unlock(&mut m);
11 }
12 fn lock() {
13    pthread_mutex_lock(&mut m);
14 }
15 fn g() {
16    lock();
17    n += 1;
18    unlock();
19 }
20 fn foo() {
21    n += 1; // safe for some reason
22    pthread_mutex_lock(&mut m);
23    n += 1;
24    pthread_mutex_unlock(&mut m);
25 }
```

Listing 2.2: Rust code before transformation

lock containing a struct value.

- Line 3: Similar to the above, but using `struct_lock_map`, instead of `global_lock_map`.
- Lines 4–8: We define an uninitialized guard variable at the beginning of each function using the guard. Each `pthread_mutex_lock` and `pthread_mutex_unlock` call is syntactically transformed into a `lock` method call and a `drop` function call, respectively. Note that the name of a guard is syntactically determined from the name of the lock according to a predefined rule. We replace each expression accessing protected data with an expression dereferencing a guard, whose lock name is found in `global_lock_map` or `struct_lock_map`, depending on the access path.
- Lines 9–11: We make a function take a guard as an argument if its `entry_lock` is nonempty.
- Lines 12–14: We make a function return a guard if its `return_lock` is nonempty. If there are multiple return guards or the original return value, tuples are constructed.
- Lines 15–19: We add a guard as an argument to a call to a function with nonempty `entry_lock`. We assign the return value of a function with nonempty `return_lock` to a guard variable.
- Lines 20–25: Even when `m` protects `n`, some accesses to `n` may not hold `m` because the developer thinks that `n` is never concurrently accessed by other threads in those specific lines. For this reason,

```
1 struct mData { n: i32 }
2 static mut m: Mutex<mData> = Mutex::new(mData{ n: 0 });
3 struct smData { n: i32 }  struct s { m: Mutex<smData> }
4 fn f() { let mut m_guard;
5     m_guard = m.lock().unwrap();
6     (*m_guard).n += 1;
7     drop(m_guard);
8 }
9 fn unlock(m_guard: MutexGuard<i32>) {
10    drop(m_guard);
11 }
12 fn lock() -> MutexGuard<i32> { let mut m_guard;
13    m_guard = m.lock().unwrap(); m_guard
14 }
15 fn g() { let mut m_guard;
16    m_guard = lock();
17    (*m_guard).n += 1;
18    unlock(m_guard);
19 }
20 fn foo() { let mut m_guard;
21    m.get_mut().n += 1; // safe for some reason
22    m_guard = m.lock().unwrap();
23    (*m_guard).n += 1;
24    drop(m_guard);
25 }
```

Listing 2.3: Rust code after transformation

we cannot blindly replace all the accesses to protected data with guard dereference. We need to figure out whether a certain guard exists in each line by checking `lock_line`. Since the summary states that `m` is held only in lines 23 and 24, the access in line 21 uses the `get_mut` method, instead of the guard. The method returns a pointer to the in-lock data.

Note that the use of `get_mut` relies on that `m` is defined mutable. If `m` is immutable, the type checker disallows calling `get_mut`. In Rust, mutable global variables are discouraged [22]. Reference-counted types, `Rc` [36] (in the sequential setting) and `Arc` [37] (in the concurrent setting), should replace mutable global variables. This makes `get_mut` succeed if the reference count equals one and panic otherwise, consequently preventing data races at run time even when the developer's assumption is wrong. Automatically replacing mutable global variables with `Rc` and `Arc` is beyond the scope of this work.

The transformed code looks similar to human-written code because the transformer utilizes code patterns that real-world Rust programmers use, e.g., putting fields into structs protected by locks, storing guards in variables, passing guards as arguments, and returning guards from functions. Still, there are some discrepancies: humans may prefer wrapping guards in structs and defining their methods instead of functions taking guards; they often omit `drop` calls at the end of a function, which can be automatically inserted by the compiler.

2.3 Static Analysis

In this section, we propose a static analysis to automatically generate lock summaries required by the transformer. The analysis must precisely determine the flow-lock relation to lead the transformer to produce compilable code. If a summary contains an imprecise flow-lock relation, the transformed code may be uncompilable due to the use of unowned guards. On the other hand, an imprecise data-lock relation does not hinder the transformed code from being compiled. If the analysis fails to find that `m` protects `n`, `n` will not be a field of a struct protected by `m`. If the analysis incorrectly concludes that `m` protects `n`, `n` will be accessed via `get_mut`. Both kinds of code are unideal but compilable and preserve the original semantics. We thus focus on designing an analysis that precisely computes the flow-lock relation.

The key intuition behind our analysis design is that the precision of the analysis does not need to exceed that of the type checker. Consider the following example:

```
if b { pthread_mutex_lock(&m); } ...
if b { pthread_mutex_unlock(&m); }
```

Even when the analysis is precise enough to track the path-sensitive use of locks, the transformed code is uncompilable:

```
let mut m_guard;
if b { m_guard = m.lock().unwrap(); } ...
if b { drop(m_guard); }
```

Since type checking is path-insensitive, it considers `m_guard` possibly uninitialized in the last line. This shows that a path-insensitive analysis is enough. Similarly, our analysis can be context-insensitive as the type checker is context-insensitive.

This intuition makes our analysis distinct from existing techniques: it is tailored to efficiently generate precise summaries for the code transformation by aiming the same precision as the type checker. Existing ones are either too imprecise or too precise. Some overapproximate the behavior of a program too much, so using their results as summaries would make the transformed code uncompilable. Some unnecessarily adopt rich techniques to make their results precise, thereby failing to finish the analyses in a reasonable amount of time.

Note that aiming the same precision as the type checker does not mean that we repeat the work of the type checker. While the goal of the type checker is to validate the use of guards, our goal is to infer the use of guards, which is more difficult. Specifically, the type checker takes code that already has guards and checks whether it uses guards properly in terms of ownership, but our analyzer takes code without any guards and reconstructs the flow of guards to determine whether each function needs to take or return certain guards.

Since guards are more concrete than information that certain locks are held, guards often make our explanation intuitive. Thus, we sometimes use guards in the explanation although the code being analyzed does not have any guards. The existence of a guard at a certain program point is equivalent to the corresponding lock always being held at the program point, and the term guard is exchangeable for the term *held lock*.

Our analysis consists of four phases: call graph construction (Section 2.3.1), bottom-up dataflow analysis (Section 2.3.2), top-down data fact propagation (Section 2.3.3), and data-lock relation identification (Section 2.3.4). The call graph is required for both bottom-up analysis and top-down propagation.

The bottom-up analysis and the top-down propagation collectively compute the flow-lock relation. Using the flow-lock relation, the last phase computes the data-lock relation.

2.3.1 Call Graph Construction

We draw call graphs by collecting the function names called in each function, without expensive control flow analysis. The drawback is that the call graph misses edges created by function pointers. However, the number of such edges is usually small because function pointers are rarely used in practice, and the subsequent analyses remain precise enough. Each node is a user-defined function; all the library functions are excluded from the graph. Therefore, each leaf node calls zero or more library functions but no user-defined functions.

We identify all the strongly connected components in the call graph to find mutually recursive functions, which need special treatment during the bottom-up analysis. We create a *merged* version of the call graph by merging each strongly connected component into a single node. We keep both original and merged call graphs to use the former for the top-down propagation and the latter for the bottom-up analysis.

2.3.2 Bottom-Up Dataflow Analysis

The goal of the bottom-up analysis is to identify the *minimum entry lock set* (MELS) and the *minimum return lock set* (MRLS) of each function. They are locks that must be held at the entry and the return, respectively. To compute the MELS and MRLS of each function, we perform two dataflow analyses on each function: *live guard analysis* (LGA) and *available guard analysis* (AGA). LGA computes MELSs, and AGA computes MRLSs. We need the control flow graph of each function for the analyses. The nodes are statements of the function, with two special nodes, `entry` and `ret`, which denote the entry and the return, respectively.

We traverse the merged call graph in post order to find the analysis target. It allows us to analyze leaf nodes first and then use their results to analyze internal nodes. Each node contains a single function or a set of mutually recursive functions. We discuss the analysis of non-recursive functions first and recursive functions afterward.

The goal of LGA is to compute MELSs. It is similar to the well-known live variable analysis [119]. Just like that the live variable analysis computes variables to be used in the future, LGA computes guards to be consumed by `pthread_mutex_unlock` in the future. Live guards at the entry of a function are the MELS of the function.

The analysis is a backward may analysis. Each `pthread_mutex_unlock` call, which consumes a guard, generates a guard. Each `pthread_mutex_lock` call, which produces a guard, kills a guard. The dataflow

equations are defined as follows:

$$
\begin{aligned}
\mathsf{In}^{\mathsf{L}}_s &= (\mathsf{Out}^{\mathsf{L}}_s - \mathsf{Kill}^{\mathsf{L}}_s) \cup \mathsf{Gen}^{\mathsf{L}}_s \\
\mathsf{Out}^{\mathsf{L}}_s &= \begin{cases} \emptyset & \text{if } s = \mathsf{ret} \\ \bigcup_{t \in \mathsf{Succ}_s} \mathsf{In}^{\mathsf{L}}_t & \text{otherwise} \end{cases} \\
\mathsf{Gen}^{\mathsf{L}}_s &= \begin{cases} \{p\} & \text{if } s = \texttt{pthread\_mutex\_unlock}(p) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathsf{Kill}^{\mathsf{L}}_s &= \begin{cases} \{p\} & \text{if } s = \texttt{pthread\_mutex\_lock}(p) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathsf{MELS} &= \mathsf{In}^{\mathsf{L}}_{\mathsf{entry}}
\end{aligned}
$$

where s and t range over statements; p ranges over paths; Succ_s denotes the set of every successor of s.

Example 2.1. The MELS of the following function is {m}:

```
fn unlock() { pthread_mutex_unlock(&mut m); }
```

Example 2.2. The MELS of the following function is {m}:

```
fn may_unlock() {
    if ... { pthread_mutex_unlock(&mut m); }
}
```

We get {m} by {m} ∪ ∅ because LGA is a may analysis. If it was a must analysis, MELS would be ∅, making the function take no guard after the transformation. Then, the function is uncompilable as it drops an unexisting guard.

The goal of AGA is to compute MRLSs. It is similar to the well-known available expression analysis [119]. Just like that the available expression analysis identifies expressions whose values have been computed in the past, AGA identifies guards constructed by `pthread_mutex_lock` in the past. Available guards at the return of a function are the MRLS of the function.

The analysis is a forward must analysis. Each `pthread_mutex_lock` call generates a guard, and each `pthread_mutex_unlock` call kills a guard. The dataflow equations are defined as follows:

$$
\begin{aligned}
\mathsf{Out}^{\mathsf{A}}_s &= (\mathsf{In}^{\mathsf{A}}_s - \mathsf{Kill}^{\mathsf{A}}_s) \cup \mathsf{Gen}^{\mathsf{A}}_s \\
\mathsf{In}^{\mathsf{A}}_s &= \begin{cases} \mathsf{MELS} & \text{if } s = \mathsf{entry} \\ \bigcap_{t \in \mathsf{Pred}_s} \mathsf{Out}^{\mathsf{A}}_t & \text{otherwise} \end{cases} \\
\mathsf{Gen}^{\mathsf{A}}_s &= \begin{cases} \{p\} & \text{if } s = \texttt{pthread\_mutex\_lock}(p) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathsf{Kill}^{\mathsf{A}}_s &= \begin{cases} \{p\} & \text{if } s = \texttt{pthread\_mutex\_unlock}(p) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathsf{MRLS} &= \mathsf{Out}^{\mathsf{A}}_{\mathsf{ret}}
\end{aligned}
$$

where Pred_s denotes the set of every predecessor of s.

Example 2.3. The MRLS of the following function is {m}:

```
fn lock() { pthread_mutex_lock(&mut m); }
```

Example 2.4. The MRLS of the following function is $\emptyset$:

```
fn may_lock() {
    if ... { pthread_mutex_lock(&mut m); }
}
```

We get $\emptyset$ by $\{\texttt{m}\} \cap \emptyset$ because AGA is a must analysis. If it was a may analysis, MRLS would be $\{\texttt{m}\}$, making the function return a guard after the transformation, which is uncompilable as it returns a possibly uninitialized guard.

Example 2.5. Both MELS and MRLS of the following are $\{\texttt{m}\}$:

```
fn unlock_and_lock() {
    if ... {
        pthread_mutex_unlock(&mut m); ...
        pthread_mutex_lock(&mut m);
    }
}
```

We get $\{\texttt{m}\}$ as the MRLS by intersecting $\{\texttt{m}\}$ and $\{\texttt{m}\}$. It is because setting $\mathsf{In}^{\mathsf{A}}_{\mathsf{entry}}$ to MELS allows `m` to be available even in the path where the condition is false. If $\mathsf{In}^{\mathsf{A}}_{\mathsf{entry}}$ was $\emptyset$, the MRLS would be $\emptyset$, making the transformed function not return an existing guard.

Analysis of internal nodes should consider the MELSs and MRLSs of callees. A function with a nonempty MELS consumes guards and acts like `pthread_mutex_unlock`. A function with a nonempty MRLS produces guards, like `pthread_mutex_lock`. Thus, during LGA, calling f kills MRLS_f and generates MELS_f, and during AGA, calling f kills MELS_f and generates MRLS_f.

Example 2.6. The MELS of `unlock2` is $\{\texttt{m}\}$.

```
fn unlock() { pthread_mutex_unlock(&mut m); }
fn unlock2() { unlock(); }
```

When structs are involved, we need to consider aliasing through argument passing. A caller and a callee represent the same lock with different paths if the path being an argument is different from the name of the corresponding parameter. Unless we recompute paths to reflect aliasing, the analyses produce incorrect results.

In this regard, we define *alias*, which recomputes paths:

$$alias(p, [x_1, \cdots, x_n], [e_1, \cdots, e_n]) = \begin{cases} e_i.p' & \text{if } p = x_i.p' \\ p & \text{otherwise} \end{cases}$$

It takes a path, a parameter list, and an argument list. If the path has one of the parameters as a prefix, *alias* replaces the prefix with the corresponding argument. Otherwise, the path remains the same. For example, $alias(\texttt{a.m}, [\texttt{a}], [\texttt{b}])$ equals `b.m`. Since we have a set of paths, we extend the definition of *alias* to recompute each path in a given set:

$$alias(\{\cdots, p, \cdots\}, \overline{x}, \overline{e}) = \{\cdots, alias(p, \overline{x}, \overline{e}), \cdots\}$$

An overlined symbol denotes a list. We revise our dataflow equations to handle user-defined function calls correctly:

$$\begin{aligned} \text{If } s = f(\overline{e}),\ \mathsf{Gen}^{\mathsf{L}}_s = \mathsf{Kill}^{\mathsf{A}}_s &= alias(\mathsf{MELS}_f, \mathsf{Params}_f, \overline{e}) \\ \mathsf{Kill}^{\mathsf{L}}_s = \mathsf{Gen}^{\mathsf{A}}_s &= alias(\mathsf{MRLS}_f, \mathsf{Params}_f, \overline{e}) \end{aligned}$$

where Params_f denotes the parameter list of f.

Example 2.7. The MELS of `lock_and_unlock` is $\emptyset$.

```
fn unlock(a: *mut s) {
    pthread_mutex_unlock(&mut (*a).m);
}
fn lock_and_unlock(b: *mut s) {
    pthread_mutex_lock(&mut (*b).m); unlock(b);
}
```

While the MELS of `unlock` is $\{\texttt{a.m}\}$, the `unlock` call in `lock_and_unlock` generates `b.m`, which is killed by the preceding `pthread_mutex_lock` call.

Analysis of a recursive function is challenging because it requires the MELS and MRLS of the function being analyzed. Our solution is an iterative analysis.

In the beginning, we have no information and set the MELS and MRLS to the bottom values: $\mathsf{MELS} = \emptyset$ and $\mathsf{MRLS} = \mathcal{L}$, the set of every possible lock path. For the MELS, $\emptyset$ is the bottom because LGA is a may analysis. On the other hand, $\mathcal{L}$ is the bottom for the MRLS because AGA is a must analysis.

We iteratively find a fixed point to compute the correct MELS and MRLS. We analyze the function with the MELS and MRLS we have. After the analysis, we update them with the result of the analysis. We repeat this until no change.

Example 2.8. The MELS of the following function is $\{\texttt{m}\}$.

```
fn unlock(n: i32) {
    if n <= 0 { pthread_mutex_unlock(&mut m); }
    else { unlock(n - 1); }
}
```

The first iteration gives us $\mathsf{MELS} = \{\texttt{m}\}$ by $\{\texttt{m}\} \cup \emptyset$. The second iteration produces the same result by $\{\texttt{m}\} \cup \{\texttt{m}\}$ and reaches a fixed point.

Example 2.9. The MRLS of the following function is $\{\texttt{m}\}$.

```
fn lock(n: i32) {
    if n <= 0 { pthread_mutex_lock(&mut m); }
    else { lock(n - 1); }
}
```

The first iteration makes $\mathsf{MRLS} = \{\texttt{m}\}$ by $\{\texttt{m}\} \cap \mathcal{L}$. The second iteration computes $\{\texttt{m}\} \cap \{\texttt{m}\}$, reaching a fixed point.

The iteration is guaranteed to terminate if $\mathcal{L}$ is finite. During the iteration, MELS can only grow, and MRLS can only shrink. Thus, the number of iterations is bounded by the size of $\mathcal{L}$. The iteration terminates almost always in practice. In most programs, $\mathcal{L}$ is finite, and the termination is guaranteed. However, some programs have a recursive data structure with locks, which makes $\mathcal{L}$ infinite. That said, a recursive function interacting with an unbounded number of locks is rare in practice, so the iteration can terminate despite $\mathcal{L}$ being infinite.

We can easily generalize this approach to mutually recursive functions. Given a set of mutually recursive functions, $f_1, \cdots, f_n$, we set all the MELSs and MRLSs to the bottom values: $\mathsf{MELS}_{f_i} = \emptyset$ and $\mathsf{MRLS}_{f_i} = \mathcal{L}$. We then analyze each function and update them with the results. We repeat the analysis until none of them change.

2.3.3 Top-Down Data Fact Propagation

A function summary for the transformation has to contain the *entry lock set* (ELS) and the *return lock set* (RLS) of the function. They are locks that can be always held at the entry and the return, respectively. It is important that they are different from the MELS and MRLS. The ELS contains guards given to a function by its caller, and the MELS contains some of them, which are dropped by the function. Consequently, the MELS is always a subset of the ELS. Similarly, the RLS contains guards returned by a function to its caller, and the MRLS contains some of them, which are constructed in the function. The MRLS is always a subset of the RLS. In addition, the following equation holds: $\mathsf{ELS} - \mathsf{MELS} = \mathsf{RLS} - \mathsf{MRLS}$. We call this common difference the *propagated lock set* (PLS). The PLS of a function is the set of guards given from and returned to its caller.

Example 2.10. Both MELS and MRLS of `inc` are $\emptyset$, but both ELS and RLS of `inc` are {`m`}.

```
fn safe_inc() {
    pthread_mutex_lock(&mut m); inc();
    pthread_mutex_unlock(&mut m);
}
fn inc() { n += 1; }
```

We need the ELS and RLS of `inc` to identify the data-lock relation correctly. If we consider only the MELS and MRLS, we incorrectly conclude that `m` does not protect `n`.

The goal of the top-down data fact propagation is to compute the ELS and RLS of each function. We first compute the ELS of each function. It allows us to find the PLS by subtracting the MELS from the ELS. Then, the union of the PLS and the MRLS is the RLS.

We first collect all the available guards at each function call. The arguments of the function call are collected together to recompute paths according to aliasing. $\mathsf{Call}_{f,g}$ is the set of pairs, each of which consists of the set of available guards and the list of arguments when f calls g. Because f may call g multiple times, multiple pairs may exist. Available guards at each call are already computed during AGA. Thus, $\mathsf{Call}_{f,g}$ is:

$$\mathsf{Call}_{f,g} = \{(\mathsf{In}^{\mathsf{A}}_s, \overline{e}) \mid s = g(\overline{e}) \ \wedge \ s \text{ is in } f\}$$

We then perform a top-down dataflow analysis to compute the ELS of each function. The analysis is cheap because it does not analyze function bodies and simply propagates data facts through call edges. If a function does not have any callers, its ELS is the same as its MELS. Otherwise, its callers propagate available guards. Each caller propagates not only the available guards identified by AGA, but also the guards propagated from its own callers. We want always-propagated guards, so we compute the intersection of the guards from each caller. The dataflow equations are as follows:

$$\mathsf{ELS}_g = \begin{cases} \mathsf{MELS}_g & \text{if } \mathsf{Pred}_g = \emptyset \\ \bigcap_{f \in \mathsf{Pred}_g} \mathsf{Prop}_{f,g} & \text{otherwise} \end{cases}$$
$$\mathsf{Prop}_{f,g} = \bigcap_{(P,\overline{e}) \in \mathsf{Call}_{f,g}} \mathit{alias}(P \cup \mathsf{ELS}_f, \overline{e}, \mathsf{Params}_g)$$

where $\mathsf{Prop}_{f,g}$ is the set of guards propagated from f to g.

We finally compute the PLS and RLS:

$$\mathsf{PLS} = \mathsf{ELS} - \mathsf{MELS} \qquad \mathsf{RLS} = \mathsf{MRLS} \cup \mathsf{PLS}$$

Example 2.10. (*continued*)

- $\mathsf{Call}_{\texttt{safe\_inc,inc}} = \{(\{\texttt{m}\}, [\,])\}$
- $\mathsf{Prop}_{\texttt{safe\_inc,inc}} = \{\texttt{m}\}$
- $\mathsf{ELS}_{\texttt{inc}} = \mathsf{PLS}_{\texttt{inc}} = \mathsf{RLS}_{\texttt{inc}} = \{\texttt{m}\}$

After finishing the top-down propagation, we can generate a function summary of each function. `entry_lock` and `return_lock` are the same as the ELS and RLS, respectively. The set of locks held in each line, required by `lock_line`, is determined by combining available guards identified by AGA and the PLS of the function.

2.3.4 Data-Lock Relation Identification

We identify the data-lock relation from the flow-lock relation computed by the preceding analysis. The key idea is to find the lock held at each access to a certain path. Since the global pattern and the struct pattern require different treatments, we split paths into global variables and struct fields and compute the data-lock relation of each.

We first discuss the global pattern. The first step is to collect every access to each global variable. We record all the available guards and whether the access is read or write. The available guards are the union of those found by AGA and the PLS of the function where the access happens. Acc_x is the set of accesses to a global variable x:

$$\mathsf{Acc}_x = \{(s, \mathsf{In}^{\mathsf{A}}_s \cup \mathsf{PLS}_f, a) \mid \mathit{access}(s, x, a) \ \wedge \ s \text{ is in } f\}$$

where $\mathit{access}(s, x, \mathsf{r})$ and $\mathit{access}(s, x, \mathsf{w})$ hold when s reads x and s modifies x, respectively. We then find a candidate lock for each global variable. A candidate lock is a lock that is held most frequently when accessing the variable:

$$\mathsf{Cand}_x = \mathrm{argmax}_y \, |\{(s, P, a) \in \mathsf{Acc}_x \mid y \in P\}|$$

We split accesses into safe and unsafe ones according to the existence of the candidate lock. We consider an access safe if it happens when the candidate is held, and unsafe otherwise:

$$\mathsf{Safe}_x = \{(s, P, a) \in \mathsf{Acc}_x \mid \mathsf{Cand}_x \in P\}$$
$$\mathsf{Unsafe}_x = \{(s, P, a) \in \mathsf{Acc}_x \mid \mathsf{Cand}_x \notin P\}$$

To determine the data-lock relation, we need to check whether each statement is *concurrent*, i.e., can run concurrently with other threads. The existence of an unsafe access does not necessarily mean that the candidate lock does not protect the global variable. If a statement is non-concurrent, it can safely access a global variable without holding a lock. Therefore, the precise identification of the data-lock relation requires a precise thread analysis. Since a precise thread analysis is expensive, we instead propose a simple heuristic. We consider a statement concurrent only if it belongs to a function reachable from an argument to `pthread_create`, the thread-spawning function.

Using the heuristic, we determine whether a candidate lock really protects the global variable. The candidate protects the variable if a safe write access exists and every unsafe access happens in a non-concurrent statement.

$$\mathsf{Cand}_x \text{ protects } x \text{ iff}$$
$$(\exists (s, P, a) \in \mathsf{Safe}_x, a = \mathsf{w}) \ \wedge \ (\forall (s, P, a) \in \mathsf{Unsafe}_x, s \text{ is non-concurrent})$$

For the struct pattern, we collect accesses to each field of a struct. A candidate lock for a field must be a field in the same struct. $\mathsf{Acc}_{T,l}$ is the set of accesses to a field l in a type T, and $\mathsf{Cand}_{T,l}$ is a candidate for it:

$$\mathsf{Acc}_{T,l} = \{(s, \mathsf{In}^{\mathsf{A}}_s \cup \mathsf{PLS}_f, p, a) \mid \mathit{access}(s, p.l, a) \ \wedge \ \mathit{type}(p) = T \ \wedge \ s \text{ is in } f\}$$
$$\mathsf{Cand}_{T,l} = \text{argmax}_{l'} |\{(s, P, p, a) \in \mathsf{Acc}_x \mid p.l' \in P\}|$$

We split accesses into safe and unsafe ones and check whether the candidate protects the field, just as we do for the global pattern. The only difference is that the condition for a statement to be considered concurrent is stricter than before. A struct value is not accessible from other threads right after its creation. It becomes accessible only after the function shares it with other threads by storing it in a global data structure or passing it as a thread argument. Determining when a value is shared requires a precise thread analysis as well, so we propose a heuristic. For a given struct value containing a lock field l, we consider a statement non-concurrent not only when its function is unreachable from an argument to `pthread_create`, but also when the function initializes l by calling `pthread_mutex_init`. Such a function is usually where the struct value is created and uniquely accessed.

2.4 Evaluation

In this section, we evaluate our approach with 46 real-world concurrent C programs. We first describe our implementation of Concrat, which realizes the proposed approach (Section 2.4.1), and the process of collecting the benchmark programs (Section 2.4.2). We then assess our approach by addressing the following research questions:

- RQ1. Scalability of transformation: Does it transform large programs in a reasonable amount of time? (Section 2.4.3)
- RQ2. Applicability: Does it handle most code patterns found in real-world programs? (Section 2.4.4)
- RQ3. Correctness: Does it preserve the semantics of the original program? (Section 2.4.5)
- RQ4. Scalability of analysis: Does it analyze large programs quickly, compared to the state-of-the-art static analyzer? (Section 2.4.6)
- RQ5. Precision: Does it produce precise lock summaries, compared to the state-of-the-art static analyzer? (Section 2.4.7)

Our experiments were conducted on an Ubuntu machine with Intel Core i7-6700K (4 cores, 8 threads, 4GHz) and 32GB DRAM. Finally, we discuss potential threats to validity (Section 2.4.8).

2.4.1 Implementation

We implemented Concrat on top of the Rust compiler [12]. The transformer lowers given code to the compiler's high-level intermediate representation [14] and walks it to replace the C lock API. The analyzer uses the compiler's dataflow analysis framework [13] for its mid-level intermediate representation [15].

Concrat handles not only mutexes, but also read-write locks, spin locks, and condition variables. Since Rust recommends using mutexes instead of spin locks [123], we replace them with `RwLock` [40], `Mutex`,

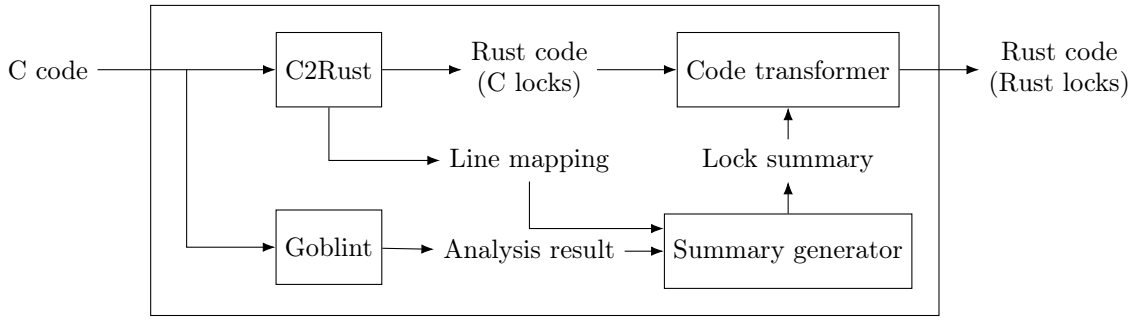


Figure 2.2: The workflow of Concrat$_G$

and `Condvar` [38] of `std::sync`. Concrat does not support re-entrant locks because the Rust standard library does not provide them.

To compare our analyzer with the state-of-the-art static analyzer for concurrent programs, we additionally built Concrat$_G$. Figure 2.2 illustrates the workflow of Concrat$_G$. It uses Goblint [192, 180] to analyze C code. Goblint computes both data-lock and flow-lock relations using abstract interpretation [73]. Since Goblint's result contains line numbers for C code, our summary generator replaces them with those for Rust code using the C-to-Rust line mappings generated by C2Rust.

Note that we can change the implementation of Concrat to analyze C code, instead of Rust code, just like Concrat$_G$, because the proposed analysis is language-agnostic. Our choice of analyzing Rust code eases implementation as we can utilize the Rust compiler's dataflow analysis framework.

2.4.2 Benchmark Program Collection

We collected 46 real-world concurrent C programs, all of the public GitHub repositories satisfying the following conditions: 1) more than 1,000 stars, 2) not a study material, 3) using the pthread lock API at least once, 4) C code less than 500,000 bytes, and 5) translatable with C2Rust. Two projects satisfied the first four conditions but not the last; they use C11 Atomics, but C2Rust supports only C99-compliant code. When C2Rust generates uncompilable code due to wrong type casts, we included such projects after manually fixing them.

Table 2.1 shows the collected programs and the code size of each. The second and third columns show the lines of C code and C2Rust-generated code, respectively; the fourth to seventh columns show the numbers of `pthread_mutex_*`, `pthread_rwlock_*`, `pthread_spin_*`, and `pthread_cond_*` calls, respectively.

2.4.3 RQ1: Scalability of Transformation

We translate the programs with Concrat to evaluate the transformer's scalability. In Table 2.2, the second column shows the transformation time; the thrid and fourth show the inserted and deleted lines, measured with `diff`.

The result shows that the transformer is scalable. As Figure 2.3 shows, the transformation time is proportional to the Rust LOC, and it takes less than 2.5 seconds to transform 66 KLOC by inserting and deleting hundreds of lines.

Table 2.1: Benchmark programs for evaluating Concrat

Program	C LOC	Rust LOC	Mutex	Rwlock	Spin	Cond
AirConnect	17516	32565	88	0	0	14
axel	5848	7685	16	0	0	0
brubeck	5635	6769	15	0	17	0
C-Thread-Pool	710	791	23	0	0	7
cava	4768	6538	10	0	0	0
Cello	20885	30796	5	0	0	0
Chipmunk2D	16053	21509	12	0	0	10
clib	25073	66287	38	0	0	0
dnspod-sr	9259	12596	0	0	99	0
dump1090	4646	6281	9	0	0	6
EasyLogger	2011	29298	4	0	0	0
fzy	2621	4013	4	0	0	0
klib	716	1016	14	0	0	14
kona	38850	48583	10	0	0	0
level-ip	5414	6651	36	23	0	4
libaco	1282	1800	6	0	0	0
libfaketime	521	806	6	0	0	6
libfreenect	627	962	10	0	0	4
libqrencode	6670	9013	4	0	0	0
lmdb	10827	16290	27	0	0	6
minimap2	17279	23531	6	0	0	4
Mirai-Source-Code	1839	2889	14	0	0	0
neural-redis	3645	6312	12	0	0	0
nnn	12091	16424	7	0	0	0
pg_repack	7420	8152	10	0	0	0
phpspy	19390	29860	8	0	0	10
pianobar	11452	33212	45	0	0	17
pigz	9118	12660	5	0	0	7
pingfs	2318	3332	26	0	0	6
ProcDump-for-Linux	4152	6961	31	0	0	11
proxychains	2686	5460	6	0	0	0
proxychains-ng	5203	9031	8	0	0	0
Remotery	7212	9361	4	0	0	0
sc	142	206	7	0	0	0
shairport	8605	12533	37	0	0	0
siege	19281	25412	21	0	0	16
snoopy	2262	4605	4	0	0	0
sshfs	7193	9914	75	0	0	13
streem	20169	31444	36	0	0	0
stud	7931	10789	12	0	0	0
sysbench	16020	41222	19	10	0	9
the_silver_searcher	7242	12453	23	0	0	5
uthash	817	1450	0	6	0	0
vanitygen	10919	9710	23	0	0	11
wrk	8658	12255	12	0	0	0
zmap	17435	24366	12	0	0	0

Table 2.2: Experimental results of Concrat

	Transformation									Analysis		
Program	Time	Ins	Del	Succ	Reason	Fix	Test	Test_C	Test_O	Time_O	Time_G	Succ_G
AirConnect	1.159	870	874	✗	cond acq					1.701	timeout	
axel	0.344	78	112	✓						0.409	error	
brubeck	0.289	99	91	✓			✓	✓	✓	0.362	80.393	✗
C-Thread-Pool	0.059	106	113	✓			✓	✓	✓	0.060	0.728	✓
cava	0.289	29	27	✓						0.334	72.908	✓
Cello	1.521	27	15	✗	func ptr					3.103	error	
Chipmunk2D	0.839	52	60	✓						2.074	error	
clib	2.416	193	207	✗	func ptr					4.234	timeout	
dnspod-sr	0.537	272	234	✗→✓		dead				0.696	1667.203	✗
dump1090	0.259	44	77	✓			✓	✓	✓	0.307	timeout	
EasyLogger	0.219	428	415	✗	cond acq					0.234	6.458	✗
fzy	0.154	16	16	✓			✓	✓	✓	0.168	error	
klib	0.076	62	100	✓			✓	✓	✓	0.078	error	
kona	2.067	37	31	✓			✓	✓	✓	3.223	timeout	
level-ip	0.329	232	317	✗	func ptr					0.442	timeout	
libaco	0.090	22	33	✓			✓	✓	✓	0.105	timeout	
libfaketime	0.059	43	85	✓			✓	✓	✗	0.059	0.147	✓
libfreenect	0.066	87	116	✓						0.069	3.439	✓
libqrencode	0.367	28	40	✓			✓	✓	✓	0.447	error	
lmdb	0.722	266	292	✗	lock arg					0.910	error	
minimap2	1.044	26	56	✓			✓	✓	✓	1.438	73.902	✓
Mirai-Source-Code	0.151	118	135	✗→✓		goto				0.164	43.736	✗→✓
neural-redis	0.261	51	59	✓						0.310	186.593	✓
nnn	0.822	37	66	✓						1.056	2264.742	✓
pg_repack	0.306	36	35	✓			✓	✓	✓	0.384	error	
phpspy	1.199	1441	1479	✗	cond acq					1.580	error	
pianobar	1.248	132	182	✓						1.624	timeout	
pigz	0.471	176	178	✗→✓		no ret	✓	✓	✓	0.615	error	
pingfs	0.180	110	137	✗	cond acq					0.198	9.278	✗
ProcDump-for-Linux	0.245	171	438	✓			✗			0.286	79.284	✗
proxychains	0.218	44	52	✓						0.223	49.203	✓
proxychains-ng	0.389	24	32	✓						0.444	1058.823	✓
Remotery	0.397	34	24	✗	cond acq					0.567	error	
sc	0.029	34	54	✓			✓	✓	✓	0.030	0.049	✓
shairport	0.576	1093	1046	✗	cond acq					0.736	error	
siege	1.053	202	293	✗	lock arg					1.882	error	
snoopy	0.198	23	38	✓			✓	✓	✓	0.234	3.855	✓
sshfs	0.388	277	302	✓			✓	✓	✓	0.517	2028.390	✗
streem	1.070	115	114	✗→✓		bug fix	✓	✓	✓	2.084	error	
stud	0.423	61	54	✓						0.505	error	
sysbench	0.999	110	244	✗→✓		goto	✓	✗		1.149	error	
the_silver_searcher	0.441	112	177	✓			✓	✓	✓	0.548	error	
uthash	0.078	56	57	✓			✓	✓	✓	0.091	0.100	✓
vanitygen	0.379	121	224	✓						0.454	error	
wrk	0.465	35	42	✓						0.525	error	
zmap	0.795	42	71	✗	lock arg					1.257	timeout	

Times are in seconds. Subscript C means C2Rust, O means ours, and G means Goblint.

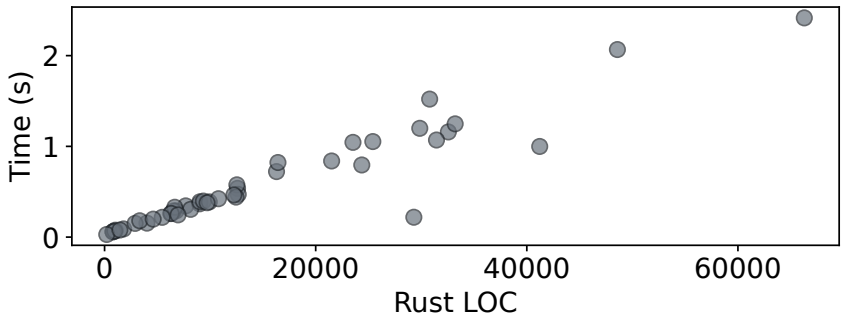


Figure 2.3: Transformation time per benchmark program using Concrat

2.4.4 RQ2: Applicability

We check whether the transformer handles diverse code patterns in real-world programs to evaluate its applicability. We consider that the transformer successfully handles a certain pattern if the transformed code is compilable. In Table 2.2, the fifth column shows compilability; the sixth shows the reason for a failure; the seventh shows our manual fix for the original code to make compilation succeed.

The transformer has high applicability. Among 46, 29 are compilable, 5 are compilable requiring manual fixes, and 12 are not. Overall, 74% of the programs are compilable.

Failures We manually investigated the reasons for compilation failures and found three code patterns.

- *Conditional acquisitions (cond acq)*: A function conditionally acquires a lock. Consider the following code:

```
fn may_lock() -> i32 {
    if b {
        pthread_mutex_lock(&mut m);
        0
    } else {
        1
    }
}
if may_lock() == 0 {
    n += 1;
    pthread_mutex_unlock(&mut m);
}
```

  The function `may_lock` acquires a lock and returns `0` if a certain condition is satisfied, and otherwise returns `1` without acquiring the lock. Its caller accesses the shared data only when the return value is `0`. The transformer cannot handle this pattern. Since `may_lock` has empty `return_lock`, it does not return any guards after the transformation. Its caller drops an unowned guard, thereby being uncompilable. To solve this, we have to make `may_lock` return `Option` [27] of a guard. Conditionally-succeeding functions appear in most C programs, not only in concurrent ones. Translating them to functions returning `Option` will be promising future work.

- *Function pointers (func ptr)*: A function that takes or returns guards is used as a function pointer. Since adding guards changes the type of the function pointer, the transformed code is uncompilable.

To address this pattern, we need to transform functions that take function pointers. Such functions are often in a library; if it is the case, the library also should be rewritten in Rust.

- *Lock arguments (lock arg)*: A function takes only a pointer to a lock as an argument. It would be an interesting improvement to handle this pattern with generic functions. If the function does not access any data protected by a specific lock, we can make the function take an argument of `Mutex<T>`, where `T` is a type parameter.

Manual Fixes We made four kinds of manual fix. We first explain two code patterns requiring manual fixes.

- *Removing `goto` (goto)*: A function uses `goto`.

```
    if (b) {
        pthread_mutex_unlock(&m);
        goto err;
    }
    n += 1;
    ...
    return 0;
err:
    return 1;
```

For the above code, C2Rust replaces `goto` with `if`:

```
if b {
    pthread_mutex_unlock(&mut m);
    x = 123;
}
if x != 123 {
    n += 1;
    ...
    return 0;
}
return 1;
```

Then, the transformer generates uncompilable code:

```
if b {
    drop(m_guard);
    x = 123;
}
if x != 123 {
    *m_guard += 1;
    ...
    return 0;
}
return 1;
```

Since type checking is path-insensitive, it considers `m_guard` possibly unowned in the second line. We replaced `goto` with the statements after the jump target:

```
if (b) {
    pthread_mutex_unlock(&m);
    return 1;
}
n += 1;
...
return 0;
```

which Concrat translates to compilable code.

- *Changing return type of no-return function (no ret)*: A function does not return. Consider the following code:

```
fn err() {
    exit(-1);
}
if b {
    pthread_mutex_unlock(&mut m);
    err();
}
n += 1;
...
```

Since `exit` does not return, `err` does not either and accessing `n` is safe. But, the type checker does not know that `err` never returns, and the transformed code is uncompilable:

```
if b {
    drop(m_guard);
    err();
}
*m_guard += 1;
...
```

The type checker considers `m_guard` possibly unowned in the last line. We changed the return type of `err` to `!` [31]:

```
fn err() -> ! {
    ...
    exit(-1);
}
```

indicating no return. The transformed code is compilable.

Both manual fixes can be avoided by improving C2Rust. We now explain program-specific fixes.

- *Removing dead code (dead)*: We deleted a function taking a pointer to a lock because it is never called.
- *Bug fix (bug fix)*: streem has the following code (simplified):

```
pthread_mutex_unlock(&m);
if (...) {
    pthread_mutex_unlock(&m);
```

```
    return;
}
```

Due to the second `pthread_mutex_unlock` call, the transformed code is uncompilable. We believe it is a bug and removed it. We contacted the developer but have not received a response yet. This confirms the common belief that rewriting legacy programs in Rust can reveal unknown bugs.

2.4.5 RQ3: Correctness

We ran the test cases of each program whose transformation succeeds to evaluate the correctness of the transformer. A correct transformer must preserve the semantics of the original program. We consider the transformer correct if the transformed program passes all of its test cases. The eighth to tenth columns show whether the original C program, the C2Rust-generated program, and the transformed program pass the test cases, respectively.

The result shows that our approach transforms most programs correctly. Among 34 programs compilable after the transformation possibly with manual fixes, 14 have no test cases or only those covering no lock API calls, so we performed the evaluation with the remaining 20. One fails even before C2Rust's translation. One fails after C2Rust's translation because it incorrectly translates some inline assembly. After the transformation, 17 still pass their tests, but 1 fails.

The failing program does not reveal an inherent limitation of our approach. The reason for the incorrect transformation is the imprecise `timespec` tracking of our transformer implementation. While `pthread_cond_timedwait` of the C lock API takes what time to wait for, its Rust counterpart takes how long to wait. To address this discrepancy, the transformer syntactically finds a `clock_gettime` call, which sets a given `timespec` to the current time, and how many seconds are added to the `timespec` before calling `pthread_cond_timedwait`. However, the failing program uses multiple `timespec` values, whose relation cannot be found by our syntactic analysis. We can easily resolve this issue by adopting intraprocedural value analysis for `timespec`.

It is not surprising that the transformer is correct in most cases as far as the transformed code is compilable because the design of the transformation justifies the correctness. It transforms a `lock` function call to a `lock` method call and an `unlock` function call to the drop of a guard whose finalizer unlocks the connected lock. Since the names of lock and guard variables are syntactically related, the dropped guard always unlocks the correct lock. It transforms each lock-protected data access to field access through a guard. Again, the lock and guard names are syntactically related, so the transformed code always accesses the correct data. The only possible threat is guards being used before initialization or after destruction, but the compiler detects it.

2.4.6 RQ4: Scalability of Analysis

We translate the collected programs with both Concrat and Concrat$_G$ and measure the analysis time to compare the scalability of our analyzer and Goblint. For Goblint, we use the `medium-program.json` configuration [7] and additionally enable the `allfuns` option to analyze every function for programs without `main`. It makes Goblint perform flow-, context-, path-sensitive analysis. We set a 24-hour time limit. The eleventh and twelfth columns of Table 2.2 show the time required by ours and Goblint.

The result shows that our analyzer is more scalable than Goblint. Goblint fails to analyze 19 programs due to internal errors. Our analyzer processes all the remaining 27 programs faster than Goblint. It takes less than 4.3 seconds to analyze 66 KLOC. On the other hand, Goblint does not even

finish the analysis of eight programs in the time limit and takes 1.1× to 3923× more than ours to analyze the other 19 programs.

2.4.7 RQ5: Precision

We measure the precision of lock summaries generated by our analyzer and Goblint to compare their precision. We use the compilability of code translated by Concrat and Concrat$_G$ as a proxy for assessing the analyzers' precision because an imprecise flow-lock relation makes transformed code uncompilable, as discussed in Section 2.3. The fifth and thirteenth columns of Table 2.2 show compilability after Concrat's and Concrat$_G$'s translation, respectively.

The result shows that our analyzer is more precise than Goblint for four programs, generating summaries that made the transformed code compilable. Our manual investigation confirms that those translated by Concrat$_G$ are uncompilable due to imprecision in the flow-lock relations. Goblint's imprecision mainly stems from the imprecise lock-aliasing information. It knows locks `a` and `b` are aliased when `a` is locked, but this information is sometimes lost due to overapproximation, and `b` is considered still locked even after `a` is unlocked, leading to imprecise flow-lock relations.

2.4.8 Threats to Validity

The primary threats to internal validity lie in the implementation of our tool. We implemented it with the dataflow analysis framework of the Rust compiler, which is already massively used and tested by the compiler.

The threats to external validity concern the choice of the translated C projects, each of which has more than 1,000 stars and C code of fewer than 500,000 bytes. Less popular or larger projects may have code patterns unseen in the selected projects. Further experiments with more C projects can give more confidence in the generalizability of our approach.

The threats to construct validity include evaluation metrics. We used whether a compilation succeeds or not as a proxy for the applicability of the transformer and the precision of the analyzer. We ran test cases to evaluate the correctness of the transformer. While test cases cannot guarantee the semantics preservation of the transformation, in practice, running test cases is the most popular way to check whether a certain program has the intended semantics.

Chapter 3. Translation of Unions

A union is a compound data type consisting of multiple fields sharing the same memory storage, facilitating efficient memory use by allowing values of different types to be stored at the same location [23]. Since memory efficiency is crucial in system software, unions are widely used in C. Notably, Emre et al. [84] show that 18% of unsafe functions (functions using unsafe features) in C2Rust-generated code involve unions.

Reading a union field is an unsafe feature in Rust because unions do not record which field has been written to. If a program reads a field other than the last-written one, the value is *reinterpreted* as another type. While reinterpretation is useful for some uses, like packet parsing, it is dangerous in general. For example, reinterpreting an integer as a pointer can lead to invalid memory access. Thus, many C programs avoid reinterpretation when using unions.

To use unions without reinterpretation, it is essential to decide which field to read. Some programs rely on global context to determine the field, but many use *tags*, i.e., integer values signifying the last-written fields. When using tags, a union and a tag are placed in a single struct, and the program checks the tag before accessing the union's field. However, tags cannot guarantee the absence of reinterpretation. Programs may read wrong fields after checking tags or set incorrect tag values when writing to fields.

Rust directly supports this pattern of combining tags and unions as a language feature called tagged unions [17]. This allows defining a tagged union as a single type by enumerating tags and the type of a value associated with each tag. By using tagged unions, programmers can avoid mistakenly reinterpreting values. To access a value in a tagged union, programs must use *pattern matching*, which checks the tag and provides access to the associated value. When constructing a tagged union, the compiler ensures that the tag and the value's type match the type definition. Thus, tagged unions are a safe feature in Rust, making it desirable to replace unions accompanied by tags with tagged unions.

In this chapter, we propose techniques to translate C's unions to Rust's tagged unions. Figure 3.1 shows the workflow of the proposed approach. We first translate C code to Rust code that still contains unions using C2Rust. We then transform the C2Rust-generated code by replacing unions with tagged unions. To enable this transformation, we perform static analysis to obtain information related to unions: (1) the *tag field* (the field containing a tag value) for each union and (2) tag values associated with each union field. This static analysis must meet several challenging requirements.

First, the analysis needs to determine the values of struct fields. Programs typically use `switch`/`if` to access different union fields in different branches, using the tag field in the condition. If a struct field has distinct values when accessing different union fields, it is likely a tag field, and each distinct value is associated with the accessed union field. If it has the same value when accessing different union fields, it is not a tag field. Thus, we can identify tag fields by deciding the value of each struct field in `switch`/`if` branches.

To achieve this goal, we propose a *must-points-to analysis* capable of tracking *integer equality*. To determine a struct's field value at each program point based on the branch that the program point belongs to, the struct should be the same as the struct whose field is used in the `switch`/`if` condition. Since structs are often passed as pointers, deciding whether they are the same requires must-points-to relations. Additionally, programs sometimes use a local variable storing a field's value in a condition. In such cases, determining the field's value in each branch requires the knowledge that the field and the

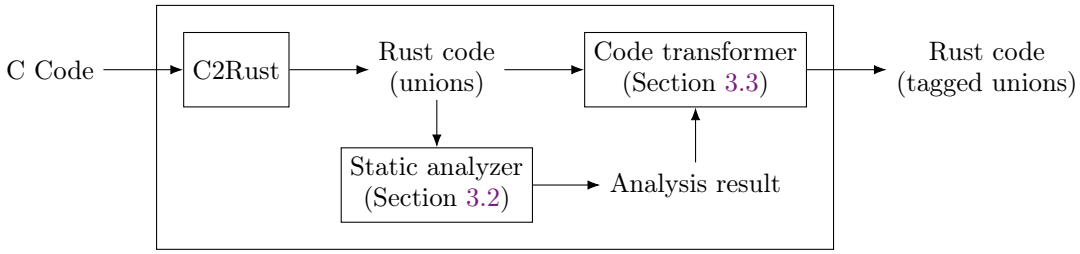


Figure 3.1: The workflow of Urcrat

local variable have the same integer.

The second requirement for the analysis is efficiency. The key idea for achieving efficiency is to selectively analyze functions. To identify tag fields, we need the field values only of the structs containing unions. It means that functions not accessing such structs do not need to be analyzed. Therefore, we adopt intraprocedural function-wise analysis, instead of interprocedural whole-program analysis, allowing only the selected functions to be analyzed.

The third requirement is the ability to identify tag values associated with each union field despite the imprecision of the analysis. Given that imprecision is a fundamental limitation of static analysis [172], it may not be possible to determine the field values at some program points. To address this, we propose a heuristic to interpret such partial information. The heuristic involves two steps. First, we examine the accessed union fields and the struct field values in `switch`/`if` branches. This provides reliable information because programs typically access the correct union field after checking the tag. However, due to imprecision, it may fail to identify some tag values. Second, we inspect the last-written union fields and the struct field values at each program point. This can capture information missed by the first step but may be incorrect because of an *intermediate* state, where only the tag or the union field has been set. Therefore, we ignore the field associated with a tag by the second step if it differs from the one associated with the tag by the first step.

Overall, the contributions are as follows:

- We propose static analysis that identifies tag fields and tag values associated with union fields, consisting of must-points-to analysis capable of tracking integer equality and a heuristic interpreting the analysis results (Section 3.2).
- We propose code transformation replacing unions with tagged unions using the analysis results (Section 3.3).
- We implement the proposed approach in a tool named Urcrat and evaluate it using 36 real-world C programs. Our evaluation shows that the approach is (1) precise, identifying 74 tag fields with no false positives and only five false negatives, (2) mostly correct, with 17 out of 23 programs passing tests after transformation, and (3) efficient, capable of analyzing and transforming 141k LOC in 4,910 seconds (Section 3.4).

3.1 Background

In this section, we briefly describe the use of unions with tags in C (Section 3.1.1), how C2Rust translates such C code to Rust (Section 3.1.2), and how tagged unions in Rust can safely represent the same logic (Section 3.1.3).

3.1.1 Unions with Tags

As an example of C code using unions, we use the syntax and evaluation of simple arithmetic expressions defined as follows:

$$e ::= 1 \mid -e \mid e + e \mid e * e$$

An expression is either a constant 1, a negation, an addition, or a multiplication. This syntax is implemented as follows:

```
struct Expr {
    int kind;
    union {
        struct Expr *e;
        struct BExpr b;
    } v;
};
struct BExpr {
    struct Expr *l;
    struct Expr *r;
};
```

`struct Expr` is the type of an expression, and its field `kind` indicates the kind of expression: `0` for constant 1, `1` for negation, `2` for addition, and `3` for multiplication. The inner union value `v` stores the necessary data for each kind of expression. When `kind` is `1`, the operand of negation is stored in `v.e`; when `kind` is `2` or `3`, the operands are stored in `v.b.l` and `v.b.r`. Since `kind` signifies which union field has been written to, it is the tag field for the union.

A function evaluating an expression is implemented as follows:

```
int eval(struct Expr *e) {
    switch (e->kind) {
        case 0:
            return 1;
        case 1:
            return -eval(e->v.e);
        case 2:
            return eval(e->v.b.l) + eval(e->v.b.r);
        case 3:
            return eval(e->v.b.l) * eval(e->v.b.r);
        default:
            exit(1);
    }
}
```

It evaluates the expression by checking the `kind` field and accessing the appropriate union field accordingly. When `kind` is `1`, it accesses the union field `e`; when `kind` is `2` or `3`, it accesses `b`. No field is accessed otherwise.

Programs sometimes use `if` to check tags, particularly to compare with a specific tag value. An example using `if` is shown below:

```
if (e->kind == 1)
    return -eval(e->v.e);
```

While tag fields are beneficial for accessing the correct union fields, they cannot prevent memory bugs. For example, the following code accesses `e->v.b` despite `kind` being `1`:

```
switch (e->kind) {
    case 1:
        return eval(e->v.b.l) + eval(e->v.b.r);
```

Here, `e->v.b.l` accesses the pointer stored in `e->v.e`, but `e->v.b.r` reads an arbitrary value, potentially causing invalid memory access. Furthermore, even when `eval` is correctly implemented, an incorrect tag value can be assigned during the construction of an `Expr`. The following code sets `e.kind` to `2` but writes to `e.v.e`:

```
struct Expr e;
e.kind = 2;
e.v.e = ...;
```

Passing a pointer to `e` to `eval` results in invalid memory access.

3.1.2 C2Rust's Translation

C2Rust translates the definition of `struct Expr` to Rust as follows:

```
struct Expr {
    kind: i32,
    v: C2RustUnnamed,
}
union C2RustUnnamed {
    e: *mut Expr,
    b: BExpr,
}
struct BExpr {
    l: *mut Expr,
    r: *mut Expr,
}
```

Since Rust does not support anonymous types, C2Rust gives the name `C2RustUnnamed` to the union. If a file contains multiple anonymous types, they get `C2RustUnnamed_`n where n is a unique integer.

The function `eval` is translated as follows:

```
fn eval(e: *mut Expr) -> i32 {
    match (*e).kind {
        0 => {
            return 1;
        }
        1 => {
            return -eval((*e).v.e);
        }
        2 => {
            return eval((*e).v.b.l) + eval((*e).v.b.r);
        }
        3 => {
            return eval((*e).v.b.l) * eval((*e).v.b.r);
```

```
        }
        _ => {
            exit(1);
        }
    }
}
```

Since Rust provides `match` statements instead of `switch`, the function employs `match` to check the tag. While `match` is mainly used for pattern matching on tagged unions, it can also handle integers, similar to `switch`, but without fall-through behavior.

3.1.3 Tagged Unions

We can implement the same syntax using tagged unions as follows:

```
struct Expr { v: C2RustUnnamed }
enum C2RustUnnamed {
    One,
    Neg(*mut Expr),
    Add(BExpr),
    Mul(BExpr),
}
struct BExpr {
    l: *mut Expr,
    r: *mut Expr,
}
```

In Rust, the `enum` keyword defines tagged unions. Although the name `C2RustUnnamed` is impractical, we retain it for consistency with the C code. A tagged union's definition lists its *variants*, i.e., values with distinct tags. Each tag is an identifier, not an integer, and the type of a value associated with each tag is specified after the tag. The defined tagged union has four variants with tags `One`, `Neg`, `Add`, and `Mul`. The `Neg` tag is associated with an `Expr` pointer, while `Add` and `Mul` are associated with a `BExpr` value. Since `C2RustUnnamed` now contains the tag, the `kind` field in the struct is no longer necessary.

Now, `eval` can be implemented with pattern matching as follows:

```
1  fn eval(e: *mut Expr) -> i32 {
2      match (*e).v {
3          C2RustUnnamed::One => {
4              return 1;
5          }
6          C2RustUnnamed::Neg(e) => {
7              return -eval(e);
8          }
9          C2RustUnnamed::Add(b) => {
10             return eval(b.l) + eval(b.r);
11         }
12         C2RustUnnamed::Mul(b) => {
13             return eval(b.l) * eval(b.r);
14         }
15     }
```

```
16 }
```

Each pattern matching branch specifies a tag and binds the associated value to an identifier. For instance, in lines 6 and 7, the associated value is bound to `e` and then passed to `eval`. Since the compiler ensures all variants are covered, the `_` (default) branch is unnecessary.

The code pattern using `if` to check tags is also supported through `if-let` [11], which is another form of pattern matching. The following code performs computation only when the tag is `Neg`:

```
if let C2RustUnnamed::Neg(e) = (*e).v {
    return -eval(e);
}
```

Pattern matching enables the compiler to detect programmers' mistakes that can cause memory bugs. Consider the following Rust code where the value associated with `Neg` is treated as a `BExpr` type:

```
match (*e).v {
    C2RustUnnamed::Neg(b) => {
        return eval(b.l) + eval(b.r);
    }
```

The type of `b` is `*mut Expr`, as specified in the type definition. Since `*mut Expr` lacks the fields `l` and `r`, the compiler raises an error.

Additionally, the compiler verifies the correct construction of tagged union values. Consider the following code, which incorrectly initializes a tagged union with the tag `Add` and an `Expr` pointer:

```
let e1: Expr = ...;
let e2 = Expr { v: C2RustUnnamed::Add(&mut e1) };
```

Since the type definition requires a `BExpr` value for `Add`, the code does not pass type checking.

3.2 Static Analysis

In this section, we present static analysis designed to facilitate the transformation of unions with tags into tagged unions. The objectives of this analysis are to (1) identify a tag field for a union, if one exists, and (2) determine the tag values associated with each union field. The proposed static analysis consists of four steps:

1. Identification of *candidate structs*, those containing unions and their potential tag fields (Section 3.2.1).
2. Whole-program may-points-to analysis (Section 3.2.2).
3. Intraprocedural must-points-to analysis for selected functions (Section 3.2.3).
4. Interpretation of the analysis results using a heuristic (Section 3.2.4).

As we analyze Rust code generated by C2Rust, rather than the original C code, code examples in this section are written in Rust.

3.2.1 Candidate Identification

The first step of the analysis is to identify structs that likely contain unions and their tag fields. We also determine which functions to analyze based on the identified structs. If no such structs are found, the analysis terminates at this step, concluding that the program does not contain unions to be transformed into tagged unions.

To concretely define candidate structs, we first define *tag-eligible fields*, those that can potentially serve as tag fields. A field of a struct is considered tag-eligible if (1) it has an integer type, (2) when it appears on the left side of an assignment, the operator is `=`, excluding others such as `+=`, and (3) it is never referenced by a pointer. For the first condition, integer types include `bool`, `i8`, `u8`, `i16`, `u16`, `i32`, `u32`, `i64`, and `u64`, which C2Rust translates from `_Bool`, `signed`/`unsigned char`, `short`, `int`, and `long`. The second and third conditions arise from limitations in the expressibility of tagged unions. Tags of tagged unions are not integers and thus cannot undergo integer operations. Moreover, since tags do not exist as fields, they cannot be referenced. Consequently, fields that exhibit such behavior cannot be transformed into tagged unions.

We also define *candidate unions*, which can potentially be accompanied by tags. A union is a candidate if (1) it is a field of a struct that contains at least one tag-eligible field, and (2) its name begins with `C2RustUnnamed`. The second condition indicates that the union is anonymous in the C code. If a union has a name, it can be used independently of the struct, not being expressible as tagged unions.

We finally define candidate structs. A candidate struct is a struct containing at least one candidate union.

We now describe how to determine which functions to analyze. To identify tag fields for unions and values associated with union fields, we need to ascertain the possible values of candidate struct fields. If a function neither reads from nor writes to a field, intraprocedural analysis of the function provides no information about that field's value. Therefore, we analyze only the functions that access fields of candidate structs. Such functions can be easily identified through syntactic examination.

3.2.2 May-Points-To Analysis

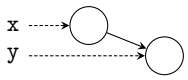
The second step of the analysis is to conduct a whole-program may-points-to analysis. This step is essential because may-points-to relations are required to ensure the soundness of the subsequent must-points-to analysis. The utilization of may-points-to relations in the must-points-to analysis is detailed in Section 3.2.3.

We employ a field-sensitive Andersen-style analysis [167] for the may-points-to analysis. This analysis is flow-insensitive and has a time complexity of $O(n^3)$. Although other may-points-to analyses exist, such as field-insensitive Andersen-style [52] and Steensgaard-style analyses [186, 185], they are too imprecise, leading to unacceptably imprecise must-points-to analysis results.

3.2.3 Must-Points-To Analysis

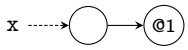
The third step is to perform an intraprocedural must-points-to analysis for each selected function. Our overall algorithm is akin to typical must-points-to analyses [118]. The execution state at each program point is represented as a graph, with nodes denoting memory locations and edges expressing points-to relations. The analysis iteratively updates the state at each program point until reaching a fixed point. Each state is derived by joining the state from the previous iteration with the state resulting from applying the current instruction's effect to the previous program point's state.

We first describe how we visualize graphs throughout this section. Since nodes denote memory locations, some nodes correspond to the stack locations used by local variables. For clarity in visualization, we draw a dashed arrow from the name of a local variable to the node representing its memory location. Note that a variable name is not a node and thus this arrow is not an edge. For example, `x = &y` constructs the following graph:

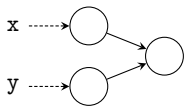


The graph has two nodes and one edge and indicates that the pointer at the memory location of `x` points to the memory location of `y`.

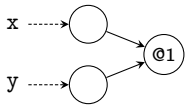
Since our goal is not only to compute must-points-to relations but also to track fields' integer values, we optionally label each node with `@`N, where N is a set of integers. The label `@`N encodes integer value information in pointer graphs. If a node is unlabeled, the value at the location is a usual C value (such as an integer, pointer, struct, union, or array). If a node is labeled `@`N, the value at the location is an imaginary value at addresses N, i.e., the possible addresses of the location are N. Consequently, if a node v has an outgoing edge to a node labeled `@`N, then the possible values at v's location are N. For instance, `x = 1` constructs the following graph:



Using the `@`N label is beneficial since it enables efficient propagation of integer values to memory locations known to hold the same value. Consider the following graph, constructed by `x = y`:



If we obtain the fact that `x` equals 1, we update the graph as follows:

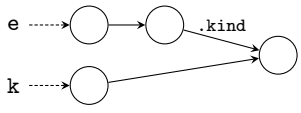


Then, we automatically discover that `y` also equals 1. As this example demonstrates, the `@`N label facilitates the update of the value at multiple memory locations by labeling only a single node.

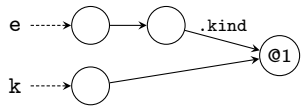
We now discuss how to analyze code involving unions. Consider the following example, where `Expr` is defined as in Section 3.1.2:

```
1 fn eval(e: *mut Expr) -> i32 {
2     let k = (*e).kind;
3     match k {
4         1 => {
5             return -eval((*e).v.e);
6         }
```

After line 2, we have the following graph:



Here, the edge labeled `.kind` indicates the presence of a struct/union at the location of the node where the edge originates, with the pointer stored in the `kind` field referring to the pointed node's location. Since line 3 uses `k` as the condition for `match`, we determine that `k` equals `1` in line 5. Consequently, the graph at the beginning of line 5 is as follows:



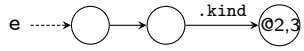
From this graph, we conclude that the value of `kind` is `1` when the union field `e` is accessed in line 5.

More complex scenarios involve joining two graphs. Consider the following code example:

```
1  match (*e).kind {
2      2 => {
3          ...
4      }
5      3 => {
6          ...
7      }
8      _ => {
9          return;
10     }
11 }
12 let lv = eval((*e).v.b.l);
```

In lines 3 and 6, we have the following graphs, respectively:

Since line 12 is reachable from both lines 3 and 6, we need to join the graphs. When joining graphs, the integer sets in the labels are unioned, resulting in the following graph:



From the graph, we conclude that the value of `kind` is `2` or `3` when the union field `b` is accessed in line 12.

The join of graphs is carefully defined to maintain the soundness of the analysis. Consider the following code, where control flow splits based on the value of `kind` and then merges:

```
1 if (*e).kind == 1 {
2     ...
3 } else {
4     ...
5 }
6 ...
```

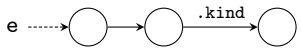
The states in lines 2 and 4 are as follows, respectively:

In line 2, `(*e).kind` is known to be `1`. However, in line 4, its value is unknown. When entering line 6, these graphs are joined as follows:

Algorithm 3.1: Graph joining

Input : g_1, g_2
Output: g

1 g.nodes := $\emptyset$;
2 g.edges := $\emptyset$;
3 worklist := $\emptyset$;
4 **for** $x \leftarrow$ *function's local variables*:
5 **if** g_1 *has a node* v_1 *corresponding to* x:
6 **if** g_2 *has a node* v_2 *corresponding to* x:
7 g.nodes.insert((v_1, v_2));
8 worklist.insert((v_1, v_2));
9 **while** worklist $\neq \emptyset$:
10 (v_1, v_2) := worklist.pop();
11 **for** $(v_1, v_1', f) \leftarrow v_1$*'s outgoing edges in* g_1:
12 **if** g_2 *has an edge* (v_2, v_2', f):
13 g.edges.insert($(v_1, v_2), (v_1', v_2'), f$);
14 **if** $(v_1', v_2') \notin g$.nodes:
15 g.nodes.insert((v_1', v_2'));
16 worklist.insert((v_1', v_2'));
17 **if** v_1 *has a label* $@N_1$ *in* g_1:
18 **if** v_2 *has a label* $@N_2$ *in* g_2:
19 set (v_1, v_2)'s label to $@(N_1 \cup N_2)$ in g;



Since the node from line 4 is unlabeled, the joined graph's node is also unlabeled. This indicates that we do not know the value of `(*e).kind`, which is correct. This example shows that no-label signifies no-information regarding the address of the node's location, i.e., it represents $@\mathbb{Z}$ (all integers), not $@\emptyset$ (empty set).

Algorithm 3.1 illustrates the algorithm for graph joining. Edges are intersected to retain must-point-to relations, while integer sets in labels are unioned. In the pseudo-code, each node in the input graphs g_1 and g_2 is denoted by a unique identifier v, and each node in the output graph g is represented by a pair of identifiers (v_1, v_2). Each edge is represented by (v, v', f), denoting an edge labeled f from node v to node v'. Unlabeled edges are treated as edges with the empty label ϵ. Initially, g and the worklist, which stores g's nodes to be visited, are both empty (lines 1–3). Then, we add nodes corresponding to local variables to g (lines 4–8). Finally, we visit each added node until the worklist is empty (lines 9–10). During the visit, edges with the same label in g_1 and g_2 are added to g (lines 11–16), and node labels are unioned if they exist (lines 17–19).

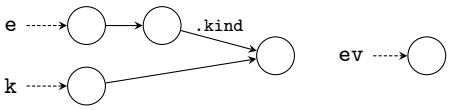
We now discuss the utilization of may-points-to relations during the must-points-to analysis. Consider the following code:

```
let e = if ... {
    &mut ev
} else {
```

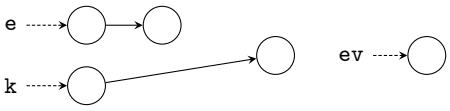
```
4       ...
5   };
6   let k = (*e).kind;
7   ev = ...;
8   match k {
9       1 => {
10          ...
11      }
```

Since `e` may not point to `ev`, the state after line 5 is as follows:

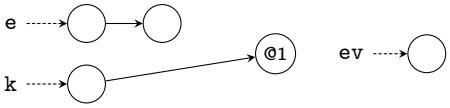
This graph indicates that `e` points to some location, not necessarily the same as `ev`'s location. Line 6 updates the graph as follows:



Line 7 mutates the value of `ev`, possibly changing the value of `(*e).kind`. Thus, `(*e).kind` is not necessarily equal to `k` after line 7, necessitating an appropriate update to the graph. As the graph itself does not reveal any relations between `e` and `ev`, we rely on the precomputed may-points-to relations, which indicate that `e` may point to `ev`. Consequently, we remove all outgoing edges, including the `kind` edge, from `*e`'s node, resulting in the following graph:



In line 10, we successfully avoid the incorrect conclusion that `(*e).kind` equals `1`, as shown in the following graph:



Like this, the analysis removes the appropriate edges from the graph at each (indirect) assignment and function call according to the may-points-to relations. The effect of a function call is equivalent to the cumulative effects of all assignments reachable by the call.

3.2.4 Analysis Result Interpretation

The last step of the analysis is to interpret the results of the must-points-to analysis using a heuristic, as demonstrated in Algorithm 3.2. The entry point is `IdentifyTags`, which takes three arguments: a candidate struct s, a candidate union u in s, and a tag-eligible field f_s in s. Its goal is to determine whether f_s serves as a tag field for u and, if it does, to identify the tag values associated with each field of u.

As the first step of the heuristic, we invoke `CollectFromAccesses` (line 1) to identify the associated tag values by examining the value of f_s when a union field is accessed after f_s has been checked by

Algorithm 3.2: Identifying tag values associated with fields

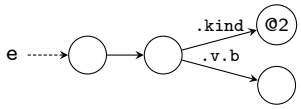
```
1  def IdentifyTags(struct s, union u, field f_s):
2      res := CollectFromAccesses(s, u, f_s);
3      if res = None:
4          return None;
5      (field_tags, access_tags) := res;
6      field_tags' := CollectFromStructs(s, u, f_s);
7      struct_tags := ∅;
8      for f_u ← fields of u:
9          tags := field_tags'[f_u] \ access_tags;
10         if struct_tags ∩ tags ≠ ∅:
11             return None;
12         field_tags[f_u] := field_tags[f_u] ∪ tags;
13         struct_tags := struct_tags ∪ tags;
14     all_tags := CollectAllTags(s, u, f_s);
15     rem_tags := all_tags \ (access_tags ∪ struct_tags);
16     return (field_tags, rem_tags);
17 def CollectFromAccesses(struct s, union u, field f_s):
18     field_tags := Map(); all_tags := ∅;
19     for l ← analyzed program points:
20         if a field f_u of u is accessed at l:
21             N := possible values of f_s at l;
22             if N is from if or match:
23                 tags := N \ field_tags[f_u];
24                 if all_tags ∩ tags ≠ ∅:
25                     return None;
26                 field_tags[f_u] := field_tags[f_u] ∪ tags;
27                 all_tags := all_tags ∪ tags;
28     return (field_tags, all_tags);
29 def CollectFromStructs(struct s, union u, field f_s):
30     field_tags := Map();
31     for l ← analyzed program points:
32         if l is end of a basic block:
33             for v ← struct s reachable at l:
34                 if union u in v has a field f_u:
35                     N := possible values of f_s of v;
36                     field_tags[f_u] := field_tags[f_u] ∪ N;
37     return field_tags;
38 def CollectAllTags(struct s, union u, field f_s):
39     all_tags := ∅;
40     for l ← analyzed program points:
41         for v ← struct s reachable at l:
42             N := possible values of f_s of v;
43             all_tags := all_tags ∪ N;
44     return all_tags;
```

`match`/`if`. This subroutine returns field_tags, a map from union fields to their tag values, and all_tags, a set containing all tag values in field_tags. Initially, both are empty (line 18). We then iterate over every program point in the analyzed functions and check if any field of u is accessed (lines 19–20). If accessed, we determine the possible values of f_s from the @N label of the graph computed by the must-points-to analysis (line 21). Here, we treat no-label as the empty set. The possible values should originate from `match`/`if` on f_s, and not from an assignment to f_s (line 22). If any of these values are already associated with other fields, we immediately return None, indicating that f_s is not a tag field (lines 23–25). Otherwise, we add the values to field_tags and all_tags (lines 26–27).

If `CollectFromAccesses` succeeds, `IdentifyTags` proceeds to the next step by calling `CollectFromStructs` (line 6). This complements the previous step by discovering tag-field associations that may have been missed due to the analysis's imprecision. This subroutine considers the field values and the last-written union fields at the end of each basic block. For instance, consider the following code:

```
(*e).kind = 2;
(*e).v.b = ...;
return e;
```

We have the following graph at return:

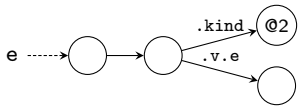


From this, we can conclude that `kind` equals `2` when `b` is the last-written union field, likely associating `2` with `b`.

We inspect only the states at the end of basic blocks because the states of other program points are prone to provide incorrect information from *intermediate* states. Consider the following example:

```
1 (*e).kind = 1;
2 (*e).v.e = ...;
3 ...
4 (*e).kind = 2;
5 (*e).v.b = ...;
6 return e;
```

Initially, `e` is used as a negation expression by setting `kind` to `1`, but it becomes an addition expression by setting `kind` to `2` in the end. If we examine the state after line 4, we will get an incorrect association between `2` and `e` from the following graph:



To avoid this issue, we examine only the end of basic blocks, where both the tag value and the union field are likely to be correctly set.

However, this approach cannot completely prevent reading intermediate states. For example, a function call can occur between the tag set and the union field set. Therefore, we prioritize the data from `CollectFromAccesses`, which are less likely to be affected by intermediate states, over `CollectFromStructs`. If `CollectFromAccesses` associates a certain tag value with a specific union field, that tag value in `CollectFromStructs`' results is ignored.

To implement this approach, `CollectFromStructs` iterates over the end of every basic block in each analyzed function, identifying every value of type s reachable from local variables at that point (lines 31–33). If the struct's union has a written field, that field is associated with the values of f_s (lines 34–36). Then, `IdentifyTags` combines results from both subroutines, prioritizing data from `CollectFromAccesses` (lines 8–13). It removes tags already associated with union fields by `CollectFromAccesses` from the results of `CollectFromStructs` (line 9) and returns None if a tag value remains associated with two fields even after this removal (lines 10–11).

Finally, we search for tag values not associated with any union fields. To achieve this, we call `CollectAllTags` (line 14), which collects all possible tag values by examining the state at every program point (lines 40–43). By removing the tags associated with union fields, we isolate the tags not associated with union fields (line 15).

To determine the tag field for each union, we run `IdentifyTags` on all tag-eligible fields. The field for which `IdentifyTags` returns a value other than None is identified as the tag field. If multiple tag fields are found, we select the one with the highest number of distinct tag values.

3.3 Code Transformation

In this section, we present code transformation that replaces unions with tagged unions in C2Rust-generated code, using the results of the static analysis. We demonstrate the transformation of the `Expr` type defined in Section 3.1.2 as an example. We assume that the tag field and the tag values associated with each union field are correctly identified by the static analysis.

The type definitions are transformed as follows:

```
struct Expr { v: C2RustUnnamed }
enum C2RustUnnamed {
    Empty0,
    e1(*mut Expr),
    b2(BExpr),
    b3(BExpr),
}
struct BExpr {
    l: *mut Expr,
    r: *mut Expr,
}
```

This result is the same as the hand-written code in Section 3.1.3, except for the variant names. We generate variant names by concatenating the union field name with the tag value. For tags not associated with any fields, we prepend `Empty` to the tag value. To improve variant names, one option is to utilize global variables' names. When using unions with tag fields, C programmers often define an *enum*, i.e., a group of constant integers with associated names, to use them as tag values. These names typically reflect the programmers' understanding of the tag values' meaning, e.g., `EXPR_ONE`. Since C2Rust translates each enum definition into multiple constant global variable definitions while preserving the names, we can use these variable names instead of the union field names. Although the names generated with this strategy may still be unsatisfactory, developers can easily rename them to more meaningful names using IDEs.

We now discuss the transformation of code using unions. We propose two approaches: naïve transformation, which can be applied to any code but does not adhere to Rust idioms (Section 3.3.1), and

idiomatic transformation, which follows Rust idioms but is applicable only to specific code patterns (Section 3.3.2). We use both methods within a single codebase, prioritizing idiomatic transformation wherever possible and resorting to naïve transformation when necessary.

3.3.1 Naïve Transformation

Naïve transformation involves defining helper methods for the transformed structs and unions. These methods are categorized into two groups: reading and writing. We first focus on reading. Below are the read-related methods for `Expr` and `C2RustUnnamed`:

```
1  impl Expr {
2      fn kind(self) -> i32 {
3          match self.v {
4              C2RustUnnamed::Empty0 => {
5                  0
6              }
7              C2RustUnnamed::e1(_) => {
8                  1
9              }
10             ...
11         }
12     }
13 }
14 impl C2RustUnnamed {
15     fn get_e(self) -> *mut Expr {
16         if let C2RustUnnamed::e1(v) = self {
17             v
18         } else {
19             panic!()
20         }
21     }
22 }
```

The `kind` method (lines 2–12) of `Expr` replaces the `kind` field in the original code. This method returns the appropriate tag value by applying pattern matching to the tagged union value. Code that reads a tag field is replaced with a call to the tag-returning method.

The `get_e` method (lines 15–21) of `C2RustUnnamed` replaces the union field `e`. This method returns a value by applying pattern matching to the tagged union value. If the current variant does not contain such a value, the method triggers a panic. This allows the dynamic detection of potential bugs by identifying read from a field other than the last-written one, rather than silently reinterpreting the value. Although not shown in the example, a method `get_`f is defined similarly for each union field f. We replace code reading a union field with a call to the corresponding getter method.

Using these methods, we transform code reading tags and union fields as follows, where the former represents the code before transformation and the latter represents the code after transformation:

```
// before
match (*e).kind {
    1 => {
        eval((*e).v.e);
```

```
    }
```

```
// after
match (*e).kind() {
    1 => {
        eval((*e).v.get_e());
    }
```

Although this transformation preserves the semantics, the resulting code is not idiomatic. It applies pattern matching twice to the tagged union value—once to get the tag value and once to get the union field value—instead of applying pattern matching only once to directly access the associated value of each variant.

We now describe the write-related methods, defined as follows:

```
1  impl Expr {
2      fn set_kind(&mut self, v: i32) {
3          match v {
4              0 => {
5                  self.v = C2RustUnnamed::Empty0;
6              }
7              1 => {
8                  let v = if let C2RustUnnamed::e1(v) = self.v {
9                      v
10                 } else {
11                     std::ptr::null_mut()
12                 };
13                 self.v = C2RustUnnamed::e1(v);
14             }
15             ...
16         }
17     }
18 }
19 impl C2RustUnnamed {
20     fn deref_e_mut(&mut self) -> *mut *mut Expr {
21         if let C2RustUnnamed::e1(_) = self {
22         } else {
23             *self = Self::e1(std::ptr::null_mut());
24         }
25         if let C2RustUnnamed::e1(v) = self {
26             v
27         } else {
28             panic!()
29         }
30     }
31 }
```

The `set_kind` method (lines 2–17) of `Expr` replaces assignments to `kind`. It takes a tag value as an argument and updates the tagged union to the appropriate variant. If the variant has an associated value, we check if this value already exists and reuse it if it does (line 9). If the value does not exist, we create an arbitrary value, which in this case is the null pointer (line 11).

The `deref_e_mut` method (lines 20–30) of `C2RustUnnamed` provides a pointer to the inner value, which we use to replace code that mutates `e` or takes its address. First, we check whether the current variant is appropriate (line 21), and, if not, update the value to the correct variant (line 23). Then, we return a pointer to the value (line 26), while the `panic!()` in line 28 is never reached. We also define a method `deref_`f`_mut` for each union field f in a similar manner.

We transform code that updates tags and union fields as follows:

```
// before
(*e).kind = 1;
(*e).v.e = ...;
```

```
// after
(*e).set_kind(1);
*(*e).v.deref_e_mut() = ...;
```

In the transformed code, `set_kind` changes the variant to `e1`. Then, `deref_e_mut` returns a pointer to the arbitrary value set by `set_kind`, and the indirect assignment to the pointer updates this value.

Note that `deref_e_mut` does not trigger a panic even when the current variant differs from the expected one, unlike `get_e`. This behavior ensures the correct transformation of code that first writes to a union field and then sets the tag. For example, the following transformation preserves the semantics:

```
// before
(*e).v.e = ...;
(*e).kind = 1;
```

```
// after
*(*e).v.deref_e_mut() = ...;
(*e).set_kind(1);
```

After the transformation, `deref_e_mut` changes the variant to `e1`, and the indirect assignment to the pointer sets the associated value. Then, `set_kind` retains both the variant and associated value. Although this approach preserves the semantics, it is not idiomatic in Rust, as we can create a tagged union value within a single expression instead of setting the tag and the union field separately.

3.3.2 Idiomatic Transformation

Idiomatic transformation uses pattern matching on tagged union values and constructs a tagged union value with a single expression, avoiding helper methods. Below is the transformation of `match`:

```
// before
match (*e).kind {
    1 => {
        eval((*e).v.e);
    }
```

```
// after
match (*e).v {
    C2RustUnnamed::e1(ref x) => {
        eval(*x);
    }
```

The `match` condition is the tagged union value itself, rather than the tag value obtained by `get_kind`. In addition, each branch directly matches the variant, instead of comparing the tag value to an integer. The `ref` keyword before the identifier `x` binds a pointer to the associated value, not the value itself, to `x`, allowing the branch to read and modify the value. This eliminates the need to call methods such as `get_e` and `deref_e_mut`, as `x` directly accesses the value.

We also transform `if` that checks tag values to use pattern matching with `if-let`, as illustrated in the following example:

```
// before
if (*e).kind == 1 {
    eval((*e).v.e);
}
```

```
// after
if let C2RustUnnamed::e1(ref x) = (*e).v {
    eval(*x);
}
```

We handle the disjunction of multiple equality comparisons using `|`, the *or* operator in pattern matching. An example is shown below:

```
// before
if (*e).kind == 2 || (*e).kind == 3 {
    eval((*e).v.b.l);
}
```

```
// after
if let C2RustUnnamed::b2(ref x) | C2RustUnnamed::b3(ref x) = (*e).v {
    eval((*x).l);
}
```

However, the idiomatic approach is not applicable to other kinds of conditions, such as conjunctions and inequalities, because Rust's patterns currently cannot represent these conditions. A conjunction specifies that the tag is a particular integer and also that some boolean formula is satisfied. When using `match` for pattern matching, Rust offers *match guards* [18] to express such logic. However, when using `if-let`, this logic is supported through *let chains* [86], which is currently an unstable feature and requires a special flag to enable. For this reason, we chose not to transform conjunctions into pattern matching in this work, leaving it for future work once this feature is stabilized. On the other hand, an inequality specifies that the tag is not a particular integer. However, the purpose of pattern matching is to check whether a value conforms to a certain pattern, not that it does not, and thus it cannot replace inequalities. Thus, the following code is transformed using the naïve approach:

```
if (*e).kind == 1 && ... {
    eval((*e).v.e);
}
if (*e).kind != 1 {
    ...
} else {
    eval((*e).v.e);
}
```

The idiomatic transformation is also not applicable when union fields are accessed without using `match` or `if` to check the tag field. C programmers sometimes assume that a tag field has a specific value at a certain point based on their understanding of the program's behavior and directly access the union field without checking the tag. Since tag check is not performed, we cannot transform such code into pattern matching and must resort to the naïve transformation.

The idiomatic transformation consolidates multiple assignment expressions within a single code block into a single assignment expression that constructs a tagged union value if the assignments set the tag and union fields. Below illustrates this transformation:

```
// before
{
    ...
    (*e).kind = 1;
    (*e).v.e = ...;
    ...
}
```

```
// after
{
    ...
    (*e).v = C2RustUnnamed::e1(...);
    ...
}
```

When multiple assignments are distributed across different code blocks, they are individually transformed using the naïve approach.

3.4 Evaluation

In this section, we evaluate our approach with 36 real-world C programs. We first describe our implementation of Urcrat, which realizes the proposed approach (Section 3.4.1), and the process of collecting the benchmark programs (Section 3.4.2). We then assess our approach by addressing the following research questions:

- RQ1. Precision and recall: Does it identify tag fields without false positives or false negatives? (Section 3.4.3)
- RQ2. Correctness: Does it transform code while preserving its semantics? (Section 3.4.4)
- RQ3. Efficiency: Does it efficiently analyze and transform programs? (Section 3.4.5)
- RQ4. Code characteristics: How much does the code change due to the transformation, and how frequently are the helper methods called? (Section 3.4.6)
- RQ5. Impact on performance: What is the effect of replacing unions with tagged unions on program performance? (Section 3.4.7)

Our experiments were conducted on an Ubuntu machine with Intel Core i7-6700K (4 cores, 8 threads, 4GHz) and 32GB DRAM. Finally, we discuss potential threats to validity (Section 3.4.8).

3.4.1 Implementation

We built Urcrat on top of the Rust compiler [12]. Urcrat analyzes Rust code after it has been lowered to Rust's mid-level intermediate representation (MIR) [15], which expresses functions as control flow graphs with basic blocks. For code transformation, Urcrat utilizes Rust's high-level intermediate representation (HIR) [14], akin to abstract syntax trees but with syntactic sugar removed and symbols resolved. We employed C2Rust v0.18.0 with minor modifications.

3.4.2 Benchmark Program Collection

We collected benchmark programs from three sources: (1) Crown [203], (2) GNU packages [6], and (3) GitHub. Only 2 out of 20 programs used by Crown have candidate unions, so we expanded the benchmarks with additional sources. We chose GNU packages for their representative C projects and GitHub for its diverse code patterns. We avoided large codebases because C2Rust often produces Rust code with type errors, primarily due to missing type casts, which require significant manual effort to correct. From GNU, we gathered C packages with less than 250k LOC and individual Wikipedia entries, indicating they are well-known. From GitHub, we gathered C projects with less than 1 MB of code and over 1,000 stars. In both collections, we retained programs (1) compiled successfully on Ubuntu, (2) transpiled successfully by C2Rust, and (3) containing candidate unions. This resulted in 24 programs from GNU and 10 from GitHub, giving us a total of 36 benchmark programs. Of these, 20 required manual fixes after C2Rust's translation, with an average of 26.6 lines modified. Columns 2–5 of Table 3.1 present the C LOC, Rust LOC, number of unions, and number of candidate unions in each program, respectively. In the benchmarks, 21% of unsafe functions involve unions, and 11% involve tagged unions.

3.4.3 RQ1: Precision and Recall

We evaluate the precision and recall of the proposed approach. We first identified the tag field of each candidate union through static analysis. Column 6 of Table 3.1 shows the number of candidate unions for which a tag field is identified in each benchmark program. Out of 141 candidates across 36 programs, Urcrat identifies tag fields for 74 candidates in 29 programs. We then manually inspected each candidate union to determine the presence of a tag field, checking for false positives (a tag field identified for a union that does not have one) and false negatives (no tag field identified for a union that has one). For practical use of our tool, false positives are problematic because they change the program's semantics. Conversely, false negatives are less problematic as they only prevent the replacement of unions with tagged unions, and the semantics remains correct.

Our manual inspection shows that the static analysis is precise, revealing no false positives and only five false negatives. This results in a precision of 100% and a recall of 93.7% (= 74/79). Specifically, two false negatives occur in `glpk-5.0`, while the others are in `gawk-5.2.2`, `screen-4.9.0`, and `uucp-1.07`, respectively. Three of these (from `glpk-5.0`, `gawk-5.2.2`, and `screen-4.9.0`) are due to intermediate states involving tag values that are not associated with any union fields by `CollectFromAccesses`. We now discuss the reasons for the remaining two false negatives:

glpk-5.0 The false negative arises from `goto` in the C code:

```
switch (tab->type) {
    case 112:
```

Table 3.1: Benchmark programs for evaluating Urcrat

Program	C LOC	Rust LOC	#Unions	#Candidates	#Identified
bc-1.07.1	10810	16982	4	1	1
binn-3.0**	5686	4298	1	1	0
brotli-1.0.9**	13173	127691	6	4	0
cflow-1.7	20601	26375	5	4	3
compton*	8748	14084	2	2	2
cpio-2.14	35934	80929	10	4	3
diffutils-3.10	59377	95835	7	5	4
enscript-1.6.6	34868	78749	9	5	3
findutils-4.9.0	80015	139858	13	6	3
gawk-5.2.2	58111	140566	17	10	3
glpk-5.0	71805	145738	18	14	3
gprolog-1.5.0	52193	74381	5	2	0
grep-3.11	64084	84902	11	9	6
gsl-2.7.1	227199	422854	14	14	0
gzip-1.12	20875	21605	4	2	1
hiredis*	7305	14042	1	1	1
make-4.4.1	28911	36336	1	1	1
minilisp*	722	2149	1	1	1
mtools-4.0.43	18266	37021	2	1	0
nano-7.2	42999	74994	6	4	3
nettle-3.9	61835	82742	5	2	1
patch-2.7.6	28215	103839	3	1	1
php-rdkafka*	3771	28864	1	1	1
pocketlang*	14267	41439	4	3	3
pth-2.0.7	7590	12950	1	1	1
raygui*	1588	17218	1	1	1
rcs-5.10.1	28286	36267	1	1	1
screen-4.9.0	39335	72199	1	1	0
sed-4.9	48190	68465	8	7	4
shairport*	4995	10118	2	1	1
tar-1.34	66172	134972	16	12	9
tinyproxy*	5667	12825	5	2	2
twemproxy*	22738	74593	8	7	5
uucp-1.07	51123	77872	3	3	0
webdis*	14369	29474	2	2	2
wget-1.21.4	81188	192742	6	5	4
Total			204	141	74

*: from GitHub, **: from Crown

```
            goto input_table;
        case 119:
            goto output_table;
        default:
            abort();
    }
input_table:
    ...
    return;
output_table:
    ...
```

In this code, `type` is the tag field. Since Rust does not support `goto`, C2Rust translates the code as follows:

```
match (*tab).type {
    112 => {
        current_block = 1;
    }
    119 => {
        ...
    }
    _ => {
        abort();
    }
}
if current_block == 1 {
    ...
}
```

The resulting code has a variable, `current_block`, which mimics `goto`'s effect. As the analysis is path-insensitive, it concludes that both `112` and `119` are possible values in the true branch of `if`, hindering the identification of the tag field. To address this issue, we need to either analyze the original C code or modify the C code to avoid `goto`. We chose the latter and revised the code as follows:

```
switch (tab->type) {
    case 112:
        ...
        break;
    case 119:
        ...
        break;
    default:
        abort();
}
```

Urcrat can identify the tag field from the modified version.

uucp-1.07 This false negative is due to a bug in the C code:

```
if (qport->uuconf_ttype == 5) {
```

```
    if (qport->uuconf_u.uuconf_stli.zdevice != NULL)
        fprintf(e, "%s", qport->uuconf_u.uuconf_smodem.zdevice);
}
```

Here, `uuconf_ttype` is the tag field, and the union field associated with `5` is `uuconf_stli`. However, the code erroneously accesses `uuconf_smodem` in the `fprintf` statement. The function contains multiple `fprintf` statements, suggesting that this bug was likely introduced through copy-pasting. After correcting the code to access `uuconf_stli`, Urcrat successfully identifies the tag field.

3.4.4 RQ2: Correctness

We evaluate the correctness of our approach by checking whether the transformed program is compilable and exhibits the same behavior as the original. For the experiments, we used the fixed code for `glpk-5.0` and `uucp-1.07`, allowing the identification of additional tag fields. Consequently, we examined 30 programs: 29 identified without code fixes and 1, which is `uucp-1.07`, identified only after the code fix. All 30 programs are compilable after transformation. Among 23 programs with test suites, 17 pass the tests post-transformation. Those failed are `gawk-5.2.2`, `grep-3.11`, `make-4.4.1`, `minilisp`, `twemproxy`, and `wget-1.21.4`.

We manually investigated the reasons for the failures and found that only `gawk-5.2.2` and `twemproxy`'s failures result from imprecise identification of tag values in the static analysis. The other failures are due to two specific C code patterns requiring minor manual code fixes after transformation: (1) reading a union field other than the last-written one, and (2) code relying on memory layout. We now discuss the failure reasons for each program.

gawk-5.2.2 The failure arises from tag values not being identified due to intraprocedural analysis.

```
INSTRUCTION *bcalloc(int op, ...) {
    INSTRUCTION *cp = ...;
    cp->opcode = op;
    ...
    return cp;
}
```

In this code, `opcode` is the tag field, assigned the argument value. The analysis cannot identify values at the call-site of `bcalloc` as possible `opcode` values. This causes the transformed program to panic when an unknown tag value is passed to `set_opcode`.

twemproxy The failure is due to the lack of C library modeling, preventing identification of tag values.

```
getaddrinfo(n, s, &h, &ai);
si->family = ai->ai_family;
```

Here, `family` is the tag field, and `ai_family` determines its value, set by the `libc` function `getaddrinfo`. While `2` and `10` are possible values, the analysis fails to recognize them as tag values. This issue can be resolved by incorporating library modeling into the analysis.

grep-3.11 It fails because of reading a field not lastly written.

```
fetch_token(token, input, syntax);
c = token->opr.c;
```

```
if (token->type == 2)
    return;
```

In this code, `type` is the tag field, and `opr` is the union value. The code first reads `c` and then checks `type`, causing a panic when `get_c` is called in the transformed code. Swapping the order of these statements allows the transformed program to pass the tests.

make-4.4.1 This is also due to reading a field not lastly written.

```
struct function_table_entry {
    ...,
    int alloc_fn;
    union {
        fptr1 func_ptr;
        fptr2 alloc_func_ptr;
    } fptr;
};
if (!entry_p->fptr.func_ptr)
    abort();
```

The union has two fields, `func_ptr` and `alloc_func_ptr`, with `alloc_fn` as the tag field. The code checks if `func_ptr` is null regardless of `alloc_fn`'s value, exploiting that different function pointer types have the same size and null representation. After transformation, this causes a panic if `alloc_func_ptr` is the last-written field. The code can be fixed as follows to pass the tests post-transformation:

```
if (entry_p->alloc_fn == 0 && !entry_p->fptr.func_ptr ||
    entry_p->alloc_fn == 1 && !entry_p->fptr.alloc_func_ptr)
```

minilisp This failure is caused by different memory layouts of unions and tagged unions.

```
struct Obj {
    int type;
    int size;
    union { ... };
};
Obj *alloc(size_t size, ...) {
    size += 8;
    ...
}
```

The `alloc` function allocates an `Obj` in a global byte array. It determines the memory size by taking the size of a union field and adding `8` to account for the offset of the union within `Obj`. After transformation, the code becomes as follows:

```
struct Obj {
    size: i32,
    c2rust_unnamed: C2RustUnnamed_1
}
```

Now, each variant value is at offset `16` due to alignment requirements, resulting in `alloc` allocating insufficient memory. We corrected the code to allocate larger memory, enabling the transformed program to pass the tests.

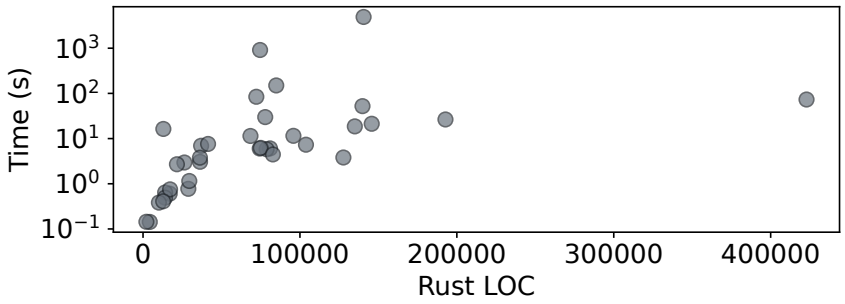


Figure 3.2: Execution time of Urcrat across benchmark programs

wget-1.21.4 It also stems from different memory layouts.

```
addr->family = 2;
memcpy(&addr->data, tmp, 4);
```

Here, `family` is the tag field, and `2` is associated with the union field `d4` of `addr->data`. The C program writes to `addr->data` because `addr->data` and `addr->data.d4` denote the same address. However, after transformation, `memcpy` overwrites the tag at offset `0` of `addr->data`. We could pass the tests by fixing the code as follows, facilitating `deref_d4_mut` to be called in the transformed code:

```
addr->family = 2;
memcpy(&addr->data.d4, tmp, 4);
```

Currently, for end users of the tool, detecting and fixing incorrect translations is challenging. To identify incorrect translations, they must manually inspect the code or run tests. If test suites do not exist, creating new test cases can be costly. Additionally, test cases may not always reveal incorrect translations. Fixing incorrect code is even more challenging, as users need to identify the root cause. The difficulty of this process depends on the root cause. When tag values are not correctly identified or fields not lastly written are read, investigating the cause is relatively straightforward. The test fails due to a panic, allowing users to check which tag value or field read triggered the panic. However, issues related to different memory layouts are more difficult to diagnose since they do not trigger panics. In such cases, users must rely on their debugging skills. We believe it would be beneficial to identify common incorrect translation patterns and design analyses to detect these patterns, thereby providing users with warnings. We leave this as future work.

3.4.5 RQ3: Efficiency

We evaluate the efficiency of the proposed approach by measuring the execution time of Urcrat, which includes both analysis and transformation, for each program. Figure 3.2 presents the execution time relative to the Rust LOC, with the y-axis displayed in a log scale due to the wide range of execution times. Urcrat efficiently handles most programs, with 31 programs taking less than a minute. The longest execution time is 4,910 seconds for `gawk-5.2.2`. Execution time shows only a weak correlation with code size, as other factors, primarily the complexity of pointer graphs and the number of analyzed functions, also significantly influence execution time.

We also investigate the effectiveness of our selective function analysis in terms of efficiency. For each program, the transformation consumes less than 1% of the total time, as it involves only tree-walking, while the analysis occupies the remaining time. This highlights the importance of reducing analysis

time. We calculated the percentage of analyzed functions relative to the total functions in each program, finding a geometric mean of 4.80%. In 28 programs, the analyzed functions constitute less than 10% of the total functions. The highest percentage is 58.23% in the case of `minilisp`. These results indicate that our approach is effective for efficiency.

3.4.6 RQ4: Code Characteristics

To understand the characteristics of the transformed code, we first evaluate the extent of code changes introduced. We measured the changes in the 30 programs where tag fields are identified. On average, the transformation added 861.8 lines per program due to helper method definitions. Excluding these, an average of 252.4 lines were inserted and 301.8 lines were deleted per program. These results demonstrate the practical utility of our approach, as manually implementing such significant code modifications would be both time-consuming and error-prone.

We also evaluate the applicability of the idiomatic transformation, which avoids inserting method calls. With only the naïve transformation, each program has 194.6 method calls on average. In contrast, applying the idiomatic transformation alongside the naïve transformation reduces this to 124.8 method calls per program, achieving a 36% decrease. Specifically, 38.3 calls are removed by replacing `match` on tag values with `match` on tagged unions, 15.8 calls by replacing `if` with `if-let`, and 15.6 calls by consolidating separate assignments into tagged union construction. The results indicate that the idiomatic transformation improves code quality.

3.4.7 RQ5: Impact on Performance

We evaluate the impact of replacing unions with tagged unions on the performance of the translated programs. We compare the performance of each Rust program before and after the transformation, measured in terms of the execution time of the test suite. To ensure reliable results, we excluded test suites with execution times shorter than 0.1 seconds, resulting in 20 programs, and computed the average execution time from fifty runs for each program.

The experimental results show that the performance overhead is negligible. On average, the transformed programs were only 0.01% slower than the original ones. Among the 20 programs, 5 ran slightly slower after transformation, while the others were faster. We also performed one-sided t-test with the null hypothesis that the program after transformation is slower than the original program by at least 1%. With a significance level of 0.05, we could reject the null hypothesis for 9 out of 20 programs, implying that the transformation is unlikely to incur observable performance overhead for them. For the remaining 11 programs, we could not reject the null hypothesis, but this does not necessarily mean they have an overhead greater than 1%. Repeating the experiments may provide stronger statistical significance, rejecting the null hypothesis.

3.4.8 Threats to Validity

Threats to external validity primarily stem from the choice of benchmarks. GNU packages often share common code patterns, which may introduce bias. Although we included projects from GitHub to enhance diversity, this may not fully represent the entire C ecosystem. Notably, large codebases might exhibit different characteristics. Further experiments with a broader range of C programs would provide greater confidence in the generalizability of our approach.

The type errors produced by C2Rust also pose a threat to external validity. Before using our tool, all type errors in C2Rust-generated code must be resolved. Larger programs tend to exhibit more type errors after C2Rust's translation, requiring significant effort to correct. This could discourage users from adopting both C2Rust and Urcrat, potentially impeding the widespread adoption of our approach. To mitigate this issue, we can improve C2Rust or develop techniques for fixing type errors in C2Rust-generated code.

Threats to construct validity arise from using test suites for correctness and performance evaluation. Passing all tests does not guarantee correctness. However, tests are the most widely used method for practical semantics validation and successfully discovered incorrect behavior in some programs during our evaluation. For performance assessment, the used test suites may be inadequate as they were not designed for performance measurement.

Chapter 4. Translation of Output Parameters

In Rust, tuples and `Option`/`Result` types [32, 27, 29] are useful, especially when used as return types for functions. Tuples naturally express functions that return multiple but fixed numbers of values. `Option` and `Result` types are employed to implement *partial functions*, i.e., functions that may fail. `Option` is either `Some(v)` or `None`; `Result`, akin to `Either` in other languages, is either `Ok(v)` or `Err(e)`, where `e` represents information related to the failure. These types find widespread usage in programming, even for elementary tasks. For instance, below shows two division functions, one returning both quotient and remainder and the other returning only the quotient but failing when the divisor is zero:

```
fn div(n: i32, d: i32) -> (i32, i32) {
    (n / d, n % d)
}
fn div(n: i32, d: i32) -> Option<i32> {
    if d == 0 {
        None
    } else {
        Some(n / d)
    }
}
```

Since C does not provide types equivalent to tuples and `Option`/`Result`, C programmers employ *output parameters* to implement such functions. Output parameters are pointer-type parameters used by functions to produce values rather than take inputs [144, 174]. Instead of returning a tuple consisting of two values, a function returns a single value and writes the other value to its output parameter. Instead of returning `Option`/`Result`, a function writes the result value to its output parameter only upon success, conveying information about success or failure through the return value. Below is the previous division functions re-implemented in C:

```
int div(int n, int d, int *r) {
    *r = n % d;
    return n / d;
}
int div(int n, int d, int *q) {
    if (d == 0)
        return 1;
    *q = n / d;
    return 0;
}
```

In the second function, a return value of `1` indicates failure, and `0` indicates success.

To bridge the gap between C and Rust, it is desirable to replace output parameters with tuples and `Option`/`Result` during C-to-Rust translation. Output parameters are discouraged in general because they often make the code less readable and more error-prone [144, 174]. Parameters essentially serve as inputs, and employing them for output purposes confounds code comprehension. Additionally, when output parameters are used to implement partial functions, they fail to express the possibility of failure

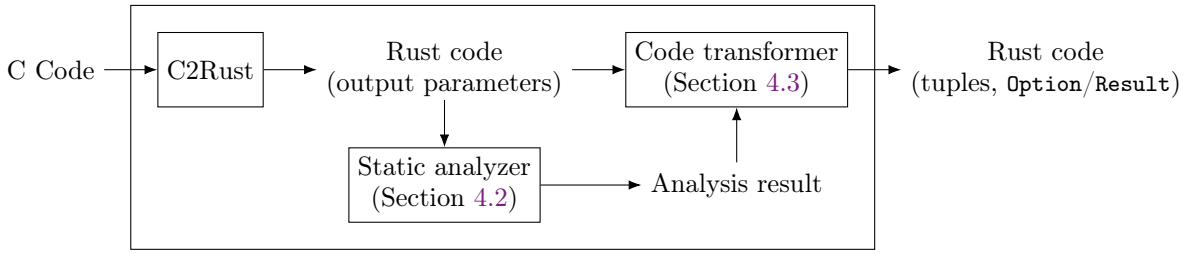


Figure 4.1: The workflow of Nopcrat

through types. Programmers may inadvertently attempt to read pointers passed to such functions even after they have failed. Consequently, in Rust, tuples and `Option`/`Result` are recommended over output parameters. Directly returning the outcome enhances code clarity, and `Option`/`Result` conveys the potential for failure at the type level, compelling callers to explicitly handle failure.

In this chapter, we present a technique for removing output parameters during C-to-Rust translation. Figure 4.1 illustrates the workflow of the proposed approach. We initially translate C code to Rust using C2Rust and subsequently enhance the resulting Rust code by removing output parameters. To facilitate this transformation, we extract information about output parameters through static analysis, which presents several challenges.

The first challenge is to identify output parameters of each function. To accomplish this, we propose a static analysis based on the abstract interpretation framework [73]. We introduce *abstract read/write sets*, which approximate the sets of pointer-type parameters that are read/written during execution. Abstract states contain not only abstract values for abstract memory locations but also abstract read/write sets. The output parameters of a function are determined by the abstract write set at the function's return point.

Second, we must determine whether each output parameter is always written. If that is the case, a `*mut` T parameter (a raw pointer to T) is removed by returning T; otherwise, the function is partial, and the parameter is removed by returning either `Option<`T`>` or `Result<`T`, `E`>`. Furthermore, in the latter scenario, we have to identify which return value indicates success/failure. This information is essential for transforming the call site inspecting the return value to use pattern matching on `Option`/`Result`. To achieve this, we introduce *write set sensitivity*, which distinguishes different sets of written pointers. By examining these distinguished write sets, we can determine whether each parameter is always written; by examining the return values associated with different write sets, we can determine the return values when a particular pointer is written and when not written.

Third, we have to differentiate the behavior of each function when a null pointer is given and when it is not. In C, a caller passes a null pointer as a value for an output parameter when it does not require the result, and the callee writes the result to the pointer only when it is non-null. Since the absence of writing to a null pointer cannot be deemed evidence of the function's partialness, we should check whether a pointer is always written, except when it is null. To accomplish this, we introduce *nullity sensitivity*, which distinguishes the nullity of pointers.

Overall, the contributions are as follows:

- We define output parameters and classify them (Section 4.1).
- We propose an efficient bottom-up static analysis based on abstract interpretation, accompanied by abstract read/write sets, write set sensitivity, and nullity sensitivity (Section 4.2).

- We propose code transformation removing output parameters using the analysis result (Section 4.3).
- We realize the proposed approach as a tool named Nopcrat and evaluate it with 55 real-world C programs. Our evaluation shows that the approach is (1) scalable, by analyzing and transforming 190k LOC within 213 seconds, (2) useful, by detecting 1,670 output parameters across the 55 programs, and (3) mostly correct, as 25 out of 26 programs pass their test suites after the transformation (Section 4.4).

4.1 Definition of Output Parameters

In this section, we define output parameters and provide illustrative code examples for clarity. Since determining whether a certain parameter is an output parameter depends on how the function introducing the parameter behaves, the definitions in this section only consider the events that occur between the moments a function is called and when it returns. For example, an execution means the execution of the specific function under consideration, not the entire program's execution. Given our objective of removing output parameters in Rust code generated by C2Rust, we present the code examples in Rust rather than C.

We begin by defining a *path*, which represents a memory location pointed by a pointer-type parameter. A parameter can refer to a struct value, with each field of the struct being a distinct memory location. Consequently, we define paths as follows:

Definition 4.1 (Path)**.** A *path* p is a pointer-type parameter x followed by zero or more field projections $l_1, \cdots, l_n$: $p \ ::= \ x \ \mid \ p.l$

Not every path represents a valid memory location because it may include non-existing fields. To filter out such invalid paths, we define *well-typed paths*:

Definition 4.2 (Well-typed path)**.** The *path-type* of x is T if the type of the parameter x is `*mut` T. The path-type of $p.l$ is T if the path-type of p is S and the struct S has a field l of type T. A path is *well-typed* if it has some path-type.

For illustration purposes, assume the following definition of `S`, used throughout this section:

```
struct S { a: i32, b: i32 }
```

If `x` has type `*mut S`, then `x` and `x.a` are well-typed, but `x.c` and `x.a.a` are not. Note that well-typed paths can point to overlapping memory regions. For example, `x` and `x.a` overlap because `x` refers to the entire struct, while `x.a` refers to its first field. To consider only non-overlapping paths, we define *maximal paths* and *x-path sets*:

Definition 4.3 (Maximal path and x-path set)**.** A path p is *maximal* if it is well-typed but $p.l$ is not well-typed for any l. An *x-path set* is the set of every maximal path starting with x.

For instance, if `x` has type `*mut i32`, then the `x`-path set is {`x`}; if `y` has type `*mut S`, then the `y`-path set is {`y.a`, `y.b`}. Finally, we define *reads* from and *writes* to maximal paths as follows:

Definition 4.4 (Read and write)**.** A *read* from a maximal path is obtaining the value at the memory location denoted by the path. A *write* to a maximal path is modifying the value at the memory location denoted by the path.

A read involves dereferencing a pointer, and a write entails an indirect assignment to a pointer:

- $(*x).l_1.\cdots.l_n$ performs a read from $x.l_1.\cdots.l_n$.
- $(*x).l_1.\cdots.l_n = v$ performs a write to $x.l_1.\cdots.l_n$.

In addition, reads and writes can also occur through aliased pointers and function calls. For example, in both of the following code snippets, `f` writes to `x`.

```
fn f(x: *mut i32) {
    let y = x;
    *y = 1;
}
```

```
fn f(x: *mut i32) {
    g(x);
}
fn g(y: *mut i32) {
    *y = 1;
}
```

When dealing with structs, a single expression can read from or write to multiple maximal paths. For instance, `f` writes to both `x.a` and `x.b` in the following code:

```
fn f(x: *mut S) {
    *x = S { a: 1, b: 2 };
}
```

Using the notion of a path, we now define different kinds of parameters. The following examples illustrate that reads do not necessarily prevent a parameter from being an output parameter, and writes do not necessarily make a parameter an output parameter:

```
fn f(x: *mut i32) {
    *x = 1;
    let v = *x;
    g(v);
}
```

```
fn f(x: *mut i32) {
    let v = *x;
    g(v);
    *x = 1;
}
```

In the former, `f` reads `x` after writing to it. Since the function never uses the value originally referred to by `x`, we can consider `x` an output parameter. In contrast, in the latter, `f` writes to `x` after reading it. This means that the function relies on the value referred to by `x`, and we cannot consider `x` an output parameter. Based on this observation, we define *effective reads and writes* as follows:

Definition 4.5 (Effective read and write)**.** An *effective read* from a path p is a read from p that is not preceded by any write to p. An *effective write* to a path p is a write to p that is not preceded by any read from p.

In the above examples, the former effectively writes to `x`, and the latter effectively reads from `x`. By definition, a single execution can either effectively read or effectively write to a path but not both. However, within a single function, one execution may effectively read a path, while another execution effectively writes to the path. For instance, in the following code, `f` effectively writes to `x` if `c` is true and effectively reads `x` otherwise:

```
fn f(x: *mut i32, c: bool) {
    if c {
        *x = 1;
    }
    let v = *x;
    g(v);
}
```

If a function effectively reads a path in some execution, the function requires the value pointed to by the pointer, implying that the parameter is used for input. Therefore, we define *input parameters*:

Definition 4.6 (Input parameter)**.** A parameter x is an *input parameter* if there is an execution in which at least one path in the x-path set is effectively read.

Furthermore, when dealing with structs, the absence of effective reads does not necessarily imply that the parameter is an output parameter. Consider the following code examples:

```
fn f(x: *mut S) {
    (*x).a = 1;
    (*x).b = 2;
}
```

```
fn f(x: *mut S) {
    (*x).a = 1;
}
```

In the former, `f` effectively writes to both `x.a` and `x.b`. Since this is equivalent to assigning `S { a: 1, b: 2 }` to `*x`, we can consider `x` an output parameter. In contrast, the latter implementation of `f` effectively writes to `x.a` but not to `x.b`. Since it does not provide a value for `x.b`, its intention is to mutate the state of the pointed struct rather than producing a new struct. Therefore, we cannot classify `x` as an output parameter. To distinguish whether all the fields are written or not, we define the concepts of *full, partial, and no writes*:

Definition 4.7 (Full/partial/no write)**.** A parameter x is *fully written* in an execution if every path in the x-path set is effectively written during the execution. A parameter x is *partially written* in an execution if some paths in the x-path set are effectively written and some paths are not during the execution. A parameter x is *not written* in an execution if none of the paths in the x-path set are effectively written during the execution.

We then define *mutation parameters*:

Definition 4.8 (Mutation parameter)**.** A parameter x is a *mutation parameter* if there is an execution in which x is partially written.

Note that if the type of x is `*mut` T, where T is a primitive type like `i32`, the x-path set is a singleton, and a partial write is impossible, rendering `x` incapable of becoming a mutation parameter.

Finally, we can define *output parameters*:

Definition 4.9 (Output parameter)**.** A parameter x is an *output parameter* if x is neither an input parameter nor a mutation parameter and there is an execution in which x is fully written.

We classify output parameters into two groups: *must* and *may*, as they require different transformations. A must-output parameter is always fully written by the function, and we have to replace a must-output parameter of type `*mut` T with a return type of T. Consider the following function:

```
fn f(x: *mut i32) {
    *x = 1;
}
```

The function always writes to `x`, leading to the following transformation:

```
fn f() -> i32 {
    1
}
```

Conversely, a may-output parameter is utilized by a partial function and may not be written. Its removal requires returning `Option<`T`>`/`Result<`T`, `E`>`. For instance, consider the following function:

```
fn f(x: *mut i32, c: bool) {
    if c {
        *x = 1;
    }
}
```

The function writes to `x` only if `c` is true, leading to this transformation:

```
fn f(c: bool) -> Option<i32> {
    if c {
        return Some(1);
    }
    None
}
```

An intricate case arises when a function checks whether a given pointer is null, as shown below:

```
fn f(x: *mut i32) {
    if !x.is_null() {
        *x = 1;
    }
}
```

It writes to `x` only when `x` is non-null, and the caller can ignore the outcome by passing a null pointer. Transforming this function to return `Option` presents challenges in determining the proper condition to replace `!x.is_null()` because there is no longer `x`:

```
fn f() -> Option<i32> {
    if ??? {
        return Some(1);
    }
    return None;
}
```

A more appropriate transformation is to return just `i32`:

```
fn f() -> i32 {
    1
}
```

This transformation aligns better with the original function's intent. The original function does not write to `x` because the caller does not require the result, not because the function has failed. Consequently, we do not classify `f` as a partial function, and `x` should be a must-output parameter. Based on these observations, we define *must- and may-output parameters*:

Definition 4.10 (Must-/may-output parameter)**.** An output parameter x is a *must-output parameter* if x is fully written in every execution in which x is non-null. An output parameter x is a *may-output parameter* if there is an execution in which x is non-null but not fully written.

Unfortunately, we cannot remove every output parameter. We now discuss four reasons that make output parameters unremovable. Note that we do not claim the reasons to be exhaustive. They are derived from our analysis of real-world C code, in which we identified output parameters and investigated whether each could be removed without altering the program's behavior. As our observations span 55 C programs, used for evaluation in Section 4.4, we believe they cover most scenarios where output parameters cannot be removed, but different reasons can occur (e.g., see Section 4.4.5).

Array Pointers We cannot remove an output parameter pointing to an array. For example, in the following code, `x` points to an integer array:

```
fn f(x: *mut i32, len: usize) {
    for i in 0..len {
        *x.offset(i) = i;
    }
}
```

The second parameter `len` denotes the array's length, and the function fills the array with integers from `0` to `len - 1`. Removing `x` requires the transformed function to return an array of length `len`, which Rust disallows, as the size of the return value must be known at compile time. One option is to return `Vec`, a pointer to a heap-allocated array in Rust, but we consider this beyond the scope of this work. In C-to-Rust translation, replacing dynamically sized arrays with `Vec` is an important goal, even for code without output parameters. Addressing this involves transforming array operations into corresponding `Vec` operations, which is challenging. Therefore, we defer the treatment of array pointers to future research, treating them as unremovable in this work.

Void Pointers We cannot remove output parameters of type `*mut c_void`, equivalent to `void *` in C. C programmers typically use the `void *` type to produce values of varying types depending on the context. For example, the following function returns either a 32-bit or 64-bit integer:

```
fn f(x: *mut c_void, c: bool) {
    if c {
        *(x as *mut i32) = i32::MAX;
    } else {
        *(x as *mut i64) = i64::MAX;
    }
}
```

Transforming this function necessitates returning either `i32` or `i64`, which Rust disallows.

Null-Specific Behavior If a function exhibits specific behavior only when an output parameter is null, we cannot remove it. For instance, the following function prints `"null!"` when `x` is null:

```
fn f(x: *mut i32) {
    if !x.is_null() {
        *x = 1;
    } else {
        println!("null!");
    }
}
```

If we remove `x`, the function cannot determine when to print `"null!"`.

Stored Pointers We cannot remove an output parameter stored in a memory location accessible even after the function returns. Such locations include pointers provided as arguments, return values, and global variables. Consider the following function that stores `x` to another parameter `y`:

```
fn f(x: *mut i32, y: *mut *mut i32) {
    *x = 1;
    *y = x;
}
```

If we remove `x`, the function cannot correctly update `y`.

To rule out such cases, we define *unremovable and removable parameters*:

Definition 4.11 (Unremovable/removable parameter)**.** A parameter x is *unremovable* if any of the following condition holds: (1) x points to an array; (2) the type of x is `*mut c_void`; (3) there is a program point reachable only when x is null; (4) x is stored in a memory location accessible after the function returns. A parameter x is *removable* if it is not unremovable.

4.2 Static Analysis

In this section, we propose a static analysis for identifying output parameters based on the abstract interpretation framework [73]. The analysis aims to find removable output parameters for each function and categorize them as either *must* or *may*. For each may-output parameter, it computes two kinds of information: (1) possible return values when the parameter is written and when not written; (2) program points where the parameter is fully written. The subsequent transformation phase utilizes this information.

Our analysis design prioritizes scalability, relying on two key decisions. First, it adopts a bottom-up approach, ensuring each function undergoes analysis only once, except for recursive ones. Second, the analysis intentionally sacrifices soundness to avoid the expensive overhead of being sound. We discuss the potential sources of unsoundness throughout this section.

At the same time, our goal is to achieve high precision: minimizing false positives, i.e., non-output parameters being misclassified as output parameters. Due to unsoundness, both false positives and false negatives are possible. A false positive occurs when the analysis fails to account for a read in a concrete execution, while a false negative arises when a write is not encompassed. Among these, false positives are more concerning, as they lead the subsequent transformation to change the semantics of the target program. False negatives merely result in fewer output parameters being removed. To enhance

$$
\begin{array}{rrcll}
\text{Label} & l & \in & \mathbb{L} & \\
\text{Variable} & x & \in & \mathbb{X} & \\
\text{Abstract pointer} & p^\# & \in & \mathbb{P}^\# & \\
 & p^\# & ::= & x \mid \mathsf{arg}(x) & \\
\text{Abstract address} & a^\# & \in & \mathbb{A}^\# & = \quad \mathcal{P}(\mathbb{P}^\#) \\
\text{Abstract integer} & z^\# & \in & \mathbb{Z}^\# & = \quad \{Z \in \mathcal{P}(\mathbb{Z}) : |Z| < N\} \cup \{\mathbb{Z}\} \\
\text{Abstract value} & v^\# & \in & \mathbb{V}^\# & = \quad \mathbb{A}^\# \times \mathbb{Z}^\# \\
\text{Abstract memory} & m^\# & \in & \mathbb{M}^\# & = \quad \mathbb{P}^\# \to \mathbb{V}^\# \\
\text{Abstract read/write/exclude set} & r^\#, w^\#, e^\# & \in & \mathbb{R}^\#, \mathbb{W}^\#, \mathbb{E}^\# & = \quad \mathcal{P}(\mathbb{X}) \\
\text{Abstract state} & s^\# & \in & \mathbb{S}^\# & = \quad \mathbb{L} \to (\mathbb{M}^\# \times \mathbb{R}^\# \times \mathbb{W}^\# \times \mathbb{E}^\#)
\end{array}
$$

Figure 4.2: Abstract domains

precision, the analysis generally overapproximates possible concrete behavior, except for determining which pointers are written. Since considering non-written pointers as written would generate a false positive, we underapproximate the set of written pointers.

We now provide a detailed explanation of the analysis. We first introduce a bottom-up static analysis with *abstract read/write/exclude sets* (Section 4.2.1). The read/write sets aid in identifying must-output parameters, while exclude sets identify unremovable parameters, particularly those associated with array pointers and stored pointers. We then extend the analysis by incorporating *write set sensitivity*, enabling the detection of may-output parameters and the computation of possible return values (Section 4.2.2). Finally, we further extend the analysis with *nullity sensitivity* (Section 4.2.3). This prevents must-output parameters from being misclassified as may-output in the presence of nullity checking and identifies unremovable parameters due to null-specific behavior. Note that we can syntactically identify unremovable parameters due to void pointers without relying on the analysis.

4.2.1 Abstract Read/Write/Exclude Sets

We begin the analysis by constructing a call graph of the entire program. This allows us to analyze the leaf nodes of the graph first and then analyze their callers using the results obtained from the callees. As call graph construction is not our primary focus, we syntactically build the call graph, ignoring function pointers. Existing control flow analysis techniques [182, 149] can be used to create a sound call graph and improve the analysis precision.

To maintain focus on the core concepts, we illustrate our analysis using a simplified language where values are limited to integers and pointers. We briefly discuss the handling of structs later in this section. Figure 4.2 defines the abstract domains used in the analysis. $\mathbb{L}$ is the set of labels (program points). $\mathbb{X}$ is the set of variable names, including parameters. $\mathbb{P}^\#$ is the set of abstract pointers, each is either a variable address or a pointer given as an argument. The notation $\mathsf{arg}(x)$ denotes a symbolic pointer given to the parameter x. $\mathbb{A}^\#$ is the set of abstract addresses, each is a set of abstract pointers. $\mathbb{Z}^\#$ is the set of abstract integers, each is a set of fewer than N integers or the top element $\mathbb{Z}$. By setting a finite value for N, the lattice is ensured to have only finite chains. In our implementation, N is 12. Our implementation employs widening for faster convergence, ensuring termination even when $N = \infty$. $\mathbb{V}^\#$ is the set of abstract values, each is a pair of an abstract address and an abstract integer. We use pairs instead of disjoint unions to precisely analyze C code that casts pointers to integers and then back

to pointers. $\mathbb{M}^\#$ is the set of abstract memories, adhering to the conventional definition. $\mathbb{R}^\#$, $\mathbb{W}^\#$, and $\mathbb{E}^\#$ are the set of abstract read/write/exclude sets, respectively. Each abstract read/write/exclude set is a set of variables. For read/exclude sets, we perform overapproximation, where the bottom element is empty, and the join operation is union. For write sets, we perform underapproximation, where the top is empty, and the join is intersection. Finally, $\mathbb{S}^\#$ is the set of abstract states, with each state mapping a label to a product of an abstract memory and abstract read/write/exclude sets. For ease of notation, we use $m_l^\#$, $r_l^\#$, $w_l^\#$, and $e_l^\#$ to denote the elements of $s^\#(l)$, where $s^\#$ is the computed abstract state of a function. The read/write/exclude sets in abstract states can be interpreted as follows:

- $x \in r_l^\#$ implies that x *may* be effectively read before reaching l.
- $x \in w_l^\#$ implies that x *must* be effectively written before reaching l.
- $x \in e_l^\#$ implies that x may be unremovable.

We now demonstrate how abstract states evolve during the analysis with concrete examples. We begin by focusing on read/write sets and then proceed to exclude sets. Subsequently, we discuss function calls and finally extend the analysis to support structs. In our examples, we use line numbers in labels: labels (n, in) and (n, out) denote the program points right before and after the statement in line n, respectively.

Read/Write Sets Consider the following example, where `f` effectively writes to `x`:

```
1 fn f(x: *mut i32, y: i32) {
2     *x = y;
3     let v = *x;
4 }
```

- Line 1: At the beginning, the read/write sets are empty. Given our bottom-up analysis approach, we lack information about the function's arguments, except for their types. Consequently, we initialize each integer parameter to the top, and each pointer parameter, denoted as x, to $\mathsf{arg}(x)$, which maps to the top. Thus, the abstract state is as follows:

$$m^\#_{1,\mathsf{out}} = [\mathtt{x} \mapsto \mathsf{arg}(\mathtt{x}), \mathtt{y} \mapsto \top, \mathsf{arg}(\mathtt{x}) \mapsto \top],\ r^\#_{1,\mathsf{out}} = \emptyset,\ w^\#_{1,\mathsf{out}} = \emptyset$$

  During the analysis, we assume that $\mathsf{arg}(x_1)$ and $\mathsf{arg}(x_2)$ are not aliased, where x_1 and x_2 are different variables. This assumption is unsound because aliasing can occur in concrete executions. However, this assumption allows us to obtain meaningful analysis results, preventing a write to a single parameter from being interpreted as affecting all parameters.

- Line 2: To analyze `*`E_1 `=` E_2, we compute the abstract value of E_1, which is a pair of an abstract address and an abstract integer, and obtain its abstract address element. If the abstract address is $\{\mathsf{arg}(x)\}$, we add x to the write set only if x is not already in the read set. When the abstract address contains multiple pointers, we refrain from updating the write set, as we cannot say a write must happen to any of those pointers. In the case of the current example, only one pointer belongs the abstract address, and the read set is empty, so the write set is updated:

$$m^\#_{2,\mathsf{out}} = [\mathtt{x} \mapsto \mathsf{arg}(\mathtt{x}), \mathtt{y} \mapsto \top, \mathsf{arg}(\mathtt{x}) \mapsto \top],\ r^\#_{2,\mathsf{out}} = \emptyset,\ w^\#_{2,\mathsf{out}} = \{\mathtt{x}\}$$

- Line 3: To analyze `let` x `= *`E, we compute the abstract address of E and add every x that $\mathsf{arg}(x)$ is in the address to the read set only if x is not already in the write set. In this particular example, `x` is already in the write set, so the read set remains unchanged:

$$m^{\#}_{3,\mathsf{out}} = [\mathtt{x} \mapsto \mathsf{arg}(\mathtt{x}), \mathtt{y} \mapsto \top, \mathtt{v} \mapsto \top, \mathsf{arg}(\mathtt{x}) \mapsto \top],\ r^{\#}_{3,\mathsf{out}} = \emptyset,\ w^{\#}_{3,\mathsf{out}} = \{\mathtt{x}\}$$

  Then, the analysis of the function terminates, and we can conclude that `x` is an output parameter because it belongs to the write set upon the function's return.

Exclude Sets Consider the following example, where `y` and `z` are unremovable parameters:

```
1 fn f(x: *mut *mut i32, y: *mut i32, z: *mut i32) {
2     *x = y;
3     *z.offset(1) = 0;
4     ...
5 }
```

- Line 1: At the beginning, the exclude set is empty.

$$m^{\#}_{1,\mathsf{out}} = [\mathtt{x} \mapsto \mathsf{arg}(\mathtt{x}), \mathtt{y} \mapsto \mathsf{arg}(\mathtt{y}), \mathtt{z} \mapsto \mathsf{arg}(\mathtt{z}), \ldots],\ e^{\#}_{1,\mathsf{out}} = \emptyset$$

- Line 2: When analyzing `*`E_1 `=` E_2, let $a^{\#}_1$ and $a^{\#}_2$ be the abstract addresses of E_1 and E_2, respectively. If $a^{\#}_1$ contains $\mathsf{arg}(x_1)$ for some x_1, we add every x_2 that $\mathsf{arg}(x_2)$ belongs to $a^{\#}_2$ to the exclude set. It is because pointers being stored in a memory location accessible after the function returns are unremovable. In the current code, the abstract addresses of `x` and `y` are $\{\mathsf{arg}(\mathtt{x})\}$ and $\{\mathsf{arg}(\mathtt{y})\}$, respectively. As a result, `y` is added to the exclude set:

$$m^{\#}_{2,\mathsf{out}} = [\mathtt{x} \mapsto \mathsf{arg}(\mathtt{x}), \mathtt{y} \mapsto \mathsf{arg}(\mathtt{y}), \mathtt{z} \mapsto \mathsf{arg}(\mathtt{z}), \ldots],\ e^{\#}_{2,\mathsf{out}} = \{\mathtt{y}\}$$

- Line 3: If a statement contains E_1`.offset(`E_2`)`, we add every x that $\mathsf{arg}(x)$ is in the abstract address of E_1 to the exclude set. Since C2Rust translates C's array indexing operation to the `offset` method invocation in Rust, such an expression indicates x being an array pointer and thus unremovable. In the current example, the abstract address of `z` is $\{\mathsf{arg}(\mathtt{z})\}$, so `z` is added to the exclude set:

$$m^{\#}_{3,\mathsf{out}} = [\mathtt{x} \mapsto \mathsf{arg}(\mathtt{x}), \mathtt{y} \mapsto \mathsf{arg}(\mathtt{y}), \mathtt{z} \mapsto \mathsf{arg}(\mathtt{z}), \ldots],\ e^{\#}_{3,\mathsf{out}} = \{\mathtt{y}, \mathtt{z}\}$$

  Since `y` and `z` belong to the exclude set, they are considered unremovable parameters and will not be removed by the transformation even when they are in the write set.

Function Calls As we analyze a callee before its caller, we use the analysis result of the callee when analyzing the caller. Consider the following example, where `f` calls `g`:

```
1 fn f(x: *mut i32, y: *mut i32, z: *mut i32) {
2     g(x, y, z);
3 }
4 fn g(a: *mut i32, b: *mut i32, c: *mut i32) {
5     ...
6 }
```

Suppose we have the following analysis result for `g` upon its return:

$$r^{\#}_{\mathsf{return}} = \{\mathtt{a}\},\ w^{\#}_{\mathsf{return}} = \{\mathtt{b}\},\ e^{\#}_{\mathsf{return}} = \{\mathtt{c}\}$$

It means that the first argument may be effectively read, the second argument must be effectively written, and the third argument may be unremovable. Since `f` passes `x`, `y`, and `z`, whose abstract addresses are $\{\mathsf{arg}(\mathtt{x})\}$, $\{\mathsf{arg}(\mathtt{y})\}$, and $\{\mathsf{arg}(\mathtt{z})\}$, the call to `g` updates the abstract state as follows:

$$r^{\#}_{2,\mathsf{out}} = \{\mathtt{x}\},\ w^{\#}_{2,\mathsf{out}} = \{\mathtt{y}\},\ e^{\#}_{2,\mathsf{out}} = \{\mathtt{z}\}$$

When the abstract address of an argument contains multiple pointers, the write set is not updated to maintain underapproximation, but the read/exclude sets are updated as they are overapproximated.

A challenge arises when dealing with a recursive function as its analysis result is unavailable when analyzing the function itself. To address this issue, we employ an iterative analysis approach for recursive functions. In the initial iteration, we analyze the function under the assumption that it never returns. In subsequent iterations, we analyze the function using the results obtained in the previous iterations. This process continues until the analysis results converge. Consider the following example, where `f` is a recursive function:

```
1 fn f(x: *mut i32, y: u32) {
2     if y == 0 {
3         *x = 1;
4     } else { f(x, y - 1); }
5 }
```

In the first iteration, line 5 is reachable only from line 3 because we assume that `f` does not return. As a result, the analysis result is as follows:

$$w^{\#}_{5,\mathsf{out}} = w^{\#}_{5,\mathsf{in}} = w^{\#}_{3,\mathsf{out}} = \{\mathtt{x}\}$$

In the second iteration, we know that `f` returns in line 4 using the previous result:

$$w^{\#}_{4,\mathsf{out}} = \{\mathtt{x}\}$$

Since line 5 is now reachable from both lines 3 and 4, we must join the two write sets:

$$w^{\#}_{5,\mathsf{out}} = w^{\#}_{5,\mathsf{in}} = w^{\#}_{3,\mathsf{out}} \sqcup w^{\#}_{4,\mathsf{out}} = \{\mathtt{x}\}$$

One more iteration is sufficient to achieve convergence in the analysis results, and we can conclude that `x` is an output parameter. When dealing with mutually recursive functions, we initially assume that all functions in the mutual recursion cycle never return and then iteratively analyze them until the analysis results for all functions converge.

Structs Extending the analysis to support structs is straightforward. We need to modify abstract read/write/exclude sets to be sets of maximal paths, not just variables. If $\mathsf{arg}(x_1), \ldots, \mathsf{arg}(x_n)$ belong to the abstract address of E, the expression $(*E).l_1.\cdots.l_m$ adds each maximal path whose prefix is one of $x_1.l_1.\cdots.l_m, \ldots, x_n.l_1.\cdots.l_m$ to the read set, except those already in the write set. Similarly, if $\{\mathsf{arg}(x)\}$ is the abstract address of E_1, the expression $(*E_1).l_1.\cdots.l_m = E_2$ adds maximal paths starting with $x.l_1.\cdots.l_m$ to the write set, except those already in the read set.

4.2.2 Write Set Sensitivity

The analysis in Section 4.2.1 can detect must-output parameters but not may-output parameters. Consider the following example, where `x` is a may-output parameter:

```
1 fn f(x: *mut i32, c: bool) -> i32 {
2     let mut v = 0;
3     if c {
4         *x = 1;
5     } else { v = 1; }
6     v
7 }
```

Listing 4.1: May output parameter

When `c` is true, `f` writes to `x` and returns `0`; otherwise, it does not write to `x` and returns `1`. However, analyzing this function cannot figure out that `x` is an output parameter because the write set is underapproximated. Since line 4 writes to `x`, we get the following state:

$$m_{4,\mathsf{out}} = [\mathtt{v} \mapsto \{0\}, \ldots],\ w^{\#}_{4,\mathsf{out}} = \{\mathtt{x}\}$$

On the other hand, line 5 does not write to `x`, and we get the following result:

$$m_{5,\mathsf{out}} = [\mathtt{v} \mapsto \{1\}, \ldots],\ w^{\#}_{5,\mathsf{out}} = \emptyset$$

Since the join for write sets is intersection, `x` does not belong to the write set of line 6 and is not considered an output parameter:

$$m_{6,\mathsf{out}} = m_{6,\mathsf{in}} = m_{4,\mathsf{out}} \sqcup m_{5,\mathsf{out}} = [\mathtt{v} \mapsto \{0,1\}, \ldots],\ w^{\#}_{6,\mathsf{out}} = w^{\#}_{6,\mathsf{in}} = w_{4,\mathsf{out}} \sqcup w_{5,\mathsf{out}} = \emptyset$$

To address this limitation, we introduce write set sensitivity, which distinguishes different write sets. We change the definition of an abstract state to map a product of a label and an abstract write set to a product of an abstract memory and abstract read/exclude sets:

$$\mathbb{S}^{\#} = (\mathbb{L} \times \mathbb{W}^{\#}) \to (\mathbb{M}^{\#} \times \mathbb{R}^{\#} \times \mathbb{E}^{\#})$$

Lines 4 and 5 are analyzed the same as before:

$$m^{\#}_{4,\mathsf{out},\{\mathtt{x}\}} = [\mathtt{v} \mapsto \{0\}, \ldots],\ m^{\#}_{5,\mathsf{out},\emptyset} = [\mathtt{v} \mapsto \{1\}, \ldots]$$

However, in line 6, the write sets are not joined because they are different.

$$\begin{aligned} m^{\#}_{6,\mathsf{out},\{\mathtt{x}\}} &= m^{\#}_{6,\mathsf{in},\{\mathtt{x}\}} = m^{\#}_{4,\mathsf{out},\{\mathtt{x}\}} = [\mathtt{v} \mapsto \{0\}, \ldots] \\ m^{\#}_{6,\mathsf{out},\emptyset} &= m^{\#}_{6,\mathsf{in},\emptyset} = m^{\#}_{5,\mathsf{out},\emptyset} = [\mathtt{v} \mapsto \{1\}, \ldots] \end{aligned}$$

From the analysis result, we can conclude that `x` is effectively written in some executions but not in others and thus is a may-output parameter. Furthermore, we naturally obtain the possible return values for each case: $\{0\}$ when `x` is written and $\{1\}$ otherwise. We also need to find program points where `x` is fully written. For this purpose, we examine each statement that writes to `x`, where the write set before the statement lacks `x`, and the write set after the statement includes `x`. Given these criteria, we determine that `x` is fully written in line 4.

Unfortunately, write set sensitivity can be extremely costly, especially when structs and loops are involved. Consider the following code, where C2Rust translates C's `switch-case` to Rust's `match`:

```
struct S { a0: i32, ..., a9: i32 }
fn f(x: *mut S) {
    ...
    while c {
        match v {
            0 => {
                (*x).a0 = 0;
            }
            ...
            9 => {
                (*x).a9 = 0;
            }
            _ => {}
        }
        ...
    }
}
```

The struct `S` has ten fields, from `a0` to `a9`, and `f` has a loop that writes to one of the fields in each iteration, depending on the value of `v`. When analyzing `f` with write set sensitivity, 1,024 ($= 2^{10}$) different write sets occur because executions can write to different combinations of the ten fields. It means that every program point in the loop is analyzed under 1,024 different contexts, consuming a significant amount of time. Adopting widening for write sets is not a solution to this problem because write sets are not even joined due to the sensitive nature of the analysis.

One solution is to insensitively analyze loops: write sets are joined at every program point in a loop. However, this approach prevents the analysis from detecting output parameters within loops that can be sensitively analyzed without a significant computational cost. The following function serves as an example:

```
fn f(x: *mut i32) {
    ...
    while c {
        if d {
            *x = 0;
        }
        ...
    }
}
```

If we analyze `f` sensitively, `x` is classified as a may-output parameter, and the analysis can terminate within a reasonable amount of time because only two distinct write sets occur. However, when analyzed insensitively, `x` is not considered an output parameter.

Therefore, a more preferable solution is to sensitively analyze only non-costly loops and insensitively analyze the rest. The problem is that we cannot know whether a loop is costly or not before analyzing it. To address this issue, we adopt a try-and-restart approach: we take a hyperparameter *max sensitivity* (M) from the user and initiate the analysis with every loop being analyzed sensitively. If a program point within a certain loop exhibits more than M different write sets, we restart the analysis with the loop being analyzed insensitively. When $M = 1$, every loop is analyzed insensitively, and when $M = \infty$,

every loop is analyzed sensitively.

In fact, the current use of write set sensitivity is unsound. The issue lies in the fact that different write sets do not represent disjoint sets of concrete states. Since the write set is underapproximated, $x \notin w^{\#}$ is interpreted as x being either effectively written or not, but not as x being guaranteed not to be effectively written. Consequently, two different write sets can encompass the same concrete state. For example, both $w^{\#} = \{\mathtt{x}\}$ and $w^{\#} = \{\mathtt{y}\}$ can approximate states where both `x` and `y` are effectively written.

Two approaches exist to make the use of write set sensitivity sound. First, we can properly join the states associated with different write sets [121]. For example, in Listing 4.1, the line 6 can be analyzed as follows:

$$m^{\#}_{6,\mathsf{out},\{\mathtt{x}\}} = m^{\#}_{6,\mathsf{in},\{\mathtt{x}\}} = m^{\#}_{4,\mathsf{out},\{\mathtt{x}\}} \sqcup m^{\#}_{5,\mathsf{out},\emptyset} = [\mathtt{v} \mapsto \{0,1\}, \ldots]$$
$$m^{\#}_{6,\mathsf{out},\emptyset} = m^{\#}_{6,\mathsf{in},\emptyset} = m^{\#}_{4,\mathsf{out},\{\mathtt{x}\}} \sqcup m^{\#}_{5,\mathsf{out},\emptyset} = [\mathtt{v} \mapsto \{0,1\}, \ldots]$$

We can still detect that `x` is an output parameter, but the return values are now imprecise: $\{0,1\}$ are possible both when `x` is written and when not written. Since the subsequent transformation relies on the precise identification of return values, we decided to sacrifice soundness.

Another option is to make each write set track the exact set of written pointers. Currently, underapproximation occurs due to three reasons. First, we overapproximate read sets, and a parameter can be added to the write set only when it is not in the read set. Thus, we have to make the read set exact as well. This involves introducing read set sensitivity. Second, when an abstract address contains multiple pointers, a read from it makes the read set imprecise, and a write to it makes the write set imprecise. To resolve this, we must create distinct read/write sets for different pointers in the abstract address. Finally, insensitively analyzing loops makes the read/write sets joined in loops. Preventing this requires analyzing all loops sensitively. However, these changes would significantly degrade the scalability of the analysis, and we decided to sacrifice soundness.

4.2.3 Nullity Sensitivity

We now focus on the analysis of code with nullity checking, specifically addressing the identification of unremovable parameters without misclassifying must-output parameters as may-output parameters. Consider the following code snippets:

```
1 fn f(x: *mut i32) {
2     if !x.is_null() {
3         *x = 1; }
4 }
5
```

Listing 4.2: Must-output parameter

```
1 fn f(x: *mut i32) {
2     if !x.is_null() {
3         *x = 1;
4     } else { println!("null!"); }
5 }
```

Listing 4.3: Null-specific behavior

A precise analysis should conclude that `x` in Listing 4.2 is a must-output parameter, and `x` in Listing 4.3 is an unremovable parameter.

If we interpret $\mathsf{arg}(x)$ as a non-null pointer, we get an incorrect result. When analyzing Listing 4.3, the condition `!x.is_null()` evaluates to true, causing line 2 to be deemed unreachable. Consequently, we cannot detect the null-specific behavior of `f`, thereby misclassifying `x` as a removable parameter.

On the other hand, considering $\mathsf{arg}(x)$ as a nullable pointer also fails to produce a correct result. In this case, `!x.is_null()` can be both true and false. Therefore, in Listing 4.2, line 4 can be reached from

both lines 2 and 3, resulting in the following analysis result:

$$m^{\#}_{4,\mathsf{out},\emptyset} = m^{\#}_{4,\mathsf{in},\emptyset} = m^{\#}_{2,\mathsf{out},\emptyset} = [\ldots],\ m^{\#}_{4,\mathsf{out},\{\mathtt{x}\}} = m^{\#}_{4,\mathsf{in},\{\mathtt{x}\}} = m^{\#}_{3,\mathsf{out},\{\mathtt{x}\}} = [\ldots]$$

From the result, we can deduce only that `x` may not be effectively written but not that it is not written only when it is null. This leads to a misclassification of `x` as a may-output parameter.

To enhance the precision of the analysis, we introduce nullity sensitivity, which distinguishes between $\mathsf{arg}(x)$ being null and non-null. $\mathbb{U}^{\#}$ represents the set of abstract null sets, each of which is a set of variables, and the definition of abstract states is revised accordingly:

$$\text{Abstract null set} \quad u^{\#} \in \mathbb{U}^{\#} = \mathcal{P}(\mathbb{X}) \qquad\qquad \mathbb{S}^{\#} = (\mathbb{L} \times \mathbb{W}^{\#} \times \mathbb{U}^{\#}) \to (\mathbb{M}^{\#} \times \mathbb{R}^{\#} \times \mathbb{E}^{\#})$$

Similar to write sets, we underapproximate null sets, resulting in the empty set for the top element and intersection for the join operation.

When analyzing E`.is_null()` with nullity sensitivity, if the abstract address of E is $\mathsf{arg}(x)$, we create two distinct null sets: one maintains the current null set, and the other extends the current set with x. In the state associated with the former, E`.is_null()` evaluates to false, while in the state associated with the latter, it evaluates to true. Using nullity sensitivity, we obtain the following analysis result for Listing 4.2:

$$m^{\#}_{4,\mathsf{out},\emptyset,\{\mathtt{x}\}} = m^{\#}_{4,\mathsf{in},\emptyset,\{\mathtt{x}\}} = m^{\#}_{2,\mathsf{out},\emptyset,\{\mathtt{x}\}} = [\ldots],\ m^{\#}_{4,\mathsf{out},\{\mathtt{x}\},\emptyset} = m^{\#}_{4,\mathsf{in},\{\mathtt{x}\},\emptyset} = m^{\#}_{3,\mathsf{out},\{\mathtt{x}\},\emptyset} = [\ldots]$$

Since `x` is effectively written when it is non-null, we can correctly classify `x` as a must-output parameter. In Listing 4.3, line 4 is reachable with the following state:

$$m^{\#}_{4,\mathsf{out},\emptyset,\{\mathtt{x}\}} = [\ldots]$$

This implies that the function has a program point that is reachable only when `x` is null, allowing us to correctly classify `x` as an unremovable parameter.

Since null sets are underapproximated, nullity sensitivity is unsound, like write set sensitivity. The same solutions can ensure soundness, but we prioritize precision and scalability over soundness.

4.3 Code Transformation

In this section, we describe the transformation of Rust code generated by C2Rust based on the analysis result. Note that the analysis result cannot fully capture the programmers' intentions. Even when an output parameter can be removed without altering the program's behavior, developers may prefer to retain it. We suggest that a practical solution would be receiving a list of parameters from the programmers that they wish to retain.

We demonstrate the code transformation for must-output parameters (Section 4.3.1) and may-output parameters (Section 4.3.2). For clarity, we provide illustrative examples comprising code snippets before and after the transformation, accompanied by a line-by-line explanation of the modifications. The transformation aims to: (1) introduce Rust-idiomatic function signatures by replacing output parameters with return types comprising tuples and `Option`/`Result` types; and (2) modify the function bodies to align with these new signatures. The new signatures offer programmers more precise information about the functions' behavior, compared to the original ones, thereby improving code comprehension. However, modifications in the function bodies, such as the introduction of additional local variables and the

complexification of data flow, may negatively impact readability. We believe further transformation simplifying function bodies could enhance readability and briefly discuss potential simplification strategies after demonstrating our code transformation.

4.3.1 Must-Output Parameters

Consider the following definitions of function `f`: on the left is the original code before the transformation, with `x` as a must-output parameter, and on the right is the code after the transformation.

```
1  fn f(x: *mut i32) -> i32 {
2
3
4      *x = 2;
5      return 1;
6  }
7  let y = f(x);
8
9
10
11
```

```
fn f() -> (i32, i32) {
    let xv = 0;
    let x: *mut i32 = &mut xv;
    *x = 2;
    return (1, xv);
}
let (v0, v1) = f();
if !x.is_null() {
    *x = v1;
}
let y = v0;
```

- Line 1: We remove the output parameter `x` of type `*mut i32` and add `i32` to the return type by changing the return type to `(i32, i32)`.
- Lines 2–3: We introduce a new variable `xv` of type `i32` with an arbitrary initial value. This is where a value originally stored in the output parameter is stored. Additionally, we redefine `x` as a pointer to `xv`, ensuring expressions referencing `x` behave correctly without any modification.
- Line 5: For each `return`, we construct a tuple to return both the original return value and `xv`.
- Lines 7–11: We remove the argument `x` and destructure the tuple returned by `f` into `v0` and `v1`. If `x` is non-null, we assign `v1` to `x`. Then, `v0` is used in place of the original return value.

We can simplify the function body by removing local variables to enhance readability. If `x` is solely utilized for indirect assignments and dereferences without being passed to functions or assigned to other variables, it can be removed by replacing `*x` with `xv`. Moreover, if `xv` is written to immediately before a return, `xv` can be removed as well, directly returning the written value. These simplifications can be easily achieved through syntactic transformation.

4.3.2 May-Output Parameters

We employ two methods for removing a may-output parameter. If we cannot identify return values that indicate success (the function has written to the parameter) and failure (the function has not written to the parameter), we retain the original return value and additionally return the value of the output parameter as a tuple. Conversely, if such return values can be identified, we remove the original return value and return only the value of the output parameter.

To identify return values indicating success and failure, we rely on the analysis result. The analysis computes return values when the output parameter is written. If there are multiple such values, they not only signify success but also provide additional information to callers through their values. In such cases,

we must retain the original return value because removing it prevents callers from obtaining necessary information beyond mere success or failure. On the other hand, if a unique value v exists, we compare it with the return values when the output parameter is not written, which are also computed by the analysis. If v is found among these values, we cannot consider it an indicator of success. Conversely, if v does not appear among these values, we conclude that v indeed signifies success.

To facilitate successful identification of return values, we employ the set domain for integers instead of the interval domain during the analysis. For instance, if a function returns 0 for success and -1 or 1 for failure, the set domain enables us to conclude that 0 is not in $\{-1, 1\}$. However, when using the interval domain, 0 belongs to the interval $[-1, 1]$, preventing us from recognizing 0 as an indicator of success.

Retaining Original Return Values Consider the following definitions of function `f`. Before the transformation, `x` is a may-output parameter, and the return value is always `1`:

```
1  fn f(x: *mut i32, c: bool) -> i32 {
2
3
4
5      if c {
6          *x = 2;
7
8      }
9      return 1;
10
11 }
12 let y = f(x, c);
13
14
15
16
17
18
```

```
fn f(c: bool) -> (i32, Option<i32>) {
    let xv = 0;
    let x: *mut i32 = &mut xv;
    let xw: bool = false;
    if c {
        *x = 2;
        xw = true;
    }
    return (1,
        if xw { Some(xv) } else { None });
}
let (v0, v1) = f(c);
if let Some(v) = v1 {
    if !x.is_null() {
        *x = v;
    }
}
let y = v0;
```

- Line 1: We remove the output parameter `x` and change the return type to `(i32, Option<i32>)`.
- Lines 2–3: This step aligns with Section 4.3.1.
- Line 4: We introduce a new variable `xw` to track whether `x` is fully written.
- Line 7: According to the analysis, `x` becomes fully written on this line. Thus, we set `xw` to `true`.
- Lines 9–10: In addition to the original return value, we return `Some(xv)` if `x` is fully written, or `None` otherwise.
- Lines 12–18: If `v1` is `Some(v)`, we assign `v` to `x` using an `if-let` expression [11]. Other changes are the same as Section 4.3.1.

Note that we can also use `Result<`T`, ()>` instead of `Option<`T`>` as the return type, where `()` is the unit type; this choice is just a matter of preference.

To improve readability, we can simplify return statements, currently depending on `xw` to select between `Some(xv)` and `None`. If the analysis result indicates that `x` must have been written to before reaching a specific return statement, the statement can be modified to directly return `Some(xv)` without checking `xw`. Similarly, if `x` has not been written to, we can return `None`. Furthermore, if this simplification makes `xw` never read, we can remove its definition and assignments. Note that the proposed analysis already suffices for providing information required by these simplifications.

Removing Original Return Values Consider the following definitions of function `f`. Before the transformation, `x` is a may-output parameter, and the return value is `0` for success and `1` for failure:

```
1  fn f(x: *mut i32, c: bool) -> i32 {
2
3
4
5      let y = 1;
6      if c {
7          y = 0;
8          *x = 2;
9
10     }
11     return y;
12 }
13 if f(c) == 0 {
14
15
16
17
18     ...
19 } else {
20
21     ...
22
23 }
```

```
fn f(c: bool) -> Result<i32, i32> {
    let xv = 0;
    let x: *mut i32 = &mut xv;
    let xw: bool = false;
    let y = 1;
    if c {
        y = 0;
        *x = 2;
        xw = true;
    }
    return if xw {Ok(xv)} else {Err(y)};
}
match f(c) {
    Ok(v) => {
        if !x.is_null() {
            *x = v;
        }
        ...
    }
    Err(_) => {
        ...
    }
}
```

- Line 1: We remove `x` and change the return type to `Result<i32, i32>`. The first `i32` corresponds to the type of the output parameter, and the second `i32` corresponds to the original return type.
- Lines 2–10: These modifications are the same as before.
- Line 11: When `x` is fully written, we return `Ok(xv)`. Otherwise, we return `Err(y)`, where `y` is the original return value, potentially containing information related to the failure.
- Lines 13–23: The original code's check for the return value and subsequent actions on success and failure are replaced with pattern matching.

The return value of `f` may not be immediately checked, as shown in the following example:

```
let y = f(x, c);
```

```
let y = match f(c) {
    Ok(v) => {
        if !x.is_null() {
            *x = v;
        }
        0
    }
    Err(v) => {
        v
    }
};
```

In this case, we reconstruct the original return value using pattern matching. On success, the value `0` is obtained from the analysis result; on failure, the value is extracted from `Err`. Note that if the value indicating failure is unique, we can use `Option<`T`>` instead of `Result<`T`, `E`>`.

4.4 Evaluation

In this section, we evaluate our approach using 55 real-world C programs. First, we describe our implementation of Nopcrat, which embodies our approach (Section 4.4.1), and the process of collecting the benchmark programs (Section 4.4.2). We then assess our approach by addressing the following research questions:

- RQ1. Scalability: Does it efficiently analyze and transform large programs? (Section 4.4.3)
- RQ2. Usefulness: How many output parameters can it identify, and to what extent is code modification required to remove them? (Section 4.4.4)
- RQ3. Correctness: Does it transform code while preserving its semantics? (Section 4.4.5)
- RQ4. Impact on performance: What is the effect of removing output parameters on program performance? (Section 4.4.6)

Our experiments were conducted on an Ubuntu machine with Intel Core i7-6700K (4 cores, 8 threads, 4GHz) and 32GB DRAM. Finally, we discuss potential threats to validity (Section 4.4.7).

4.4.1 Implementation

We implemented Nopcrat on top of the Rust compiler [12]. It analyzes Rust code after lowering it to Rust's mid-level intermediate representation (MIR) [15], which represents functions as control flow graphs with basic blocks. To transform the code, Nopcrat relies on Rust's high-level intermediate representation [14], similar to abstract syntax trees but with syntactic sugar desugared and symbols resolved. We used C2Rust v0.18.0 with minor modifications.

Additionally, we require post-processing of C2Rust's output before analyzing and transforming it. C2Rust translates each C file to a Rust file, declaring definitions in other files as extern definitions, with function calls resolved at link time. However, Rust provides the `use` keyword for importing definitions, which can undergo type checking at compile time. Therefore, to ensure that changes in function signatures during our transformation properly affect type checking, we need to replace the extern definitions with

`use` statements. While Emre et al. [84]'s ResolveImports tool performs such post-processing, it supports a previous version of C2Rust. Hence, we implemented a tool that performs the same task on the output of C2Rust v0.18.0.

Although our implementation utilizes C2Rust for syntactic translation, it does not preclude the use of other C-to-Rust translators. The implemented analyzer presumes that the input Rust code employs only C-equivalent features, excluding Rust-specific features such as safe references, traits, and generics. Any existing C-to-Rust translator that adheres to this constraint can substitute C2Rust. Conversely, code produced by tools that improve C2Rust's output with Rust features [84, 83, 203] is incompatible with our current implementation. However, this limitation is not intrinsic to our approach, as extending the implementation to support Rust features is feasible without significant challenges.

4.4.2 Benchmark Program Collection

We collected benchmark programs from two sources: (1) those used by CROWN [203] and (2) GNU packages. Since CROWN's benchmark set primarily comprises small programs ($<$ 5k LOC), we decided to augment it with large programs from GNU packages. Initially, from the packages listed in the GNU Package Blurbs [6], we gathered 41 packages, which are all of C programs satisfying the following conditions: (1) not exceeding 100k LOC, as measured by cloc [76]; (2) compiled successfully on our Ubuntu machine; and (3) considered well-known, determined by whether the package has an individual entry on Wikipedia. During their translation, C2Rust crashes in four packages (`adns`, `gmp`, `parted`, and `readline`). Additionally, in two packages (`bison` and `m4`), we cannot replace extern declarations with `use` due to conflicting type definitions with the same name. Excluding these, we derived a final set of 35 packages. By adding them to 20 programs from CROWN, we obtained a total of 55 benchmark programs. C2Rust occasionally produces Rust code with type errors, mostly caused by missing type casts, and we manually corrected them. Table 4.1 reports the sizes of the benchmark programs.

4.4.3 RQ1: Scalability

We evaluate the scalability of the proposed approach by measuring the time required to analyze and transform code. Our experimental results demonstrate that the proposed approach is scalable, as each benchmark program can be successfully analyzed and transformed in a maximum of 213 seconds. In this subsection, we provide a detailed description of the results.

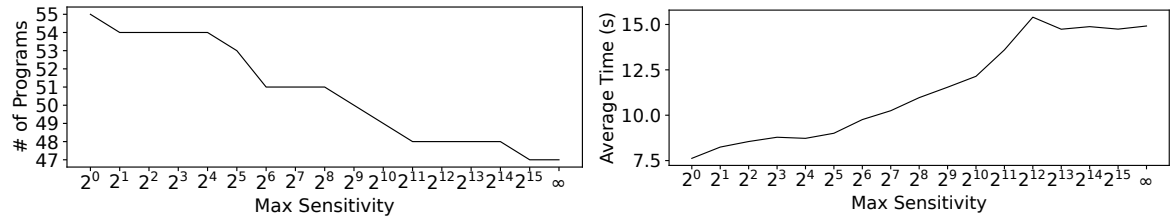
We first investigate how *max sensitivity* (M) impacts the successful analysis of complex loops. We use varying values of M ranging from 2^0 to 2^{15}, as well as ∞. M increases exponentially, doubling the value each time, as the number of possible write sets doubles when a write to another pointer is introduced within a loop. Figure 4.4a presents the number of successfully analyzed programs for each M. We conducted the experiments under a 32GB memory constraint; exceeding this results in analysis failure due to an out-of-memory error. With $M = 1$, we successfully analyze all 55 programs. However, increasing M causes more programs to fail analysis, leading to only 47 programs being analyzed when $M = \infty$. When analyzing loops with high values of M, many distinct write sets are constructed, consuming a significant amount of memory. We believe that existing techniques, e.g., sparse analysis [97, 164], can be applied to mitigate memory consumption and enable the analysis of more programs with high M values.

We now explore the impact of different values of M on analysis time. We restrict our discussion to the 47 programs that can be analyzed regardless of M. Figure 4.4b depicts the average analysis times for various M. When $M = 1$, the average analysis time is 7.6 seconds, and it increases as M grows, reaching

Table 4.1: Benchmark programs for evaluating Nopcrat

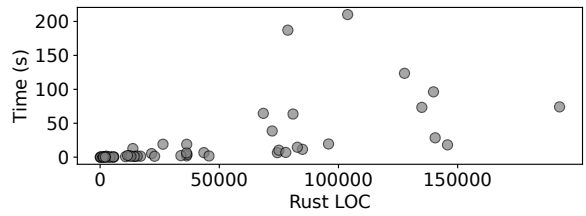
Program	C LOC	LOC	Fns	Blocks	Stmts
urlparser*	56	1360	21	585	1998
bst*	65	89	5	48	97
avl*	101	114	9	74	170
quadtree-0.1.0*	365	1057	31	378	1352
buffer-0.4.0*	395	1137	42	436	1715
rgba*	396	2128	19	345	1739
robotfindskitten*	398	1508	18	307	1283
genann-1.0.0*	608	1818	25	711	3621
ht*	680	264	10	120	336
libcsv*	965	3010	31	806	4145
libtree-3.1.1*	1412	2632	30	1193	4320
which-2.21	2010	2241	34	910	3332
libzahl-1.0*	2438	4096	108	2032	6207
ed-1.19	2439	5636	135	2815	9040
time-1.9	2828	1830	25	572	2314
lil*	2934	5558	136	2957	8739
libtool-2.4.7	3769	5701	107	2376	7200
json.h*	4490	3831	53	1827	6406
lodepng*	5098	14299	236	6854	24468
bzip2*	5316	13731	110	6357	28054
pexec-1.0rc8	5357	12301	156	5883	22002
binn-3.0*	5686	4298	165	1806	5719
heman*	7048	14690	302	5386	30907
units-2.22	7240	11521	142	4999	17348
pth-2.0.7	7590	12950	217	5481	19981
hello-2.12.1	8340	10688	174	2971	10217
bc-1.07.1	10810	16982	231	5165	20179
tulipindicators*	12371	22864	307	7133	28879
brotli-1.0.9*	13173	127691	867	18375	64394
libosip2-5.3.1	15772	36286	704	16401	53552
mcsim-6.2.0	18112	36454	502	15593	63847
mtools-4.0.43	18266	33895	664	12358	47869
indent-2.2.13	19255	15581	124	4532	22975
less-633	20063	45685	655	10596	38782
cflow-1.7	20601	26375	478	10523	36326
gzip-1.12	20875	21605	242	10917	43162
dap-3.10	22420	43549	355	25059	105696
patch-2.7.6	28215	103839	599	41304	247522
rcs-5.10.1	28286	36267	490	15979	57817
make-4.4.1	28911	36336	439	16777	59691
enscript-1.6.6	34868	78749	251	18815	81269
cpio-2.14	35934	80929	703	41054	151073
screen-4.9.0	39335	72199	691	30062	121018
nano-7.2	42999	74994	797	33598	129165
sed-4.9	48190	68465	669	32887	116429
uucp-1.07	51123	77872	791	28122	110130
gprolog-1.5.0	52193	74381	1709	30520	134299
gawk-5.2.2	58111	140566	1444	53592	209695
diffutils-3.10	59377	95835	841	47191	165136
nettle-3.9	61835	82742	1004	23832	136303
grep-3.11	64084	84902	861	40947	144016
tar-1.34	66172	134972	1665	62175	226080
glpk-5.0	71805	145738	1521	59762	236170
findutils-4.9.0	80015	139858	1220	66231	237408
wget-1.21.4	81188	192742	1353	73258	293725

(C LOC: lines of C code, LOC: lines of Rust code after C2Rust's translation, Fns: number of functions, Blocks/Stmts: number of basic blocks/statements in MIR, *: from Crown)

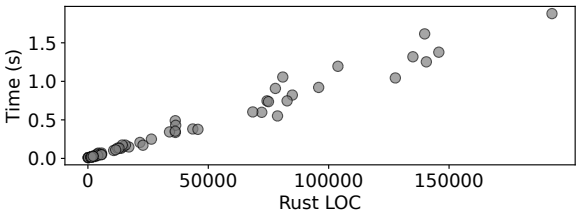


(a) Number of successfully analyzed programs

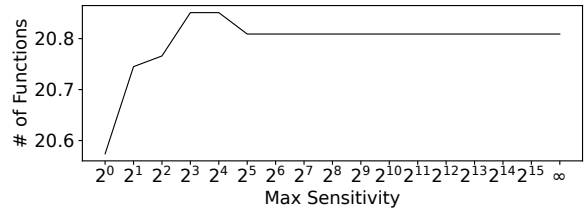
(b) Average analysis time



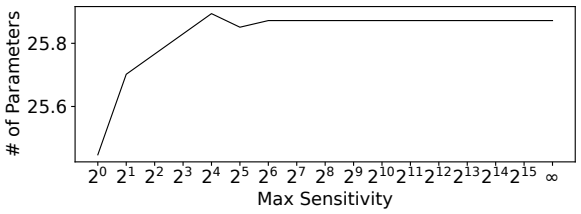
(c) Analysis time of each program



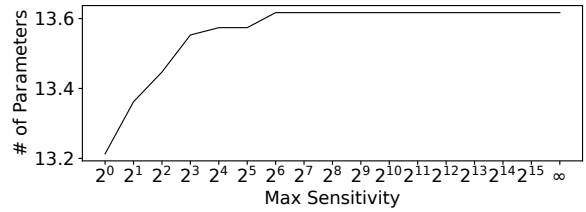
(d) Transformation time of each program



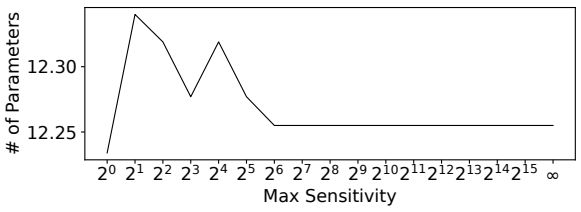
(e) Average number of functions containing output parameters



(f) Average number of output parameters



(g) Average number of must-output parameters



(h) Average number of may-output parameters

Figure 4.4: Experimental results of Nopcrat

14.9 seconds when $M = \infty$. Note that this analysis is performed only once during the whole translation process. Therefore, unless the analysis time is exceedingly long, prioritizing precision is recommended. Given the moderate increase in analysis time, choosing a higher value of M is advisable for practical purposes, as it allows higher precision.

We then investigate the impact of program size on analysis time. We set M to 1 and examine all 55 programs. Figure 4.4c shows the analysis time for each program, plotted against Rust LOC on the x-axis. Overall, the analysis time tends to increase with LOC, but the correlation is not strong. For instance, a program with 139k LOC (`glpk`) takes only 18 seconds, while one with 110k LOC (`patch`) takes 211 seconds. The reason is the bottom-up nature of our analysis, where each function is independently analyzed. The analysis of a function typically exhibits superlinear time complexity relative to its size. Therefore, analyzing a single complex function would take considerably more time than analyzing multiple simpler functions. For example, in `patch`, a single function `yyparse`, comprising 7,526 blocks and 46,878 statements, consumes 115 seconds out of the total 211 seconds.

Finally, we turn our attention to the transformation time. In this experiment, we use the analysis

result obtained with the highest possible value of M for each program. Figure 4.4d illustrates the transformation time for each program. Every program can be transformed within 2 seconds, and there is a linear relationship between the transformation time and LOC. This correlation arises because the transformation involves straightforward tree walking based on the analysis results.

4.4.4 RQ2: Usefulness

We evaluate the usefulness of the proposed approach, which aims to reduce the burden on programmers by automating the removal of output parameters. Our approach is considered useful if (1) it successfully identifies many output parameters in real-world code and (2) their removal requires substantial code modifications. To assess the usefulness, we measure the number of identified output parameters and the extent of code changes needed to remove them.

Our experimental results confirm the usefulness of the proposed approach. Using the highest possible value of M for each program, across all 55 programs, we identify an average of 30.4 output parameters, comprising 15.5 must-output parameters and 14.9 may-output parameters distributed over 24.4 distinct functions. Thirteen programs (`avl`, `bst`, `buffer`, `genann`, `ht`, `libcsv`, `libtree`, `libzahl`, `quadtree`, `robotfindskitten`, `time`, `tulipindicators`, and `urlparser`) have no output parameters, while others have at least one. We consider the average of 30.4 output parameters to be a significant indicator of the approach's usefulness. The removal of the identified parameters necessitates changes in 13.9 files, involving the insertion of 313.6 lines and the deletion of 285.3 lines. If these modifications are performed manually, it would entail a significant amount of effort.

Since the analysis may yield false negatives, we conducted manual inspections. We randomly selected one function with a parameter not identified as an output parameter from each program, totaling 55 functions. We manually examined the parameters to ascertain if they are indeed true negatives. We found one false negative in the following function in `tulipindicators`:

```
fn tc_config_set_to_default(mut config:nopcrat: *mut tc_config) {
    memcpy(config, tc_config_default(), size_of::<tc_config>());
}
```

The analyzer failed to recognize the call to `memcpy` as a write to `config` due to the absence of models for the C library functions. We can address this issue by modeling the functions. Although our investigation was not exhaustive, it suggests a low false negative rate of the proposed analysis.

We now investigate the influence of different values of M on identifying output parameters. Figure 4.4e, Figure 4.4f, Figure 4.4g, and Figure 4.4h present the average numbers of output-parameter-containing functions, output parameters, must-output parameters, and may-output parameters across the 47 programs, respectively, for each value of M. Increasing M enhances the precision of our analysis. Consequently, the number of identified output parameters tends to increase as M increases, but there are exceptions. This is due to that unremovable parameters undetected with low values of M become identified with higher values of M. Furthermore, as we increase M, the number of identified may-output parameters initially rises, but it subsequently declines. This decline occurs because some may-output parameters are reclassified as must-output parameters with higher values of M.

4.4.5 RQ3: Correctness

We evaluate the correctness of the proposed approach by checking whether the transformed program maintains the same semantics as the original program. If the analysis produces false positives, the

transformation phase would remove non-output parameters, altering the program's semantics. For the assessment, we use the test suite of each program, where 26 among the 55 programs possess test suites. Before the transformation, all the 26 programs pass their test suites, and after the transformation, 25 continue to pass their test suites. The only program that encounters failure is `tar`, due to a specific function shown below:

```
static mut current_format: archive_format;
fn decode_header(format_pointer: *mut archive_format, ...) {
    *format_pointer = format;
    ...
    if current_format == GNU_FORMAT {
        ...
    }
    ...
}
decode_header(&mut current_format, ...);
```

The analysis identifies `format_pointer` as a must-output parameter. This is not a false positive because the parameter is always effectively written by the function. The problem is that one caller provides a pointer to the global variable `current_format` as an argument, while `decode_header` reads `current_format` after writing to `format_pointer`. Consequently, the value read from `current_format` should match the value written to `format_pointer`. However, if we remove `format_pointer` and redefine it as a pointer to a local variable, writing to it no longer impacts the value of `current_format`, thus altering the function's semantics. We checked that manually excluding `format_pointer` from the identified output parameters makes the transformed program pass the test suite.

To address this issue, we need to extend the definition of unremovable parameters to encompass such patterns, necessitating a corresponding adjustment to our analysis. This extension would involve identifying pointers to global variables. While the current implementation of the analysis treats all global variables as a single abstract memory location, we can modify the analysis to distinguish different global variables.

Since our experiments revealed no false positives, we further conducted a manual investigation. We randomly selected a function where a may-/must-output parameter was identified, from each program containing such a function, totaling 41 functions with may-output parameters and 41 with must-output parameters. We then manually examined whether the identified parameters are indeed output parameters. While we confirmed that all 41 must-output parameters are true positives, we discovered that one of the 41 may-output parameters is, in fact, a must-output parameter. This parameter appears in the following function of `hello`:

```
1  void parse_options(..., char **m){
2      switch (optc) {
3          case 103:
4              *m = ...;
5              break;
6
7          case 128:
8              print_help();
9
10
11         case 116:
12             *m = ...;
13             break;
14         default:
15             lose = 1;
16             break;
17
18     }
19
20
21
22     if (lose != 0) {
23         exit(1);
24     }
25 }
```

```
fn parse_options(..., m: *mut *const char){
    match optc {
        103 => {
            *m = ...;
            block = 0;
        }
        128 => {
            print_help();
            block = 1;
        }
        116 => {
            block = 1;
        }
        _ => {
            lose = 1;
            block = 0;
        }
    }
    if block == 1 {
        *m = ...;
    }
    if lose != 0 {
        exit(1);
    }
}
```

The left is the original C code, and the right is C2Rust's translation. The parameter `m` is a must-output parameter because (1) the function writes to `m` unless it reaches `default` (line 4 lacks `break`), and (2) the `default` case sets `lose` to `1`, leading the program to exit. Since Rust's `match` does not support fall-through, C2Rust converts `switch` with fall-through into an assignment of a specific value to a temporary variable named `block`, followed by a conditional statement checking `block`. When control flow joins (line 7), the analyzer considers both `0` and `1` as possible values for `block` and `lose`, failing to recognize that the function always exits when it has not written to `m`. It suggests that analyzing the original C code can lead to more precise results compared to analyzing the translated Rust code. This incorrect classification of `m` as a may-output parameter leads to less idiomatic code, where the function always returns `Some` but never `None`, after the transformation. Nevertheless, this does not alter the program's behavior. As our experiments and manual investigation did not reveal any false positives, we conclude that the analysis is sufficiently precise for practical use.

4.4.6 RQ4: Impact on Performance

One possible concern when removing output parameters is the potential performance degradation of the target program. Compared to writing to an output parameter, directly returning a value may result in copying the value, which can be costly, especially when dealing with large structs. However, due to compiler optimizations, copying may not occur even when values are returned, and the performance impact might be negligible.

To investigate this issue, we measure the performance by executing the test suites. Since they were

not originally designed for performance evaluation, they may not exhibit consistent execution times. To ensure reliable results, we excluded test suites with execution times under 0.1 seconds, resulting in 20 remaining programs. Additionally, each test suite was executed twelve times, and we calculated the average execution time from ten runs, excluding the fastest and slowest results. We compare the execution times of the Rust programs before and after the transformation. We compiled them with the `--release` flag to enable optimizations. Our results show that the transformed programs exhibit only a 0.5% slowdown on average compared to the original ones. The original programs outperform in 11 cases, and the transformed programs outperform in 9 cases. This suggests that removing output parameters does not lead to meaningful performance degradation.

4.4.7 Threats to Validity

The primary threats to the validity of our evaluation are associated with the use of test suites for the correctness and performance assessments. Passing the test suites does not guarantee program correctness. However, in practice, test suites are the most widely used means of checking program semantics. Furthermore, during the development process, we discovered bugs in our implementation that resulted in semantics changes through the execution of the test suites. Regarding performance, it remains unclear whether these test suites can reveal performance differences, as they were not designed for this purpose. To complement our evaluation, we additionally conducted a performance comparison between the original C programs and the C2Rust-generated programs and observed an 11.5% average slowdown due to the translation. While investigating the reasons for this slowdown is outside the scope of this work, this result suggests that the test suites can effectively expose significant performance differences if they exist.

Chapter 5. Translation Using a Large Language Model

To complement the translation using static analysis, we need to develop a method that can translate any C features. In particular, we focus on migrating types in *function signatures* (parameter and return types) during translation. Our goal is to port an entire C program to Rust by translating each C function to a Rust function with a signature containing appropriate Rust types. Although types in function bodies also require migration, migrating types in signatures already poses several challenges. Thus, we make a significant step towards a type-migrating translation by addressing the difficulties in signature type migration.

The first challenge is that type migration cannot be achieved through syntactic mappings between type names. This hinders the adoption of existing *API mapping mining* techniques [204, 159], which automatically extract type mappings from existing codebases. API mapping mining has proven useful for Java-to-C# translation due to the one-to-one correspondence between most Java types and their C# counterparts. Once the mappings are constructed, a translator can migrate types by syntactically replacing a Java type with its corresponding C# type. In contrast, C-to-Rust translation involves an m-to-n correspondence. Depending on the context, a single C type can be migrated to multiple Rust types, and multiple C types can be migrated to a single Rust type. To migrate types, a translator must understand the semantics of functions and choose proper Rust types according to Rust idioms.

The second challenge arises from restructuring the bodies of a function and its callers after migrating types in the signature. Restructuring bodies requires a precise understanding of how parameters are used and how the return value is constructed in the function, as well as how arguments are constructed and the return value is used in the caller. This cannot be accomplished by simply replacing function names used by library function calls.

To address this problem, this work proposes leveraging LLMs, such as ChatGPT [165, 46], for C-to-Rust translation. LLMs are trained on vast collections of human-written code, providing an intuitive understanding of program semantics and programming idioms. This understanding includes knowledge of which Rust type should replace a certain C type in a signature. Consequently, LLMs have the potential to translate C functions to Rust while migrating the types in their signatures.

Unfortunately, simply requesting LLMs to translate each C function does not yield satisfactory results for various reasons. First, they often retain a C type as it is or migrate a C type to a Rust type not following Rust idioms. Second, they frequently fail to properly restructure the function bodies due to the lack of information about how the types in the signatures of their callees are migrated. Third, since current LLMs have limited ability to handle formal reasoning tasks, they often produce code that does not adhere to Rust's strict typing rules. These problems collectively result in generating Rust functions with unmigrated or improperly migrated types and a huge number of type errors. Such results significantly harm the usability of the translator, as programmers have to manually migrate more types and fix type errors to obtain safe and compilable Rust code. Therefore, naïvely applying LLMs to the type migration problem does not form an effective solution. We need techniques to effectively utilize LLMs for C-to-Rust translation by bringing out their capabilities at the maximum.

As a solution, we propose techniques to address these limitations and achieve effective type-migrating C-to-Rust translation using LLMs. First, to increase the possibility of migrating each C type to a proper Rust type, we explicitly instruct the LLM to perform the following steps: (1) generate multiple candidate

Rust signatures for each C function; (2) translate the function using each candidate signature; and (3) select the most idiomatic translation. Second, to facilitate proper restructuring of function bodies, we translate callees before their callers and provide the translated callees' signatures to the LLM when translating the callers. Third, to aid the LLM in producing code that adheres to the typing rules, we leverage compiler feedback. When the compiler suggests a fix in an error message, we apply it to the code; otherwise, we provide the error message to the LLM, enabling it to generate fixed code. Our techniques allow the translator to produce Rust code with more migrated types and fewer type errors compared to the naïve application of LLMs, significantly reducing the burden on programmers for manual code fixes after automatic translation.

Overall, our contributions are as follows:

- We use LLMs for type-migrating C-to-Rust translation and identify the challenges in this approach. We also propose techniques to address these challenges. (Section 5.1)
- We concretize the proposed approach as a tool named Tymcrat and evaluate it with 39 GNU programs written in C. We observe a 63.5% increase in migrated types and a 71.5% decrease in type errors compared to the baseline (the naïve LLM-based translation) with modest performance overhead. (Section 5.2)

5.1 Translation

This section presents techniques to translate C to Rust while migrating types in signatures and minimizing type errors. Type-migrating translation requires the translator to migrate types based on the understanding of functions' semantics and Rust idioms. In addition, the translator needs to restructure function bodies after type migration by handling various code patterns. Given the huge number of Rust types, designing static analysis and rewriting rules tailored to each Rust type is infeasible.

As a solution, we leverage LLMs, which possess knowledge of program semantics and language idioms derived from training on a vast corpus of human-written idiomatic code. LLMs have shown promising abilities in code-related tasks, including code generation [70, 79, 132, 133, 200], program repair [85, 199], and code summarization [49]. This leads us to expect that LLMs can properly migrate types and rewrite function bodies. However, as we show in this section, simply instructing the LLM to translate each C function does not yield satisfactory results. The remainder of this section describes the challenges in applying LLMs to type-migrating translation and proposes techniques to address them.

Figure 5.1 presents the proposed approach's workflow, which comprises several steps. Initially, we syntactically construct a call graph of a given C program by identifying the callees' names in each function. We then translate each function individually, as is common in neural code translation [160, 117, 161, 71, 175]. In our approach, the order in which the function is translated is important: we translate leaf nodes of the call graph first and then move towards their parents. This ensures that we translate each function after all of its callees have been translated and obtained Rust signatures.

We employ a four-step process to translate each function. First, we generate candidate Rust signatures (Section 5.1.1). Second, we augment the function with Rust signatures of its callees and translate it to Rust for each candidate signature (Section 5.1.2). Third, we type-check the translated code and iterate to resolve type errors using compiler feedback (Section 5.1.3). Finally, we select the most suitable translation (Section 5.1.4) and record the signature of the translated function to use it while translating its callers. Each step is explained in-depth in the remaining section.

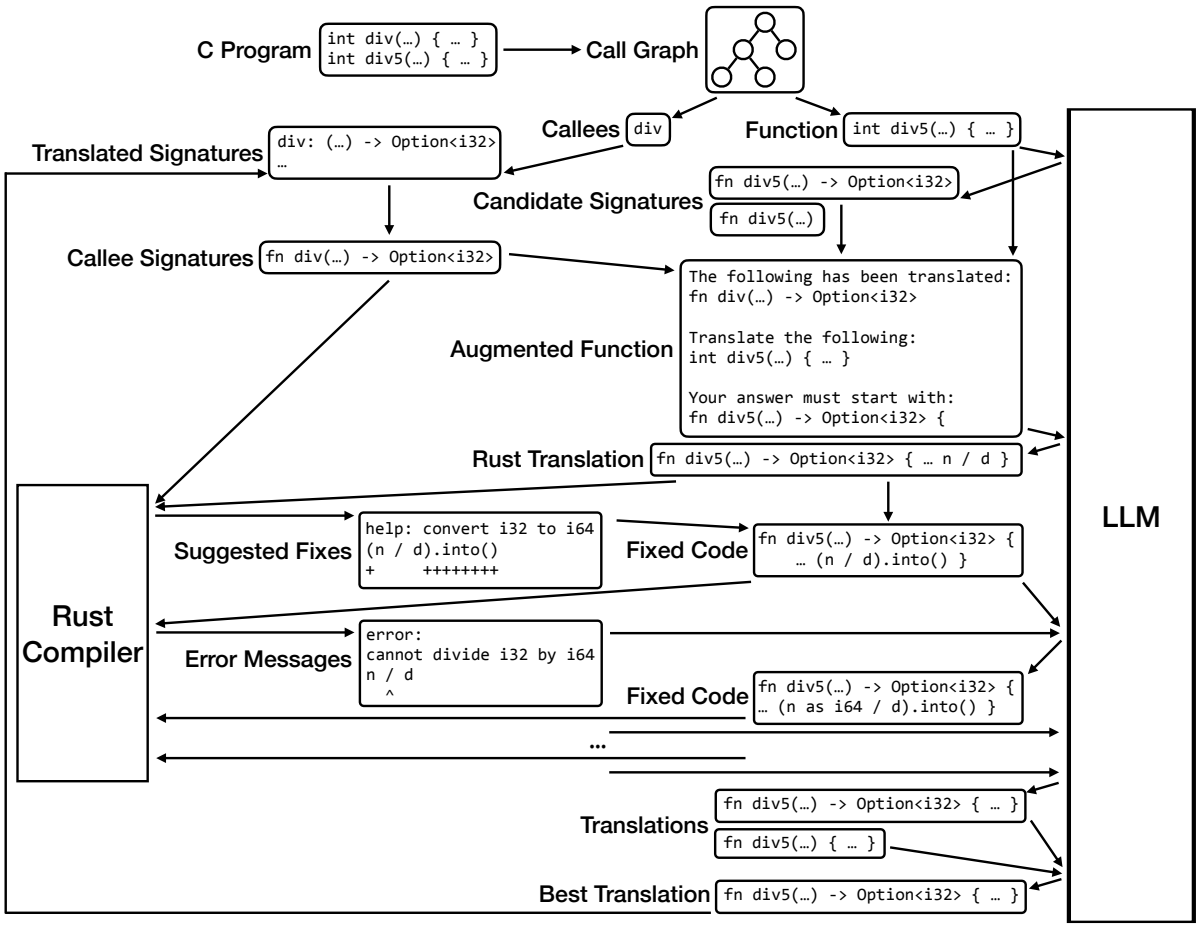


Figure 5.1: Overview of type-migrating translation via LLM

Note that examples in this section are prompts and responses from ChatGPT [165, 46] with simplifications. The examples focus on chat prompts, given that ChatGPT is trained for chat completion. However, we posit that our approach can be adapted to utilize LLMs for text completion by tweaking the prompts.

5.1.1 Candidate Signature Generation

We generate candidate signatures for a given C function using the LLM. This step is crucial because directly requesting the LLM to translate the function may not migrate types correctly. For example, the LLM translates `div` as follows:

```
[Prompt]
Translate the following function using Rust idioms:
int div(int n, int d, int *q) {
    if (d == 0) {
        return 1;
    }
    *q = n / d;
    return 0;
}
```

```
[Response]
fn div(n: i32, d: i32, q: &mut i32) -> i32 {
    if d == 0 {
        return 1;
    }
    *q = n / d;
    return 0;
}
```

Although `Option` is commonly used for partial functions in real-world Rust code on which the LLM has been trained, it fails to migrate the pointer type to `Option`.

To tackle this problem, we employ a technique in which we request the LLM to generate candidate signatures. This approach aligns with the eliciting reasoning with prompting method [148], e.g., incorporating a step-by-step thinking prompt [125]. By instructing the LLM to produce candidate signatures and use them to translate the function, we can increase the possibility of properly migrating types. An example is demonstrated below:

```
[Prompt]
Consider the following C function:
int div(int n, int d, int *q) {
    if (d == 0) {
        return 1;
    }
    *q = n / d;
    return 0;
}
If this function was written in Rust, what would be
its signature? Give 2 idiomatic signatures.
```

```
[Response]
fn div(n: i32, d: i32, q: &mut i32) -> i32;
fn div(n: i32, d: i32) -> Option<i32>;
```

One of the two proposed signatures migrates `int *` to `Option<i32>`. We set the number of candidate signatures to two for brevity in this example, while it is configurable. To prevent type errors in signatures from adversely affecting the subsequent steps, we discard signatures with type errors and proceed only with those without type errors.

5.1.2 Translation of Function Augmented with Callee Signatures

We translate the function to Rust with each candidate signature using the LLM. While our approach involves augmenting the function with the signatures of its callees, we initially consider a function that has no callees for illustration purposes. To translate a function, we supply the C code and the candidate signature to the LLM. The resulting outcome is a Rust function that has the desired signature, as illustrated in the following example:

```
[Prompt]
Translate the following function using Rust idioms:
int div(int n, int d, int *q) {
    if (d == 0) {
```

```
        return 1;
    }
    *q = n / d;
    return 0;
}
Your answer must start with:
fn div(n: i32, d: i32) -> Option<i32> {
```

```
[Response]
fn div(n: i32, d: i32) -> Option<i32> {
    if d == 0 {
        return None;
    }
    Some(n / d)
}
```

We now deal with functions that have callees. For instance, consider the function `div5`, which calls `div` to divide `5` by a given divisor. If we translate `div5` without providing the translated signature of `div` to the LLM, it fails to correctly restructure the body of `div5`, as shown in the following example:

```
[Prompt]
Translate the following function using Rust idioms:
int div5(int d, int *q) {
    return div(5, d, q);
}
Your answer must start with:
fn div5(d: i32) -> Option<i32> {
```

```
[Response]
fn div5(d: i32) -> Option<i32> {
    let mut q = 0;
    let res = div(5, d, &mut q);
    if res == 0 {
        Some(q)
    } else {
        None
    }
}
```

The body of the translated function passes three arguments to `div` and treats the return value as an integer, which matches the original C code. However, it is incompatible with the translated signature of `div`, which takes two arguments and returns `Option`. As a result, the translated code encounters type errors. The problem is that the LLM lacks knowledge about the translated signature of `div`.

To address this issue, we augment the function by incorporating the callees' signatures before feeding the code into the LLM. By starting the translation from the leaf nodes of the call graph and moving towards their parents, we ensure that the translated signatures of the callees are available. Note that signatures with type errors are already discarded during candidate signature generation, guaranteeing that the signature of every callee used for function augmentation is free of type errors. The following example demonstrates the translation using augmented code:

```
[Prompt]
The following definition has been translated:
fn div(n: i32, d: i32) -> Option<i32>;
Translate the following function using Rust idioms:
int div5(int d, int *q) {
    return div(5, d, q);
}
Your answer must start with:
fn div5(d: i32) -> Option<i32> {
```

```
[Response]
fn div5(d: i32) -> Option<i32> {
    div(5, d)
}
```

Considering the provided signature of `div`, the LLM successfully restructures the body to pass two arguments to `div` and directly return the `Option` value. This code has no type errors.

One challenge with this approach arises from the presence of mutually recursive functions. When functions are mutually recursive, it is impossible to translate all the callees before translating a function. To address this issue, we arbitrarily choose one of the mutually recursive functions and translate it without the signatures of the other functions involved in the mutual recursion. This allows us to subsequently translate the remaining functions within the mutual recursion cycle using the translated signature of the chosen function. Once the other functions have been translated, we revisit the initially chosen function and translate it again, this time considering the translated signatures of the others. This improves the translation compared to the initial trial. During this second translation, we maintain the signature determined in the initial trial to ensure that the translation of the others, which relies on the initial signature, remains valid.

Another potential concern is the presence of function pointers. However, we believe that our approach does not require special treatment for function pointers. If a function takes a function pointer as an argument, the type of the function pointer is specified in the original signature of the function itself, and the LLM can utilize this information. If a function passes a function pointer to another function, the LLM can properly translate it because we augment the function with the signature of the callee, which provides the type of the function pointer.

5.1.3 Compiler Feedback-Based Iterative Fix

After translating each function, we type-check the function along with its callees. During type checking, the body of each callee is temporarily replaced with a `todo!` macro invocation while retaining the signature, as shown below:

```
fn div(n: i32, d: i32) -> Option<i32> {
    todo!()
}
```

`todo!` is a built-in macro that can be called anywhere, regardless of the expected type of the location. By using `todo!` instead of the actual translated body, we prevent the type checking from being affected by type errors in the callees.

Despite augmenting functions with callee signatures, the LLM still frequently generates code with type errors. To address this issue, we employ an iterative approach to resolve type errors based on the Rust compiler's feedback. The compiler presents two kinds of error messages: those accompanied by suggested fixes and those without any suggested fixes. Our error resolution strategy handles these two kinds differently. We begin by illustrating each kind of error message through examples and subsequently describe our approach to fix them. Specifically, we select code snippets that exhibit missing type casts as examples due to their simplicity.

Consider the following `div` function that produces `long`, not `int`:

```
int div(int n, int d, long *q) {
    ...
    *q = n / d;
    ...
}
```

In C, the conversion between different integer types is implicit. Thus, the result of division can be assigned to `q` even if `q` has type `long`, while the result of the division has type `int`. When translating this code using the LLM, the resulting Rust code is as follows:

```
fn div(n: i32, d: i32) -> Option<i64> {
    ...
    Some(n / d)
}
```

However, Rust requires explicit type casts for every conversion between integer types. Consequently, the above code does not compile and produces the following error message:

```
error[E0308]: mismatched types
    Some(n / d)
         ^^^^^ expected `i64`, found `i32`
help: you can convert an `i32` to an `i64`
    Some((n / d).into())
         +      ++++++++
```

The compiler identifies that a value of type `i32` occurs where a value of type `i64` is expected and includes a suggested fix in the error message. The fix suggests inserting an `into` method invocation to cast `i32` to `i64`.

Unfortunately, not all error messages suggest a fix. Consider a slightly different scenario where the divisor is also a `long`:

```
int div(int n, long d, long *q) {
    ...
    *q = n / d;
    ...
}
```

In C, `n` is implicitly cast to a `long` before the division. The LLM translates the code as follows:

```
fn div(n: i32, d: i64) -> Option<i64> {
    ...
    Some(n / d)
}
```

Algorithm 5.1: Fix-by-suggestion algorithm

Input : *code*

Output: *code*, $errors_{\text{no-fix}}$

1 **def** *fix-by-suggestion*:
2 $errors_{\text{fix}}, errors_{\text{no-fix}} \leftarrow$ type-check(*code*);
3 **while** errors_{fix} *is not empty*:
4 *code* $\leftarrow$ apply-fix(*code*, $errors_{\text{fix}}$);
5 $errors_{\text{fix}}, errors_{\text{no-fix}} \leftarrow$ type-check(*code*);

Algorithm 5.2: Fix-by-LLM algorithm

Input : *code*

Output: *code*, *errors*

1 **def** *fix-by-llm*:
2 *code*, *errors* $\leftarrow$ fix-by-suggestion(*code*);
3 **while** errors *is not empty*:
4 $code' \leftarrow$ llm-fix(*code*, *errors*);
5 $code', errors' \leftarrow$ fix-by-suggestion($code'$);
6 **if** *length*(errors) $\leq$ *length*(errors'):
7 break;
8 *code*, *errors* $\leftarrow code', errors'$;

Due to the lack of type cast, this code produces a type error:

```
error[E0277]: cannot divide `i32` by `i64`
    Some(n / d)
           ^ no implementation for `i32 / i64`
```

In this case, the compiler fails to suggest a fix because it recognizes only the absence of a division operator of `i32` that accepts an `i64`.

Algorithm 5.1 shows the initial phase of error resolution, which leverages error messages that provide suggested fixes. We first distinguish such errors from those without fixes (line 2). We then apply the fixes to the code (line 4) and type-check the code again (line 5). This iterative process continues until no error messages with suggested fixes remain (line 3).

After the initial phase, the resolution of errors without suggested fixes becomes necessary. To generate the corrected code, we supply the erroneous code along with the error message to the LLM. In cases where multiple errors occur, we simultaneously provide all the error messages to the LLM, rather than making separate queries for each error message. This approach is motivated by that multiple errors are often related to each other, enabling the LLM to fix the code by considering all the errors collectively. A concrete illustration of this process is provided below:

```
[Prompt]
The following code has a compilation error:
fn div(n: i32, d: i64) -> Option<i64> {
    if d == 0 {
        return None;
    }
```

```
    Some(n / d)
}
The error message is:
error[E0277]: cannot divide `i32` by `i64`
    Some(n / d)
           ^ no implementation for `i32 / i64`
Write the code of the fixed function.
```

```
[Response]
fn div(n: i32, d: i64) -> Option<i64> {
    if d == 0 {
        return None;
    }
    Some(n as i64 / d)
}
```

The LLM may fix the code by considering the error message.

Algorithm 5.2 describes the iterative process of minimizing type errors through the aforementioned LLM-based fix generation. When the LLM generates fixed code (line 4), we type-check the code and apply all the compiler-suggested fixes (line 5). Next, we assess if the number of type errors has decreased compared to the original code (line 6). If not, we classify the fix as unsuccessful, discard it, and stop the iteration (line 7). Otherwise, we consider the fix successful and provide the corrected code and the remaining errors to the LLM for a further fix (line 8). The iteration continues as long as the fix is successful, terminating when no type errors remain (line 3).

5.1.4 Best Translation Selection

In the final step, we choose the most desirable translation from the available translations. Note that we have multiple translations because we translate a single function multiple times using different candidate signatures.

The primary criterion for selecting the best is the number of type errors. Therefore, we select the translation that exhibits the fewest type errors. It allows us to minimize type errors.

However, multiple translations can have the same number of type errors. In such cases, we rely on the LLM to select the most suitable translation, taking Rust idioms into account. If there are more than two translations, we compare two at a time until the best one is determined. This constraint arises from the token limit imposed by the LLM API. When a function is lengthy and multiple translations exist, collecting the code of all translations may exceed the token limit of the prompt. To address this issue, we only compare two translations at a time, allowing the function to occupy up to half of the token limit. An example comparison through the LLM is provided below:

```
[Prompt]
Implementation 1
fn div(n: i32, d: i32, q: &mut i32) -> i32 {
    if d == 0 {
        return 1;
    }
    *q = n / d;
    return 0;
```

```
}
Implementation 2
fn div(n: i32, d: i32) -> Option<i32> {
    if d == 0 {
        return None;
    }
    Some(n / d)
}
Which one is more Rust-idiomatic?
```

```
[Response]
Implementation 2
```

The LLM is expected to choose a translation that follows Rust idioms. For instance, if a function is a partial function, a translation that migrates a pointer type to `Option` would be favored. Conversely, if a function always succeeds, a translation that does not migrate a pointer type to `Option` would be favored because it is not idiomatic for a function to always return `Some` and never return `None`.

5.2 Evaluation

In this section, we first provide an overview of our implementation (Section 5.2.1) and the process of collecting benchmark programs (Section 5.2.2). We then evaluate the effectiveness of the proposed approach with the following five research questions:

- RQ1. Promotion of type migration: Does the proposed approach effectively promote type migration by generating candidate signatures? (Section 5.2.3)
- RQ2. Quality of type migration: Do the Rust types introduced by the proposed approach adhere to Rust idioms? (Section 5.2.4)
- RQ3. Type error reduction: Does the proposed approach effectively reduce type errors by augmenting functions and iteratively fixing errors? (Section 5.2.5)
- RQ4. Comparison with existing approaches: How does the translation of the proposed approach differ from that of the existing approaches? (Section 5.2.6)
- RQ5. Overhead: Does the proposed approach entail reasonable overhead? (Section 5.2.7)

Our experiments were conducted on an Ubuntu machine with Intel Core i7-6700K (4 cores, 8 threads, 4GHz) and 32GB DRAM. Finally, we dicuss threats to validity (Section 5.2.8).

5.2.1 Implementation

We implemented the proposed approach as a tool, Tymcrat. It is built on the Rust compiler, enabling access to the compiler's internal diagnostic data structures. This allows us to easily extract the suggested fixes from error messages without the need for text processing. For Tymcrat's language model, we use GPT-3.5 Turbo or GPT-4o mini, specifically the gpt-3.5-turbo-0125 and gpt-4o-mini-2024-07-18 models. GPT-4o mini has higher intelligence than GPT-3.5 Turbo [47]. We employ both models to assess our approach's effectiveness with models with different capabilities. Although GPT-4o is the most powerful

model offered by OpenAI, we use GPT-4o mini instead due to its lower cost. GPT-4o is ten times more expensive than GPT-3.5 Turbo, while GPT-4o mini is cheaper than GPT-3.5 Turbo. We set the temperature of the models to 0 to ensure that the behavior of the language model is mostly deterministic, although some nondeterministic behavior may still occur [9]. Tymcrat interacts with ChatGPT through the API provided by OpenAI.

Unfortunately, the token limit of the ChatGPT API poses a restriction on Tymcrat's ability to translate lengthy functions. Specifically, Tymcrat does not translate functions exceeding 3,000 tokens. This is because gpt-3.5-turbo-0125 has an output token limit of 4,096, and a translated Rust function typically requires more tokens than the original C function. Although gpt-4o-mini-2024-07-18 can output up to 16,384 tokens, we exclude functions exceeding 3,000 tokens even with this model to ensure a fair comparison between the two models.

We leverage parallelism to enhance the speed of translation. Although the translation of a caller and a callee cannot occur simultaneously, many functions, such as the leaf nodes in the call graph, are independent of one another, allowing simultaneous translation. The translation of a function using different candidate signatures is also independent of each other and thus peformed in parallel.

Since type definitions and global variables are common in real-world C programs, Tymcrat needs to translate them to handle entire programs. However, this work mainly focuses on migrating types in function signatures. Therefore, Tymcrat does not ask the LLM to generate candidate signatures for type definitions and global variables but simply requests their translation. In addition, when translating a function, Tymcrat augments it with not only its callees' signatures but also the translations of the type definitions and global variables used in the function.

5.2.2 Benchmark Collection

we collected 41 GNU packages written in C as benchmark programs. while previous studies on C-to-Rust translation provide benchmark sets, they mostly consist of small programs ($<$ 5k LOC). Since we believe that large programs with many types to migrate are more appropriate for demonstrating the characteristics of the proposed approach, we decided to use GNU packages as benchmark programs. Initially, we gathered all the C programs from the packages listed in GNU Package Blurbs [6]. We then filtered out programs that exceeded 100,000 lines, as measured by `cloc` [76], and those that do not compile. This process yields 88 packages. From these, we selected 41 packages that are considered especially famous, determined by whether the package has an individual entry on Wikipedia.

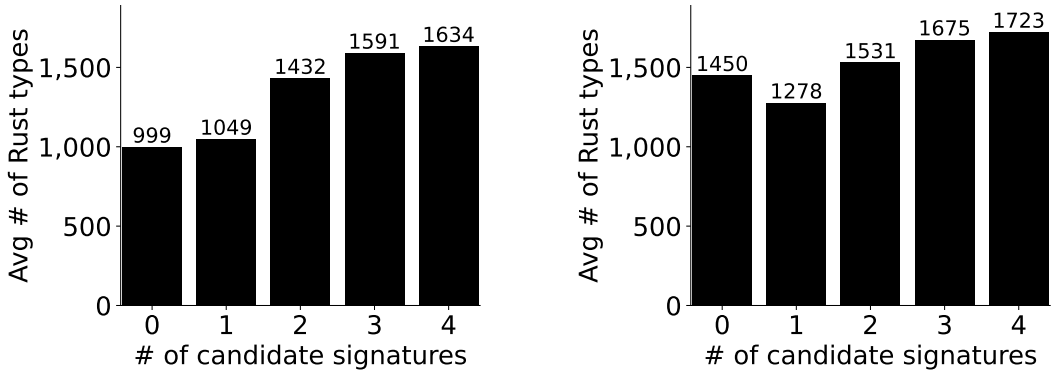
Table 5.1 presents the collected programs and their respective code sizes. The second column displays the number of lines of C code; the third to sixth columns indicate the numbers of type definitions, global variable declarations, function definitions, and call edges, respectively; the last column shows the numbers of functions omitted from translation due to exceeding 3,000 tokens.

5.2.3 RQ1: Promotion of Type Migration

To evaluate the effectiveness of the proposed approach in promoting type migration, we compare five settings, ranging from 0 to 4 candidate signatures generated for each function. Zero candidate signatures mean directly translating each function without generating candidates. We denote the number of candidate signatures with a subscript, resulting in settings from Tymcrat_0 to Tymcrat_4. Tymcrat_0 serves as the baseline, allowing us to investigate how type migration is promoted as we generate more candidate signatures.

Table 5.1: Benchmark programs for evaluating Tymcrat

Program	LOC	Types	Variables	Functions	Calls	Omitted
time-1.9	796	4	12	29	184	0
which-2.21	998	5	35	33	242	0
libtool-2.4.7	2255	30	18	101	429	0
ed-1.19	2419	11	62	132	790	0
hello-2.12.1	3699	15	16	142	507	1
pth-2.0.7	5046	51	44	202	1119	1
units-2.22	5127	22	56	136	1496	1
pexec-1.0rc8	5149	26	4	150	1382	2
gzip-1.12	6383	41	151	220	1144	2
adns-1.6.0	7132	58	80	433	2009	1
indent-2.2.13	7613	27	171	119	877	3
bc-1.07.1	7878	62	110	219	1518	1
cflow-1.7	12256	79	123	455	1878	4
libosip2-5.3.1	13219	136	27	682	3742	3
rcs-5.10.1	13607	77	89	452	2711	3
mtools-4.0.43	13687	107	116	582	2823	2
mcsim-6.2.0	14782	100	53	447	3286	3
less-633	15508	42	408	637	2833	1
make-4.4.1	15556	56	163	415	3942	4
patch-2.7.6	15601	72	186	529	2782	9
enscript-1.6.6	16693	139	270	229	3170	9
sed-4.9	16751	114	70	627	3094	4
cpio-2.14	16999	82	158	607	3311	6
readline-8.2	19373	71	421	725	3464	3
nettle-3.9	19605	186	173	967	3194	8
dap-3.10	20923	15	150	319	5719	10
diffutils-3.10	23999	132	158	723	3704	6
grep-3.11	24028	148	131	783	3699	5
m4-1.4.19	25125	175	129	933	4463	4
nano-7.2	25711	105	191	762	6466	6
screen-4.9.0	30837	53	240	673	5160	8
gmp-6.2.1	31576	80	83	898	7920	26
gprolog-1.5.0	31781	131	574	1660	6690	4
findutils-4.9.0	32245	190	134	1105	6063	7
bison-3.8.2	32390	294	325	1600	6631	6
uucp-1.07	36520	78	519	756	7035	13
parted-3.6	38958	434	222	1650	7585	8
tar-1.34	41632	215	362	1475	8570	8
gawk-5.2.2	45989	214	368	1257	10510	11
wget-1.21.4	48184	198	166	1200	7908	15
glpk-5.0	59030	338	24	1492	13482	12



(a) GPT-3.5 Turbo

(b) GPT-4o mini

Figure 5.2: Average number of Rust types introduced after translation

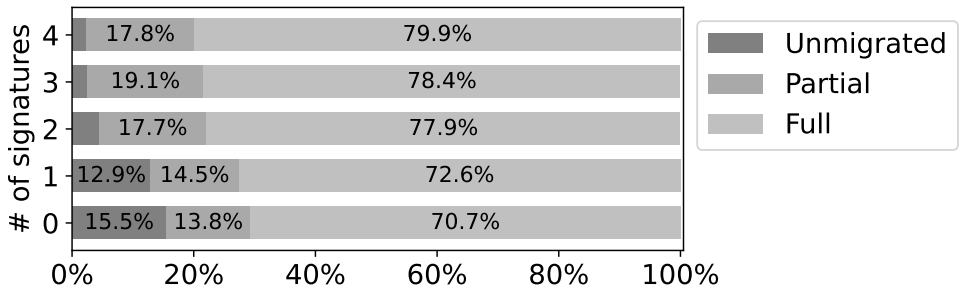
For the evaluation, we classify types into three categories: common types, C types, and Rust types. Common types are types that exist in both C and Rust and can be safely used in Rust. These include array, `bool`, `char`, floating point, integer, and unit types. Such types do not require migration. C types are types used in C and can still be used in Rust but compromise the safety guarantee of the type checker. These include C pointer types, the types provided by `std::os::raw`, `std::os::unix::raw`, `std::os::linux::raw`, `std::os::fd::raw`, and `core::ffi` of the Rust standard library, and the types provided by the official `libc` [2] library for Rust. The goal of type migration is to avoid using these types in Rust code. Finally, Rust types are types exclusive to Rust, and their safety is guaranteed by the type checker. These include never, reference, slice, `str`, and tuple types, as well as the types provided by the Rust standard library other than the aforementioned C types. These types should be introduced during type migration.

We now evaluate the number of Rust types introduced by the translation. If a single type occurs multiple times, we count all occurrences. Figure 5.2 illustrates the average number of Rust types introduced in benchmark programs translated by Tymcrat_0 to Tymcrat_4.

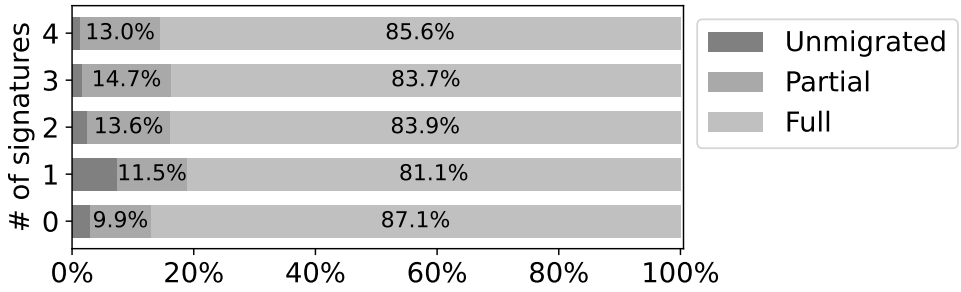
When using GPT-3.5 Turbo (Figure 5.2a), Tymcrat_1 to Tymcrat_4 introduce 5.0%, 43.3%, 59.2%, and 63.5% more Rust types, respectively, compared to the baseline. These results show that generating two or more candidates for each function effectively promotes type migration, while generating only one candidate does not exhibit significant improvement over the baseline. This underscores the importance of generating *multiple* candidates to consider various options for promoting type migration. While the gains from generating more than two candidates are small, they still lead to improved results. As shown in Section 5.2.7, the translation time increases with the number of candidates, and users can configure the number of candidates considering the trade-off between translation time and quality.

When using GPT-4o mini (Figure 5.2b), Tymcrat_1 to Tymcrat_4 introduce -11.9%, 5.6%, 15.5%, and 18.8% more Rust types, respectively, compared to the baseline. These results lead to the same conclusion as with GPT-3.5 Turbo, highlighting the importance of generating multiple candidates, although the gain is smaller than with GPT-3.5 Turbo. Even Tymcrat_0 introduces an average of 1,450 Rust types, comparable to Tymcrat_2 using GPT-3.5 Turbo. This aligns with OpenAI's claim that GPT-4o mini has higher intelligence than GPT-3.5 Turbo.

We also evaluate the number of signatures migrated by the translation. For this evaluation, we categorize signatures into three groups: unmigrated, partially-migrated, and fully-migrated signatures. Unmigrated signatures contain C types but no Rust types. They represent the worst case, as none of



(a) GPT-3.5 Turbo



(b) GPT-4o mini

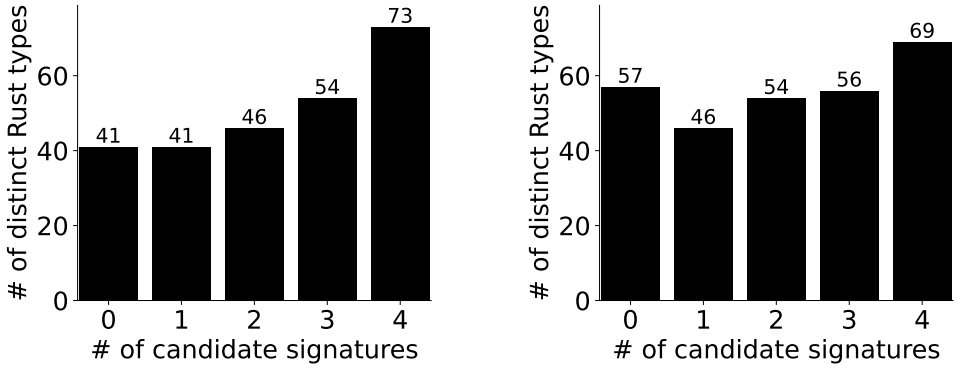
Figure 5.3: Proportions of unmigrated, partially migrated, and fully migrated function signatures after translation

the types in the signature have been migrated. Partially-migrated signatures contain both C and Rust types, representing an improvement over unmigrated ones but still with room for enhancement. Fully-migrated signatures consist of only common types and Rust types, representing the best case as they do not contain any C types. Figure 5.3 depicts the proportions of unmigrated, partially-migrated, and fully-migrated signatures in all benchmark programs translated by Tymcrat_0 to Tymcrat_4.

When using GPT-3.5 Turbo (Figure 5.3a), Tymcrat_0 and Tymcrat_1 fully migrate 70.7% and 72.6% of the signatures, respectively, while Tymcrat_2 to Tymcrat_4 achieve 77.9%, 78.4%, and 79.9% full migration, respectively. This aligns with the observation on the number of Rust types: generating two or more candidates effectively promotes type migration. Comparing Tymcrat_0 and Tymcrat_4, the proposed approach increases fully-migrated signatures by 14.8% and decreases unmigrated types by 84.7% compared to the baseline.

When using GPT-4o mini (Figure 5.3b), Tymcrat_0 fully migrates 87.1% of the signatures, while Tymcrat_1 to Tymcrat_4 fully migrate 81.1%, 83.9%, 83.7%, and 85.6%, respectively. These results show that generating candidate signatures is less effective in increasing the number of fully-migrated signatures with GPT-4o mini. However, Tymcrat_0 leaves 3.0% of the signatures unmigrated, while Tymcrat_2 to Tymcrat_4 leave only 2.5%, 1.5%, and 1.4% unmigrated, respectively. Comparing Tymcrat_0 and Tymcrat_4, the proposed approach reduces unmigrated signatures by 50.9% compared to the baseline. Thus, generating candidate signatures effectively reduces the number of unmigrated signatures even when using GPT-4o mini.

From these experimental results, we conclude that candidate signature generation significantly promotes type migration, especially when using GPT-3.5 Turbo. It is also useful when using GPT-4o mini, but the improvement is smaller. Despite GPT-4o mini being available, users may choose to employ



(a) GPT-3.5 Turbo

(b) GPT-4o mini

Figure 5.4: Number of distinct Rust types introduced after translation

GPT-3.5 Turbo or other open-source models with similar capabilities. Thus, the proposed approach can be utilized in practice to effectively promote type migration.

Summary of RQ1: Promotion of Type Migration

- Generating multiple candidate signatures significantly promotes type migration.
- Compared to GPT-3.5 Turbo, GPT-4o mini demonstrates higher baseline performance, and the improvement from generating candidates is smaller.
- Generating candidates substantially increases the percentage of fully-migrated signatures for GPT-3.5 Turbo, but not for GPT-4o mini.
- Generating candidates effectively reduces the proportion of unmigrated signatures for both GPT-3.5 Turbo and GPT-4o mini.

5.2.4 RQ2: Quality of Type Migration

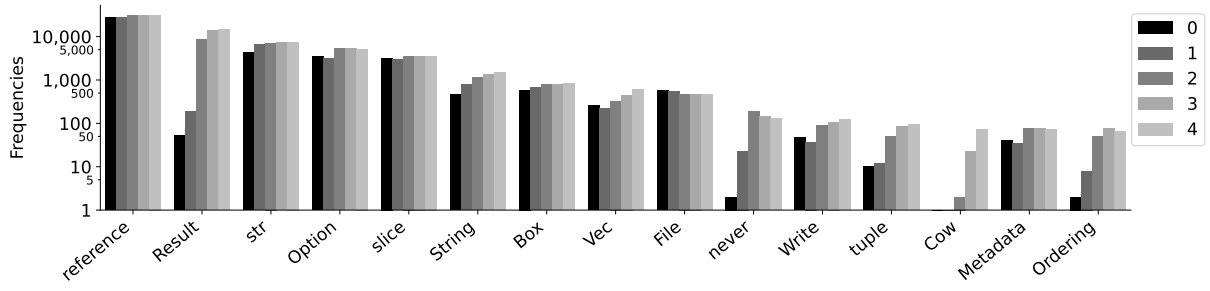
It is crucial for the translator not only to migrate many types but also to introduce types that adhere to Rust idioms. Therefore, we assess the quality of the types migrated by the proposed approach.

We measure the number of distinct Rust types introduced in signatures through the translation. Since real-world Rust code employs various Rust types, a higher count of distinct Rust types would indicate a more idiomatic translation. Figure 5.4 illustrates the number of distinct Rust types introduced in all benchmark programs translated by Tymcrat_0 to Tymcrat_4.

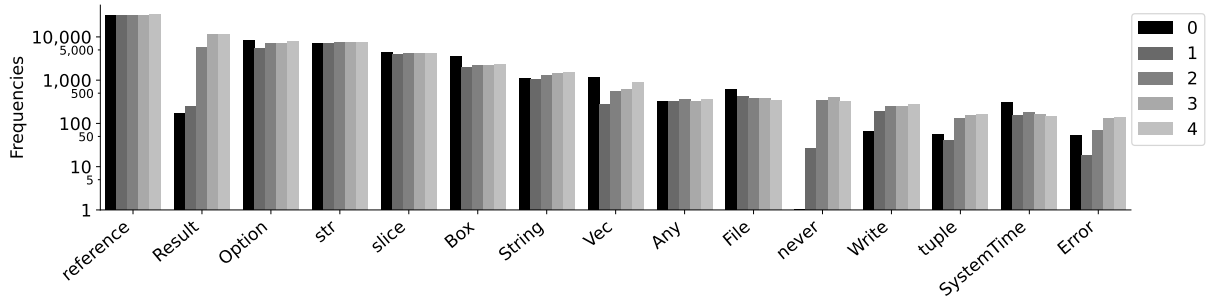
When using GPT-3.5 Turbo (Figure 5.4a), Tymcrat_0 and Tymcrat_1 introduce 41 distinct types, while Tymcrat_2 to Tymcrat_4 introduce 46, 54, and 73 distinct types, respectively. These results suggest that generating multiple candidates effectively enhances the diversity of introduced Rust types. Notably, generating four candidates for each function is particularly effective, achieving a 78.0% increase compared to the baseline.

When using GPT-4o mini (Figure 5.4b), Tymcrat_0 already introduces 57 distinct types, outperforming Tymcrat_1 to Tymcrat_3. This indicates that generating fewer than four candidates does not increase the number of distinct types. Only generating four candidates increases the count, achieving 69 distinct types, which is 21.1% higher than the baseline.

We also count the frequency of each introduced Rust type. This enables us to identify the commonly



(a) GPT-3.5 Turbo



(b) GPT-4o mini

Figure 5.5: Frequencies of each introduced Rust type after translation

introduced Rust types and understand the characteristics of type migration facilitated by our approach. The following fifteen types are the most frequently introduced by Tymcrat$_4$ using GPT-3.5 Turbo, in decreasing order: reference, `Result`, `str`, `Option`, slice, `String`, `Box`, `Vec`, `File`, never, `Write`, tuple, `Cow`, `Metadata`, and `Ordering`. In contrast, the top fifteen types from Tymcrat$_4$ using GPT-4o mini are: reference, `Result`, `Option`, `str`, slice, `Box`, `String`, `Vec`, `Any`, `File`, never, `Write`, tuple, `SystemTime`, and `Error`. While the two lists share many common types, `Cow`, `Metadata`, and `Ordering` appear only in the list from GPT-3.5 Turbo, whereas `Any`, `SystemTime`, and `Error` appear only in the list from GPT-4o mini.

Figure 5.5 shows the frequencies of these types in all benchmark programs translated by Tymcrat$_0$ to Tymcrat$_4$. The y-axis is shown on a log scale. When using GPT-3.5 Turbo (Figure 5.5a), Tymcrat$_4$ increases the frequency of all the types except `File`, compared to Tymcrat$_0$. Here, `File` decreases because it is replaced by `Write`, which is more idiomatic, and we further discuss this issue later with case studies. When using GPT-4o mini (Figure 5.5b), Tymcrat$_4$ increases the frequency of reference, `Result`, `str`, `String`, `Any`, never, `Write`, tuple, and `Error`, while decreasing the frequency of `Option`, slice, `Box`, `Vec`, `File`, and `SystemTime`. As with GPT-3.5 Turbo, `File` is replaced by `Write`. In addition, `Box`, `Vec`, and slice are replaced by `String` and `str`, which are more idiomatic by indicating the use of strings rather than arbitrary collections. Thus, these results suggest that generating multiple candidates increases the frequency of idiomatic Rust types.

For comparison, we also investigate the top fifteen Rust types in the signatures of the ten most-starred Rust projects. They are, in decreasing order: reference, `Option`, `str`, `String`, `Result`, `Vec`, slice, tuple, `Path`, `PathBuf`, `Arc`, `HashMap`, `Box`, `RefCell`, and `Rc`. Although the order differs slightly, nine types are common between the top fifteen types in our translation and those in real-world code, suggesting that the proposed approach introduces idiomatic Rust types. `File`, never, `Write`, `Cow`, `Metadata`, and `Ordering` appear only in our list from GPT-3.5 Turbo; `Any`, `File`, never, `Write`, `SystemTime`, and `Error` appear only

in our list from GPT-4o mini. Since `Cow`, never, `Write`, and `Error` still rank in the top 30 in real-world code, they align with Rust idioms. However, `File`, `Metadata`, `Ordering`, `Any`, and `SystemTime` are rarely used in real-world code, which we discuss further with case studies. On the other hand, `Path`, `PathBuf`, `Arc`, `HashMap`, `RefCell`, and `Rc` are exclusive to the list from real-world code. Nonetheless, `Arc`, `RefCell`, and `Rc` also rank in the top 30 of Tymcrat$_4$'s translation using GPT-3.5 Turbo, and `Path` ranks in the top 30 of Tymcrat$_4$'s translation using GPT-4o mini. Increasing the frequency of the other types, `PathBuf` and `HashMap`, is worth considering as a future research direction.

To achieve a more accurate analysis of type migration quality, we conducted case studies involving the manual investigation of 300 functions in the translated code. Specifically, we randomly selected 10 functions for each of the top fifteen types introduced by Tymcrat$_4$ using GPT-3.5 Turbo and repeated this process with the code generated by GPT-4o mini. The goal was to determine whether the introduced Rust types align with Rust idioms. Using the standard formula for estimating proportions, this sample size provides a 10% margin of error with a 90% confidence level. The investigation was conducted independently by both the authors and another researcher familiar with Rust.

From the case studies, we found that the use of most types conforms to Rust idioms. Specifically, 123 out of 150 functions (82%) in GPT-3.5 Turbo's translation and 128 out of 150 functions (85%) in GPT-4o mini's translation align with Rust idioms. Another researcher independently reported similar findings, with 130 out of 150 functions for GPT-3.5 Turbo and 128 out of 150 for GPT-4o mini. Of the 300 functions, our conclusions differ from the other researcher's ones in only 19 cases. Given the subjective nature of idiomatic Rust, such discrepancies are expected. Despite this, the small number of disagreements suggests that the proposed approach introduces idiomatic Rust types in most cases.

Notably, `Metadata`, `Ordering`, and `SystemTime` adhere to Rust idioms despite their rare occurrences in real-world code. `Metadata` and `SystemTime` occur frequently in the translated code because the C types `stat` and `timespec` appear frequently in the benchmark programs. This implies that the benchmark programs often deal with file status and times, while the ten most-starred Rust projects do not due to their different domains. On the other hand, `Ordering` is the return type of functions that compare values, which are common in both the benchmark programs and the Rust projects. The occurrence of `Ordering` is rare in the Rust projects because developers often use the `derive` attribute to automatically implement comparison functions without manually writing code containing `Ordering`.

Nevertheless, some types in the translated code do not conform to Rust idioms and require further improvement. In the case of GPT-3.5 Turbo, 27 unidiomatic translations are due to the use of `Result`, `File`, and `Cow` types. For GPT-4o mini, 22 unidiomatic translations are attributed to the use of `Result`, `File`, and `Any` types.

First, the translation often introduces `Result` types, which are used for partial functions like `Option`. However, we observed that some functions have `Result` types in their signatures but are not partial functions in fact. This implies that the assumption made in Section 5.1.4, stating that the LLM would not choose a function with a `Result` type as the best translation if it is not a partial function, is not always valid. Exploring the use of static analysis to determine whether a function is partial would be interesting future research. It will allow the translator to instruct the LLM to introduce `Option` or `Result` and restructure the bodies only if the function is partial.

Second, the translation often introduces `File`, a type representing an open file. Although the LLM typically migrates C's `FILE` to Rust's `File`, they are not equivalent. `FILE` is a stream that can be read or written to, representing not only files but also standard input/output/error. In contrast, `File` denotes only files. Rust provides the `Write` and `Read` traits for writable and readable streams, respectively.

Therefore, if a function writes to `FILE` , it should be migrated to `Write`, and if it reads from `FILE` , it should be migrated to `Read`. Although increasing the number of candidates promotes the use of `Write`, the LLM still frequently chooses `File`. A promising future direction would be to more effectively reduce the occurrences of `File` than the current approach.

Third, the translation using GPT-3.5 Turbo often introduces `Cow`, which represents copy-on-write heap-allocated data. In Rust, a function takes a `Cow` pointer as input to copy the data before mutating it only when it is referenced by multiple pointers. However, GPT-3.5 Turbo frequently generates functions that *return* `Cow` pointers, even when the return values do not need to be passed to functions taking `Cow`. This is far from the idiomatic use of `Cow`, and a potential future direction is to improve the translation to avoid unnecessarily returning `Cow` pointers.

Fourth, the translation using GPT-4o mini frequently introduces `Any`, a type representing an arbitrary value. This is because `void *` in C code is often migrated to `Any` by the LLM. However, Rust programmers tend to avoid using `Any`; only one (`rustc`) among the ten most-starred Rust projects utilizes `Any` in signatures. Instead, they prefer using generics to define functions that can accept any value. An interesting future direction would be to reduce the occurrences of `Any` by translating functions taking `void *` to generic functions.

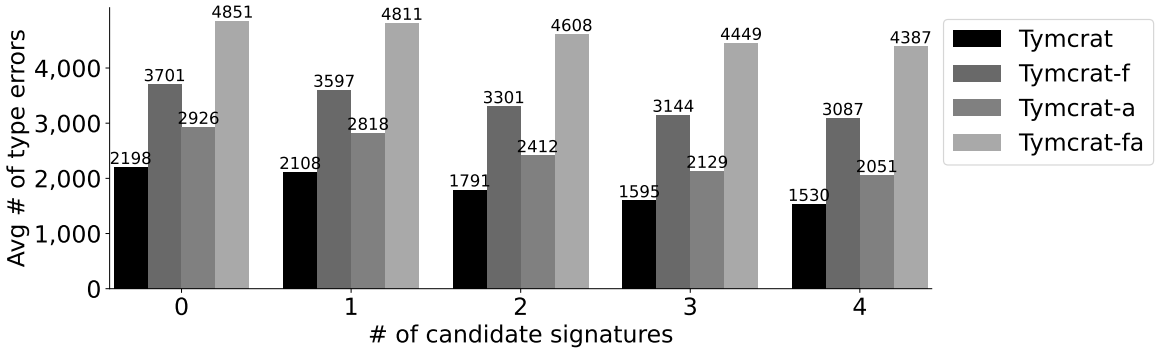
These examples demonstrate that using LLMs for candidate signature generation and best translation selection does not always yield accurate results. LLMs often produce signatures with non-idiomatic types and may favor translations with such signatures during selection. This suggests that adopting heuristic rules based on syntactic patterns or static analysis for these procedures could improve translation quality. We believe that our findings can guide the development of heuristic rules, moving toward effective integration of machine learning, static analysis, and heuristics for idiomatic and correct C-to-Rust translation.

Summary of RQ2: Quality of Type Migration

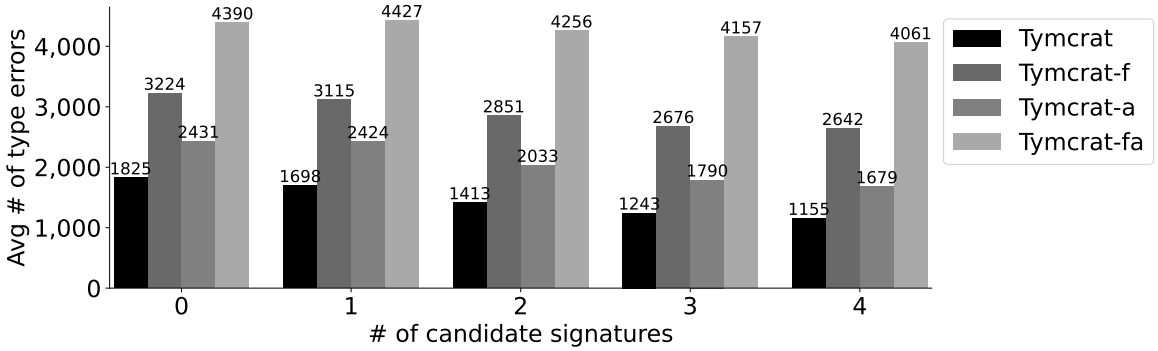
- Generating two or more candidate signatures enhances the diversity of introduced Rust types when using GPT-3.5 Turbo, but four candidates are needed for a similar effect with GPT-4o mini.
- Most of the top fifteen types in the translated code also frequently appear in real-world Rust code.
- Generating candidates often increases the occurrence of the top fifteen types, promoting the introduction of idiomatic Rust types.
- Future research should focus on improving type migration quality by reducing the incorrect use of `Result` and `Cow`, replacing `Any` with generics, and substituting `File` with `Write` or `Read`.

5.2.5 RQ3: Type Error Reduction

To evaluate the efficacy of the proposed approach in reducing type errors, we compare settings where function augmentation with callee signatures and error fixes are selectively enabled or disabled. We denote the disabled features using superscripts: -a indicates the lack of function augmentation, and -f indicates the absence of error fixes. Thus, the settings are Tymcrat$_n$, Tymcrat$_n^{\text{-f}}$, Tymcrat$_n^{\text{-a}}$, and Tymcrat$_n^{\text{-fa}}$, where n varies from 0 to 4. Tymcrat$_n^{\text{-fa}}$ serves as the baseline by not employing any error-reducing techniques.



(a) GPT-3.5 Turbo



(b) GPT-4o mini

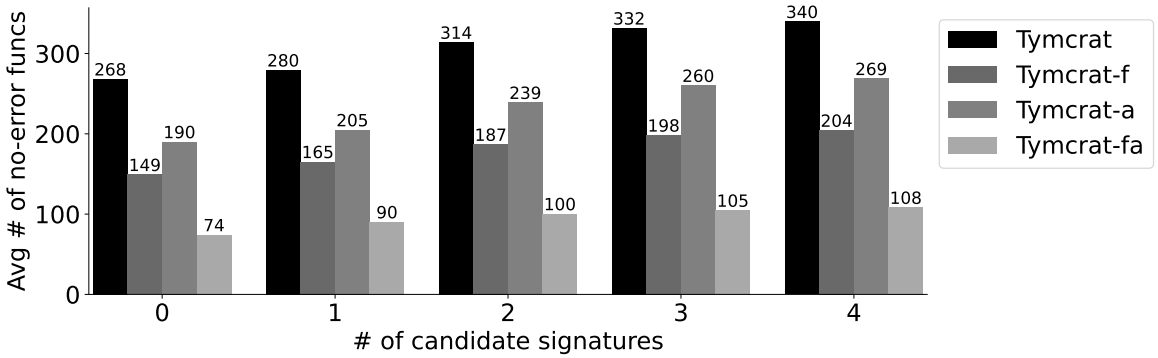
Figure 5.6: Average number of type errors after translation

Figure 5.6 shows the average number of type errors in benchmark programs translated by each setting. Since the results show consistent trends across different n values, we explain only the case of $n = 4$. When using GPT-3.5 Turbo (Figure 5.6a), $\text{Tymcrat}_4^{\text{-f}}$ outperforms $\text{Tymcrat}_4^{\text{-fa}}$, resulting in a 29.6% reduction in type errors. This demonstrates that augmenting functions with callee signatures effectively reduces type errors. Additionally, $\text{Tymcrat}_4^{\text{-a}}$ outperforms $\text{Tymcrat}_4^{\text{-fa}}$, leading to a 53.2% reduction in type errors, showing that iteratively fixing errors is also an effective method for reducing type errors. Finally, comparing Tymcrat_4 to $\text{Tymcrat}_4^{\text{-fa}}$ reveals a collective impact of the two error-reducing techniques, with a 65.1% reduction in type errors.

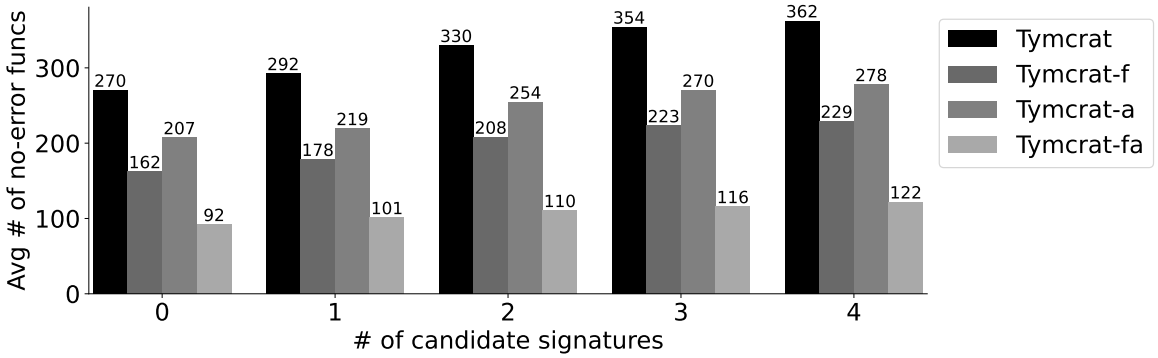
Similar results are observed when using GPT-4o mini (Figure 5.6b). Compared to $\text{Tymcrat}_4^{\text{-fa}}$, $\text{Tymcrat}_4^{\text{-f}}$ and $\text{Tymcrat}_4^{\text{-a}}$ reduce type errors by 34.9% and 58.7%, respectively. Collectively, Tymcrat_4 reduces type errors by 71.5%. With $\text{Tymcrat}_4^{\text{-fa}}$, GPT-4o mini generates 7.4% fewer type errors than GPT-3.5 Turbo, indicating its higher intelligence. Furthermore, GPT-4o mini reduces type errors more effectively than GPT-3.5 Turbo when functions are augmented and errors are iteratively fixed, suggesting its better capability of utilizing the provided additional information.

Another notable trend is that the number of type errors decreases as n increases. When using GPT-3.5 Turbo, Tymcrat_2 to Tymcrat_4 reduce type errors by 15.1%, 24.3%, and 27.5% compared to Tymcrat_1, respectively. When using GPT-4o mini, the reductions are 16.8%, 26.8%, and 32.0%. This is because generating more candidate signatures results in more diverse translations and increases the likelihood of obtaining a translation with fewer type errors.

Nevertheless, even with Tymcrat_4 using GPT-4o mini, each program exhibits an average of 1,155.3



(a) GPT-3.5 Turbo



(b) GPT-4o mini

Figure 5.7: Average number of functions without type errors after translation

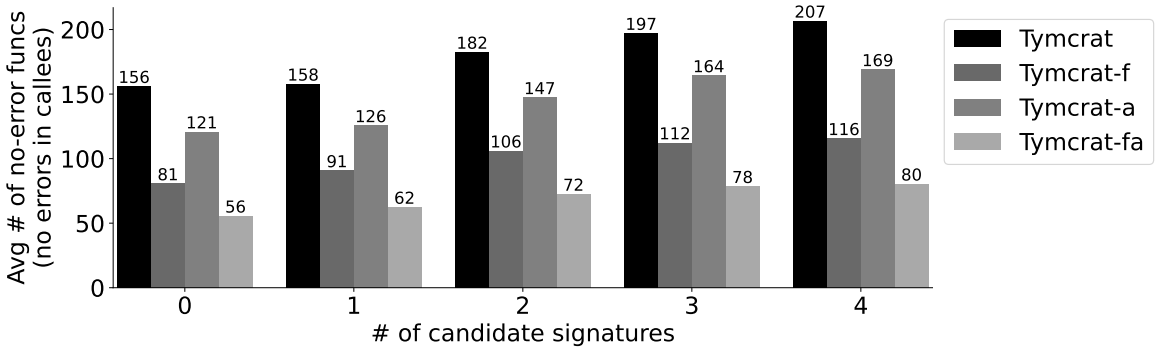
type errors, which is still a significant number. Further research is needed to reduce the number of type errors to a manageable level.

We further investigate the effectiveness of the proposed approach in reducing type errors by examining the number of functions without type errors. Figure 5.7 shows the average number of such functions in benchmark programs translated by each setting. The results demonstrate that our techniques significantly increase the number of functions without type errors by reducing type errors. When using GPT-3.5 Turbo (Figure 5.7a), Tymcrat$_4^{-a}$ and Tymcrat$_4^{-f}$ generate 147.9% and 88.5% more functions without type errors than Tymcrat$_4^{-fa}$, respectively. Collectively, Tymcrat$_4$ increases the number of such functions by 213.5%. When using GPT-4o mini (Figure 5.7b), Tymcrat$_4^{-a}$, Tymcrat$_4^{-f}$, and Tymcrat$_4$ increase the number of functions without type errors by 127.9%, 88.0%, and 197.4%, respectively. However, even with Tymcrat$_4$ using GPT-4o mini, only 55.9% of functions have no type errors, indicating that further research is needed to enhance this number.

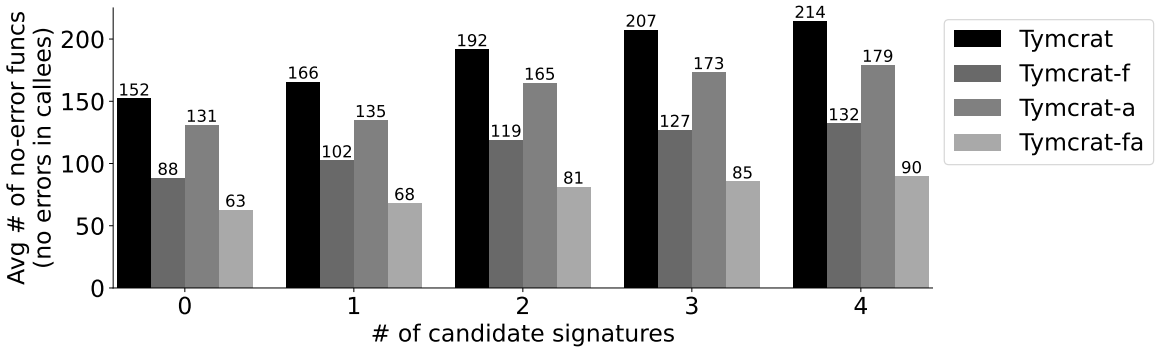
We also count the number of functions that have no type errors in themselves and their callees. By definition, these numbers are always less than or equal to the number of functions without type errors. Figure 5.8 shows the average number of such functions in benchmark programs translated by each setting. The overall trend is similar to that in Figure 5.7.

While the goal of this work is to reduce type errors in the translated code, the absence of type errors does not necessarily imply the correctness of the translation. The translated program can still exhibit different behavior from the original, even when no type errors exist.

Unfortunately, automatically verifying the correctness of the translation is challenging. The most



(a) GPT-3.5 Turbo



(b) GPT-4o mini

Figure 5.8: Average number of functions that do not have type errors in themselves and their callees after translation

common method in practice is to run test suites on the translated program. However, this cannot be applied in our evaluation because each translated program does not compile due to type errors and thus cannot be executed. Although functions without type errors can be compiled separately, the benchmark programs have test suites only for the entire programs and do not provide unit tests for individual functions.

For this reason, we conducted case studies to manually investigate the correctness of the translation, involving 41 functions. For each benchmark program, we randomly selected a function with no type errors after translation by Tymcrat_4 both using GPT-3.5 Turbo and GPT-4o mini. This sample size provides a 7% margin of error with a 95% confidence level. The investigation was conducted independently by the authors and another researcher familiar with Rust.

Our case studies show that some functions are semantically incorrect despite the absence of type errors. We found that 23 functions (56.1%) were correctly translated by GPT-3.5 Turbo, and 32 functions (78.0%) were correctly translated by GPT-4o mini. In contrast, another researcher reported that 30 functions were correctly translated by GPT-3.5 Turbo and 35 by GPT-4o mini. This discrepancy stems from our use of more conservative criteria to determine correctness. For instance, the other researcher deemed translations that replace abort-on-error with logic that returns error-indicating values as correct, whereas we considered these incorrect. Despite these differences, the case studies consistently suggest that the proposed approach sometimes produces semantically incorrect translations. Using our criteria, 18 functions were translated correctly by both, 5 were translated correctly only by GPT-3.5 Turbo,

14 were translated correctly only by GPT-4o mini, and 4 were translated incorrectly by both. While GPT-4o mini demonstrates a better ability to preserve semantics during translation compared to GPT-3.5 Turbo, the ratio of correct translations is still not satisfactory. An important future direction is to develop techniques to automatically verify the correctness of translations and properly fix incorrect translations.

Summary of RQ3: Type Error Reduction

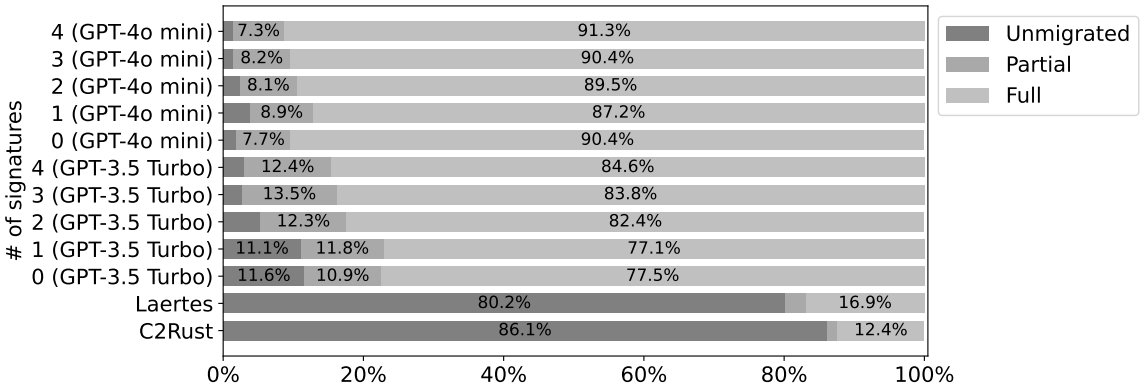
- Both function augmentation with callee signatures and iterative error fixes effectively reduce the number of type errors in the translated code and increase the number of functions without type errors.
- Generating more candidate signatures leads to fewer type errors by increasing the diversity of translations.
- Despite the proposed techniques, translated programs still contain a significant number of type errors, and further research is needed to reduce this number further.
- A function may not preserve the original behavior even when it has no type errors, highlighting the need for techniques to verify the correctness of translations.

5.2.6 RQ4: Comparison with Existing Approaches

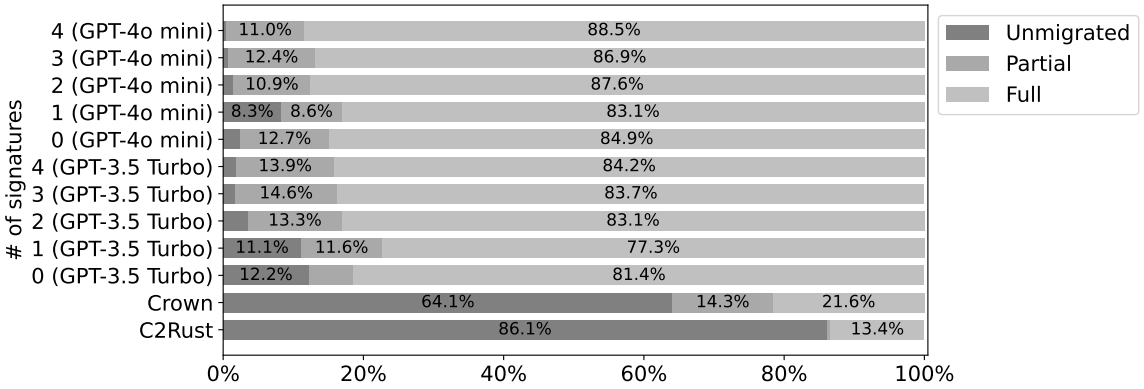
We compare the translation of the proposed approach with that of existing C-to-Rust translators: C2Rust, Laertes, and Crown. The goal of the comparison is not to claim the superiority of the proposed approach but to understand the characteristics of different approaches. While our approach uses LLMs, existing tools translate code using syntactic rules and static analysis. These two directions have their own advantages and disadvantages and can be used in a complementary manner. We believe that future work is required to unify these directions and leverage the best of both worlds.

We first compare the proposed approach with C2Rust and Laertes. As Laertes works by transforming Rust code generated by C2Rust, we do not perform a separate comparison with C2Rust. Unfortunately, Laertes can translate only 1 out of the 41 benchmark programs. While Laertes is capable of transforming compilable Rust code, C2Rust fails to produce compilable code for 32 programs. Furthermore, during code transformation, Laertes crashes in 8 out of the remaining 9 programs. To augment our evaluation, we incorporate an additional set of 14 C programs used in the Laertes paper [83], resulting in a total of 15 programs.

Figure 5.9a depicts the proportions of different kinds of signatures in the 15 programs translated by C2Rust, Laertes, or Tymcrat$_0$ to Tymcrat$_4$ using GPT-3.5 Turbo or GPT-4o mini. Our approach provides significantly better type migration capabilities compared to C2Rust and Laertes, regardless of the number of candidates. C2Rust introduces only never and `Option` types and fully migrates 12.4% of the signatures. Laertes replaces C pointers with Rust pointers in the given Rust code by introducing reference, slice, `Box`, and `Option` types. This increases the portion of fully migrated signatures to 16.9%. On the other hand, Tymcrat$_4$ using GPT-4o mini fully migrates 91.3% of the signatures. However, Laertes outperforms our approach in avoiding type errors and preserving semantics. Laertes introduces no type errors during translation. It always preserves semantics as it only replaces C pointers with Rust pointers. In contrast, our approach introduces an average of 597 type errors per program across 15 programs, even when using Tymcrat$_4$ with GPT-4o mini, and may not preserve semantics, as shown in



(a) Comparison with Laertes



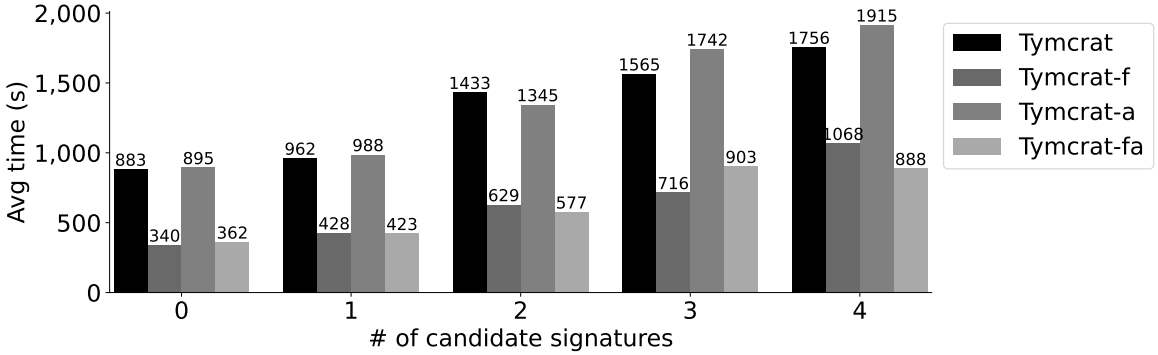
(b) Comparison with Crown

Figure 5.9: Proportions of unmigrated, partially migrated, and fully migrated function signatures after translation with C2Rust, Laertes, Crown, or Tymcrat

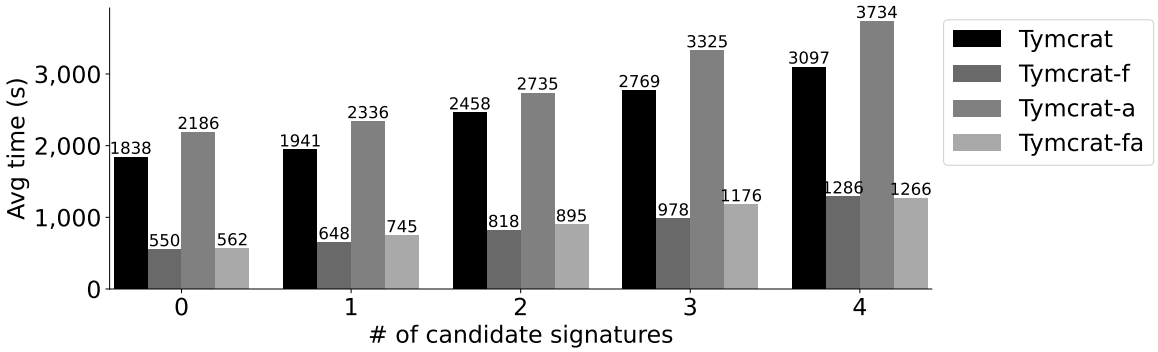
RQ3.

We then compare the proposed approach with Crown. Unfortunately, Crown is incapable of handling some C features, including variadic functions and unions, and crashes when the target code contains them. Since these features are frequently used in GNU projects, Crown cannot translate any of the 41 benchmark programs. To facilitate the comparison, we instead use 20 programs utilized in the Crown paper [203].

Figure 5.9b shows the proportions of different kinds of signatures in the 20 programs translated by C2Rust, Crown, or Tymcrat$_0$ to Tymcrat$_4$ using GPT-3.5 Turbo or GPT-4o mini. Our approach migrates significantly more types compared to Crown. C2Rust fully migrates 13.4% of the signatures, while Crown introduces reference, `Box`, and `Option` types, achieving 21.6% of the signatures being fully migrated. In contrast, Tymcrat$_4$ using GPT-4o mini fully migrates 88.5% of the signatures. However, Crown does not introduce type errors during the type migration process and always preserves semantics. This highlights the benefits of Crown, as our approach introduces an average of 148 type errors per program across 20 programs, even when using Tymcrat$_4$ with GPT-4o mini, and may not preserve semantics.



(a) GPT-3.5 Turbo



(b) GPT-4o mini

Figure 5.10: Average time taken to translate each GNU package

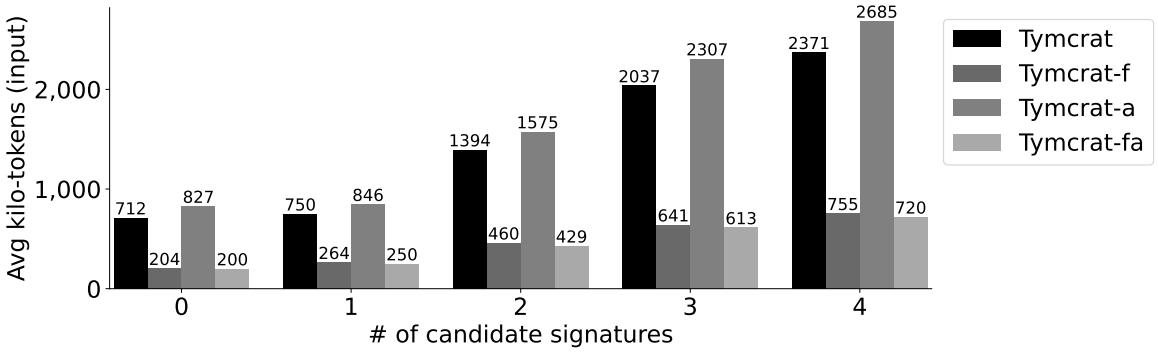
Summary of RQ4: Comparison with Existing Approaches

- The proposed approach migrates more signatures than existing approaches, including C2Rust, Laertes, and Crown.
- Laertes and Crown do not introduce type errors during translation, while the proposed approach may introduce them.
- Future research should explore the possibility of combining LLM-based translation with static analysis-based translation to effectively migrate types while avoiding type errors.

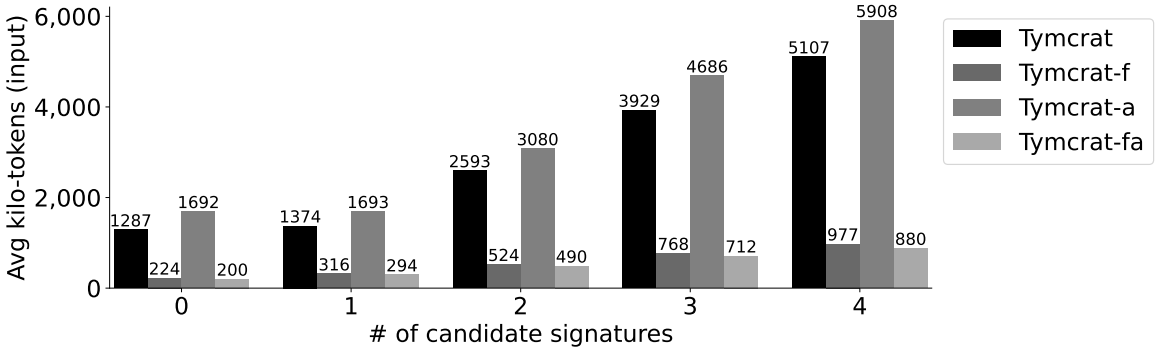
5.2.7 RQ5: Overhead

To evaluate the performance overhead of the proposed approach, we investigate the following settings: Tymcrat$_n$, Tymcrat$_n^{\text{-f}}$, Tymcrat$_n^{\text{-a}}$, and Tymcrat$_n^{\text{-fa}}$, with n ranging from 0 to 4. Figure 5.10 displays the average of the translation times of the GNU packages in each setting.

The results reveal that the proposed approach introduces a reasonable additional time compared to the baseline. First, we analyze the results from GPT-3.5 Turbo (Figure 5.10a). Comparing Tymcrat$_0$ and Tymcrat$_4$, we observe a 98.8% increase in translation time due to generating four candidate signatures per function. The overhead arises not only from generating candidate signatures but also from translating a single function multiple times with different signatures. Although these translations are attempted in parallel, the maximum translation time among them is likely to exceed the time required for a single



(a) GPT-3.5 Turbo



(b) GPT-4o mini

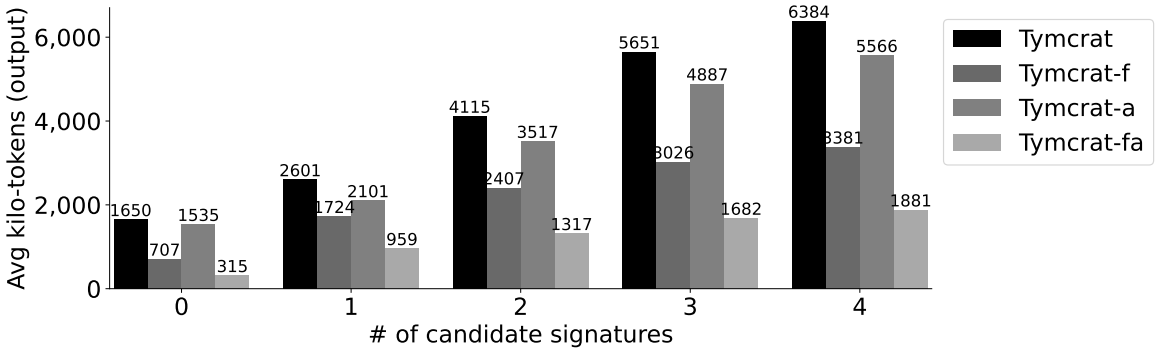
Figure 5.11: Average number of input tokens required to translate each GNU package

translation attempt. Additionally, comparing Tymcrat$_4^{\text{-fa}}$ and Tymcrat$_4^{\text{-a}}$, we find that the iterative fix process increases translation time by 115.6%. This process inherently consumes significant time as it involves a series of sequential LLM invocations. Conversely, Tymcrat$_n^{\text{-fa}}$ and Tymcrat$_n^{\text{-f}}$ exhibit similar translation times for any n because providing callee signatures does not prolong the translation process. Moreover, Tymcrat$_n$ is faster than Tymcrat$_n^{\text{-a}}$, with an 8.3% speedup when $n = 4$. This is because function augmentation mitigates type errors in the initial translation, often reducing the iterations needed to fix them.

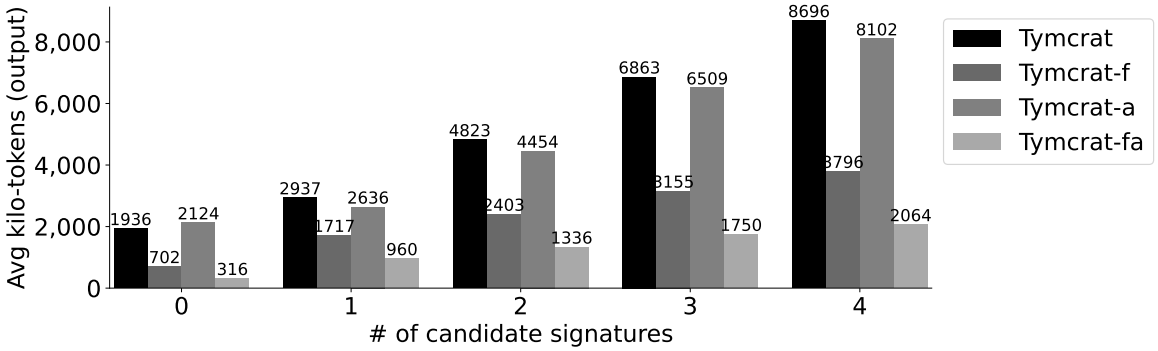
When using GPT-4o mini (Figure 5.10b), the overall trend is similar to that with GPT-3.5 Turbo. Generating four candidate signatures per function increases translation time by 68.5%, and iterative error fixes increase it by 194.9%. Tymcrat$_n^{\text{-fa}}$ and Tymcrat$_n^{\text{-f}}$ provide similar translation times, and Tymcrat$_4$ is faster than Tymcrat$_4^{\text{-a}}$ by 17.1%.

We also report important statistics to further understand the runtime of the proposed approach. When using Tymcrat$_4$ with GPT-3.5 Turbo, the minimum and maximum translation times are 150 seconds for `time-1.9` and 5,474 seconds for `gawk-5.2.2`, with a standard deviation of 1,446 seconds. When using Tymcrat$_4$ with GPT-4o mini, the minimum and maximum times are 287 seconds for `time-1.9` and 9,900 seconds for `nano-7.2`, with a standard deviation of 2,349 seconds. These statistics indicate that the proposed approach translates programs within a reasonable time frame, considering that the translation process is performed only once for each program. Additionally, the high standard deviation values suggest that translation time varies significantly depending on the size of the program.

We now investigate the overhead in terms of tokens used by the LLM. Figure 5.11 shows the average



(a) GPT-3.5 Turbo



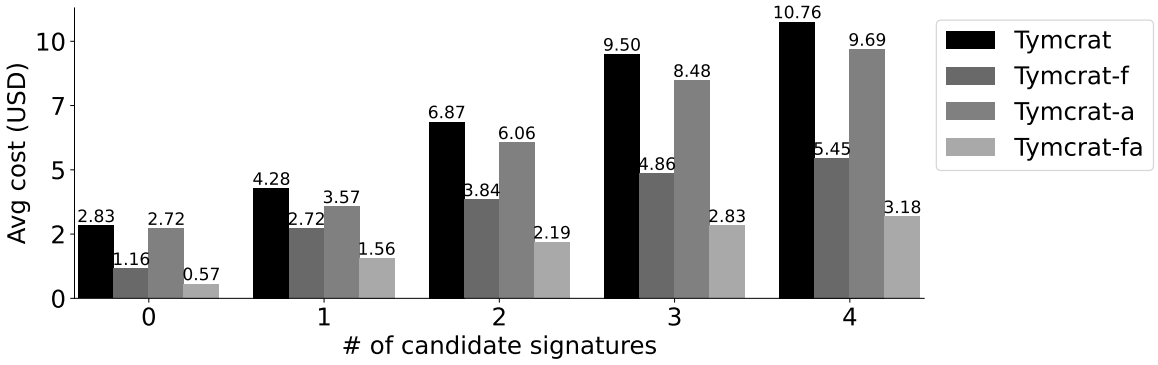
(b) GPT-4o mini

Figure 5.12: Average number of output tokens required to translate each GNU package

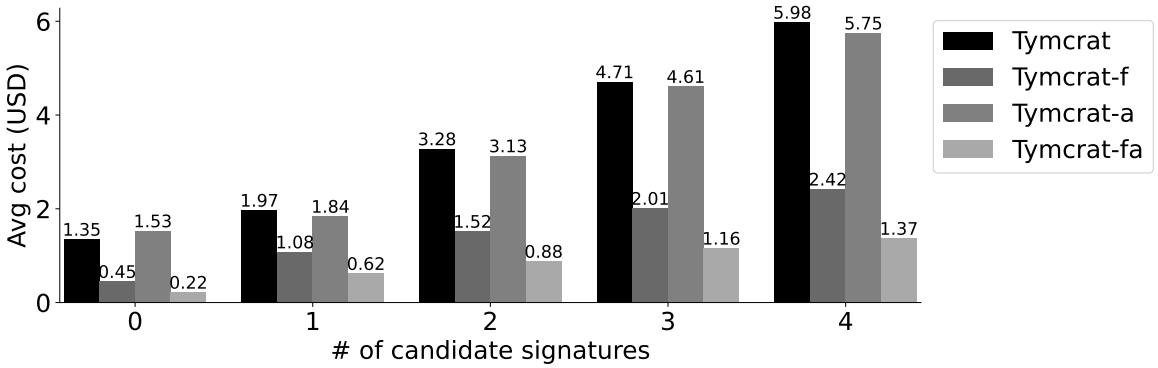
number of input tokens required to translate the benchmark programs. The trend is consistent with that of the translation time. When using GPT-3.5 Turbo (Figure 5.11a), Tymcrat_4 increases input tokens by 233.2% compared to Tymcrat_0. The increase in token usage is significantly higher than the increase in time, as token usage does not benefit from parallelism. Additionally, error fixes incur a 273.2% increase when $n = 4$. When using GPT-4o mini (Figure 5.11b), candidate signature generation and error fixes increase input tokens by 296.7% and 571.1%, respectively.

Figure 5.12 shows the average number of output tokens required to translate the benchmark programs. While the trend is similar, the number of output tokens is higher than the number of input tokens due to the longer length of the translated Rust code compared to the original C code. When using GPT-3.5 Turbo (Figure 5.12a), candidate signature generation and error fixes increase the number of output tokens by 287.0% and 195.9%, respectively. When using GPT-4o mini (Figure 5.12b), the increases are 349.1% and 292.5%, respectively.

Finally, we investigate the monetary cost of the proposed approach. The OpenAI API charges different prices for input and output tokens. The cost for gpt-3.5-turbo-0125 is \$0.50 per 1M input tokens and \$1.50 per 1M output tokens, while the cost for gpt-4o-mini-2024-07-18 is \$0.15 per 1M input tokens and \$0.60 per 1M output tokens. Figure 5.13 shows the average cost to translate the benchmark programs. While the cost increases with the number of tokens used, it remains reasonable, averaging \$10.76 even when using Tymcrat_4 with GPT-3.5 Turbo, which is the most expensive configuration.



(a) GPT-3.5 Turbo



(b) GPT-4o mini

Figure 5.13: Average cost required to translate each GNU packages

Summary of RQ5: Overhead

- Generating candidate signatures increases both translation time and token usage, with the increase being higher for token usage because it does not benefit from parallelism.
- Error fixes significantly increase translation time and token usage, but this overhead remains reasonable.
- Function augmentation does not exhibit a meaningful overhead in terms of time and tokens and even improves performance when combined with error fixes by reducing the iterations needed to fix type errors.
- The monetary cost of the proposed approach is reasonable, averaging $10.76 even when using the most expensive configuration.

5.2.8 Threats to Validity

The threats to internal validity relate to the nondeterministic nature of the approach. Despite the temperature of the API set to 0, the LLM can still exhibit nondeterministic behavior, meaning Tymcrat may yield different translation results for the same program. Additionally, since Tymcrat utilizes parallelism, measured times can be affected by thread scheduling and may vary across runs. To reduce these threats, future work could conduct more experiments and average the results.

The threats to external validity concern the selection of C programs. We collected C programs from GNU Package Blurbs, and GNU packages may share similar coding styles and idioms. Programs from other sources might exhibit different characteristics. Furthermore, filtering out programs without individual Wikipedia entries might exclude newer, lesser-known packages that potentially use modern C programming practices. Also, excluding programs with over 100,000 lines of code may omit complex patterns found in large-scale software. These limitations restrict the diversity of the C programs used in the evaluation and may impact the generalizability of the results. To mitigate these threats, future work could include additional experiments with C programs from various sources and of different sizes.

The threats to construct validity primarily concern the assessment of type migration quality. Merely introducing more diverse Rust types more frequently does not necessarily imply better quality. The idiomatic use of types involves various factors, such as the intention of the original C code and the meaning of the types. We mitigated this threat by conducting case studies, but they do not cover the entire translated codebase. Additionally, they are subject to our interpretation of the code and understanding of Rust idioms. To mitigate these threats, future work could conduct more case studies and involve more researchers to assess the quality of the type migration.

Another threat to construct validity is measuring the reduction of type errors as a proxy for the effectiveness of the translation. Fewer type errors do not necessarily indicate a superior translation. In some cases, code with fewer type errors may require more code changes to compile successfully. Notably, the Rust compiler performs two-phase checking, consisting of type checking and borrow checking, terminating after the first phase if any type errors are encountered. This means that fixing type errors may reveal errors in borrow checking, requiring additional changes. A possible solution to this threat is to consider not only the number of type errors but also the phase in which they occur in both Algorithm 5.2 and evaluation. Moreover, even when the translated code has no type errors, it may not preserve the original semantics. We addressed this threat by conducting case studies, although these do not encompass the entire translated codebase. To enhance the reliability of our approach, future work could complement the current metric, e.g., by generating unit tests.

Chapter 6. Modular Abstractions for Unsafe Features

In OSs, *shared mutable states* are necessary for low-level control and efficiency. For example, for low-level control of hardware, OSs need to mutate arbitrary shared memory locations; and for efficiency, they typically use data structures with complex sharing in the form of pointer aliasing. To ensure the safety of these shared mutable states, type checking should be precise enough to prove the absence of invalid memory accesses and data races.

Rust supports shared mutable states through a twofold design: *Safe Rust* guaranteeing type safety and *Unsafe Rust* with user-defined modular abstractions.

On the one hand, Safe Rust, the default mode of Rust, enforces the *aliasing XOR mutability* (A⊕M) discipline for shared mutable states, stating that every state is either aliased (shared) or mutated, but not both at the same time. By enforcing this discipline, Rust type checker can prove that a program written in Safe Rust with shared mutable states is *memory-safe*, i.e., no memory bugs, and even *thread-safe*, i.e., no data races.

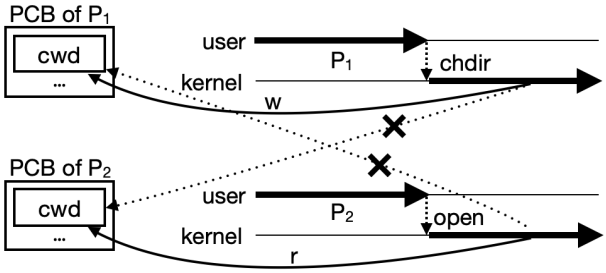
On the other hand, Unsafe Rust is an escape hatch from the restrictiveness of Safe Rust. The A⊕M discipline misses the *aliased AND mutable* (A&M) class of shared mutable states, heavily used in system programs. To use A&M states in certain regions of code, programmers elude the A⊕M discipline by locally enabling Unsafe Rust. Since Unsafe Rust does not guarantee safety, programmers must validate safety themselves, which is a big burden. Therefore, Rust programmers have "tamed" A&M states by (1) identifying groups of A&M states with the same reasoning principle, as we call *A&M patterns*, which facilitate design reuse; and (2) hiding their use of Unsafe Rust behind modular abstractions, which facilitate modular reasoning [55]. Once A&M states are encapsulated inside modular abstractions, one can informally reason about their safety or even apply formal verification methods [115].

Unfortunately, modular abstractions for A&M patterns found in legacy OSs, such as Linux, are lacking. Since A&M patterns have been studied mostly in the context of general system software, modular abstractions for A&M patterns appearing especially in legacy OSs have not been explored yet. Clean-slate Rust OSs [131, 127, 128, 67, 156] do not tackle this problem and instead replace such A&M patterns with well-known A&M patterns whose modular abstractions already exist. However, as we show in Section 6.1, replacing a certain A&M pattern with another A&M pattern undesirably degrades performance. Considering that people expect retrofitting Rust onto legacy kernel development will reduce memory bugs without sacrificing the performance, such replacement is unsatisfactory. Modular abstractions for legacy OSs' A&M patterns are necessary to acquire safety while retaining performance.

In this chapter, we propose modular abstractions of crucial A&M patterns in OSs. Specifically, our contributions are as follows:

- We show the impact of the choice of A&M patterns on the performance of OSs with an experiment. We compare the performance of an OS with a modified version of the OS that replaces a certain A&M pattern with another A&M pattern. The result of the experiment indicates that the replacement of the A&M pattern noticeably degrades the performance (Section 6.1);

- We present a general guideline on choosing an implementation strategy for A&M states in Rust. Rust provides two implementation strategies for A&M states, raw pointers and interior mutability. Since there has been no guideline to choose between them, programmers have chosen one of them



where w for write, r for read, and × for inaccessible

Figure 6.1: Process-owned value: `cwd` in `Proc`

for each A&M pattern in an unsystematic manner. We propose a simple guideline to address this issue (Section 6.2);

- We discover six A&M patterns in OSs and design their modular abstractions by rewriting the xv6 OS [74] entirely in Rust as open-source xv6$_{Rust}$ [103]. We claim that our discovery of patterns is complete with respect to xv6$_{Rust}$ as we systematically find them with a dependency graph among types. The six patterns are as follows: process-owned values, CPU-owned values, memory pools, lock-protected immovable values, lock-protected separated values, and asynchronous ownership transfer for I/O (Section 6.3);

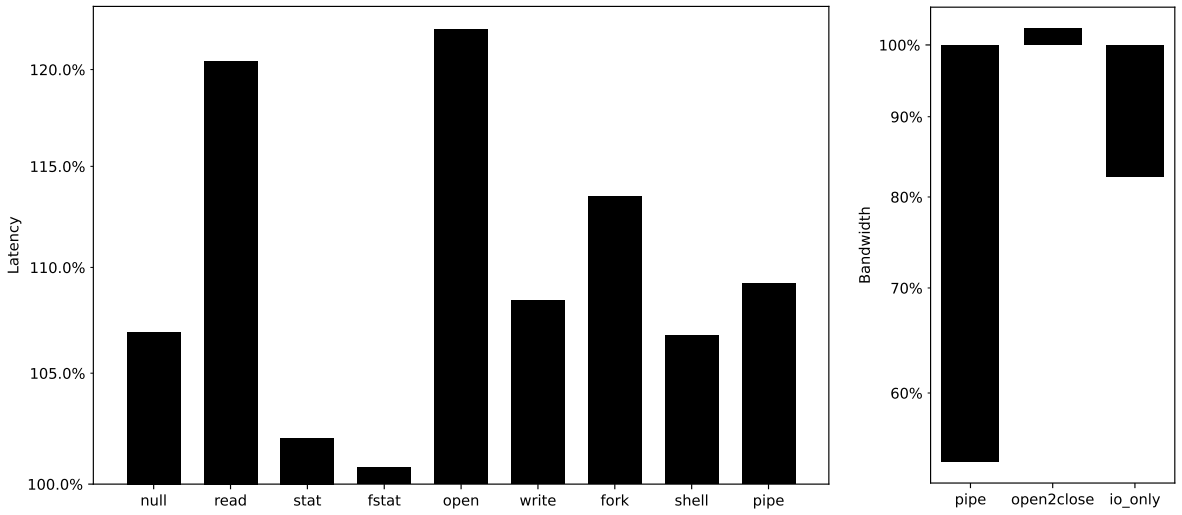
- We evaluate our modular abstractions. We show that the modular abstractions are *practical* in rewriting legacy OSs in Rust, as all the six A&M patterns are utilized in Linux, a representative legacy OS. In addition, the modular abstractions are *original*, as none of them are found in existing Rust OSs. We also show that the modular abstractions reduce the safety reasoning cost of xv6$_{Rust}$ to the level of the clean-slate Rust OSs by measuring the amount of Unsafe Rust used in each OS. At the same time, the modular abstractions incur no run-time overhead. On average, xv6$_{Rust}$ performs 31.1% faster than xv6 for benchmark programs (Section 6.4).

6.1 Motivation

This section illustrates a concrete example giving a motivation to retain A&M patterns in legacy OSs while rewriting them in Rust. We reveal by experiment that the choice of A&M patterns in OSs significantly impacts the performance and show that A&M patterns in legacy OSs must remain the same to preserve the performance.

To show the performance gap between different A&M patterns, we compare two A&M patterns: *lock-protected values* and *process-owned values*. A lock-protected value is shared among multiple threads and uses a lock for synchronization. A process-owned value does not use locks despite being shared among threads because only a thread handling a certain process's system call accesses it. This property guarantees exclusive access, so there is no need for a synchronization scheme.

In xv6, where each process consists of a single thread, process-owned values reside in process control blocks (PCBs). We represent a PCB with the type `Proc`. `Proc`'s `cwd`, an in-memory inode for a process's working directory, is an example of a process-owned value in `Proc`. In Figure 6.1, P_1 invokes a `chdir` system call to change its working directory. The thread changes the inode in the `Proc` to a new inode. At the same time, P_2 opens a file with a relative path, and the thread resolves the path with respect to



(a) Latency

(b) Bandwidth

Figure 6.2: Performance of xv6^{Lock}_{Rust} compared to xv6_{Rust}

the working directory. As this example shows, each process reads or changes its own working directory to handle system calls, but it does not need to access the working directories of the other processes. Therefore, `cwd` is a process-owned value.

To compare the performance of process-owned values with that of lock-protected values, we implement xv6_{Rust} and xv6^{Lock}_{Rust}, two different rewrites of the xv6 OS in Rust. The former keeps all the process-owned values in `Proc` by using a novel modular abstraction we propose in Section 6.3.1. Since values of this kind occur only in OSs, the Rust community has not made any effort to design a modular abstraction. The latter replaces all the process-owned values with lock-protected values whose modular abstraction, `Mutex`, is provided by the Rust standard library. Therefore, accessing `cwd` requires a thread to acquire a lock, which is unnecessary in fact. This change incurs additional overhead at run time.

Our experiment shows that replacing process-owned values with lock-protected values harms the performance. The experiment measures the latency and bandwidth of xv6_{Rust} and xv6^{Lock}_{Rust}. We run benchmark programs in LMBench [147] with the parallelism parameter set to 8. The environment for the experiment is the same as that of Section 6.4.3. Figure 6.2 shows the result of the comparison. Compared to xv6_{Rust}, xv6^{Lock}_{Rust} performs worse by 9.80% and 22.93% in terms of latency and bandwidth, respectively.

Despite the significant impact of the choice of A&M patterns on performance, modular abstractions for A&M patterns in legacy OSs are missing. Clean-slate Rust OSs [131, 128, 67, 156, 10] prefer well-known A&M patterns with modular abstractions provided by existing libraries over A&M patterns in legacy OSs whose modular abstraction is unknown. For example, they replace process-owned values, whose modular abstraction is unknown, with another well-known A&M pattern, lock-protected values.

Considering the current status, we believe that it is important to design modular abstactions for A&M patterns in legacy OSs, which will help OS developers preserve the performance of legacy OSs when rewriting them in Rust.

6.2 Background

This section reviews Safe Rust's A⊕M discipline (Section 6.2.1) and Unsafe Rust's A&M support (Section 6.2.2) with two example A&M patterns in Rust (Section 6.2.3). It also proposes a guideline on choosing unsafe operations to implement A&M patterns (Section 6.2.4).

6.2.1 Safe Rust's A⊕M Discipline

Rust enforces the A⊕M discipline by leveraging an *ownership and borrowing* type system.

Ownership In Rust, a value is owned by a unique variable called an *owner*. One can change a value's owner by assigning it to another variable or passing it as a function argument:

```
1 fn main() {
2     let x = vec![0, 1, 2];
3     // can use `x`
4     foo(x);
5     // cannot use `x`
6 }
7 fn foo(v: Vec<i32>) { ... }
```

Here, `x` owns a heap-allocated integer array. After line 2, the function `main` can access the array with `x`. Because passing `x` to `foo` transfers the ownership of the array to `v`, `x` is no longer its owner and `main` cannot access it after line 4. Ownership not only ensures safety but also removes the burden of manual destruction on programmers without resorting to garbage collection. When an owner goes out of scope without passing the ownership to another variable, the compiler automatically inserts a destructor invocation.

Mutable Reference Rust allows temporary acquisition of ownership via *borrowing* instead of complete transfer:

```
1 fn main() {
2     let mut x = vec![0, 1, 2];
3     let r = &mut x;
4     // cannot use `x`
5     foo(r);
6     // can use `x`
7 }
8 fn foo(v: &mut Vec<i32>) { ... }
```

On line 3, `&mut x` indicates that the ownership is temporarily borrowed from `x` to create a *mutable reference*. Like an owner, a mutable reference guarantees unique accesses so that `main` cannot use `x` after line 3 till line 5. Now `foo` takes a mutable reference and can freely mutate the array as if it owns it. When `foo` returns, the borrowing finishes because `foo` does not return `r`, preventing its further use [24]. This makes `main` able to use `x` again.

Shared Reference While owners and mutable references allow mutation, *shared reference*s support aliasing:

```
1 fn main() {
2     let x = vec![0, 1, 2];
3     let r1 = &x; let r2 = &x;
4     foo(r1);
5     // cannot create `&mut x`
6     println!("{}", r2[0]);
7 }
8 fn foo(v: &Vec<i32>) { ... }
```

Line 3 creates two shared references to `x` via `&x`. Rust supports aliasing by creating multiple shared references to the same location. Although `foo` takes `r1`, `main` still can access the array with `r2`. Shared references are read-only, e.g., `foo` can read data in `v` but not mutate.

Lifetime Since a reference gains ownership temporarily, each reference has a *lifetime* of how long it retains ownership. The type of a reference that lives for a lifetime `'a` is `&'a mut T` or `&'a T`. Assume that the following function takes a reference to an array and returns a reference to its first element:

```
fn head<'a>(v: &'a Vec<i32>) -> &'a i32 {...}
```

It is polymorphic over a lifetime parameter `'a`. The input array's and output element's lifetimes are the same because an element cannot outlive its array. Since the compiler infers some omitted lifetimes based on predefined rules [21], the previous `head` can be simplified as follows:

```
fn head(v: &Vec<i32>) -> &i32 { ... }
```

6.2.2 Unsafe Rust's A&M Support

Rust's ownership and borrowing type system ensures strong type safety, but Safe Rust restricts the scope of the safety guarantee to A⊕M states. A&M states require unsafe operations of Rust to relax type checking at the cost of forgoing the safety guarantee. A primary example of unsafe operations is dereference of raw pointers as in C: it allows concurrent accesses from multiple threads while the compiler does not guarantee its thread safety.

Safety of Unsafe Rust Because the compiler does not guarantee the safety of unsafe operations, programmers should validate it. They must avoid *undefined behaviors* (UBs) [20] by inspecting every unsafe operation. For example, programmers must ensure that (1) every dereference of raw pointers is neither invalid memory access nor data race; and (2) every reference points to a valid object and follows A⊕M.

To facilitate manual inspection of unsafe operations, Rust requires them to be enclosed within unsafe blocks:

```
1 let mut x = vec![0, 1, 2];
2 let p = &raw mut x;
3 unsafe { (*p)[0] = 1; }
4 // *can* use `x`
```

Line 2 creates a mutable raw pointer to `x`, and line 3 dereferences it, which is unsafe, enclosed by an unsafe block. We call code in unsafe blocks *unsafe code*.

A&M Pattern and Modular Abstraction Even with unsafe blocks, validating A&M states within unsafe code is challenging primarily due to its intrusive nature: a single buggy use of an unsafe operation may corrupt the entire program, and errors may manifest syntactically far from the unsafe operation.

To address this, programmers have (1) identified groups of A&M states that share the same reasoning principle as A&M patterns; and (2) encapsulated these patterns within Rust types as modular abstractions [55]. For example, the *lock-protected values* pattern includes A&M states that use locks to synchronize simultaneous accesses by multiple threads. All of them rely on the same principle: only a single thread can hold a lock at a time. This pattern is encapsulated in `Mutex<T>`.

Modular abstractions confine the consequences of unsafe operations within types, facilitating modular reasoning about safety [56, 55]. Ideally, each method's implementation with unsafe code should be validated within its type so that its use sites can freely invoke it without unsafe blocks. Such modularity significantly reduces the safety reasoning cost.

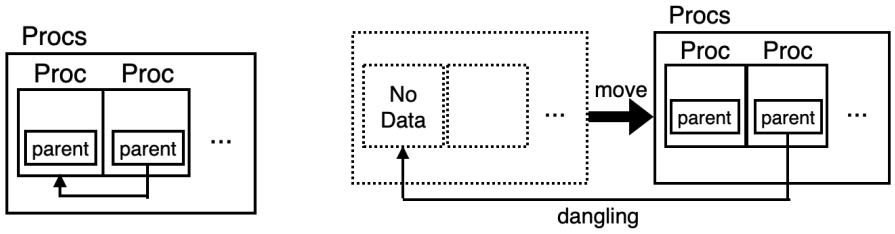
However, types with completely safe APIs cannot represent some A&M patterns. Such types provide a few *unsafe methods*, marked `unsafe`, that are safe only when their callers satisfy certain method-specific conditions described by the type's implementor. Every call to an unsafe method must be within an unsafe block. For example, `Rc` [36], a type for the *reference counting* pattern, provides the methods `get_mut` and `get_mut_unchecked`, each of which returns a mutable reference to the reference-counted data. While `get_mut` succeeds only when the reference count is one, `get_mut_unchecked` always succeeds regardless of the reference count. Therefore, a caller of `get_mut_unchecked` must ensure that no other threads are accessing the same reference-counted data, rendering `get_mut_unchecked` an unsafe method. Types with unsafe methods still reduce reasoning costs because most of the reasoning is confined within the types.

Unsafe Operations for A&M Patterns To support A&M patterns, Unsafe Rust provides two classes of unsafe operations: *raw pointers* and *interior mutability*.

Raw pointers in Rust are similar to pointers in C. The type of a raw pointer to a value of type `T` is `*mut T`. Unlike references, raw pointers are not tracked by the ownership and borrowing type system: raw pointers lack lifetime information and do not borrow ownership. Consequently, one can create raw pointers regardless of the existence of other pointers, and these raw pointers can persist even after the deallocation of their referent. Therefore, dereferencing raw pointers is an unsafe operation. One can support A&M patterns by creating multiple raw pointers to the same value.

On the other hand, interior mutability is a mechanism to circumvent the immutability of shared references. Recall that shared references permit aliasing but are immutable by default, enabling compiler optimizations based on the assumption that shared references are not mutated. A value of the `UnsafeCell<T>` type—the sole source of interior mutability—contains a value of type `T` and provides an unsafe operation to mutably dereference the interior value from `&UnsafeCell<T>`. For compiler correctness, optimizations do not presume that interior values are immutable even in the presence of `&UnsafeCell<T>`. One can support A&M patterns by placing a value in an `UnsafeCell` and creating multiple shared references to it.

For modular abstractions of A&M patterns, we propose when to use each class: raw pointers if the lifetimes of pointers are dynamically determined, and interior mutability otherwise. For more detail, see Section 6.2.4.



(a) Structure of `Procs`

(b) Moving `Procs` makes pointers dangling

Figure 6.3: Self-referential value example: process manager

```
struct Proc { parent: UnsafeCell<*mut Proc>, ... }
struct Procs { process_pool: [Proc; NPROC], ... }
impl Procs {
    fn init(self: Pin<&mut Self>) { ... }
}
```

Listing 6.1: Modular abstraction of self-referential value

6.2.3 A&M Pattern Examples

Now, we review two well-known A&M patterns: self-referential values implemented with raw pointers and lock-protected values with interior mutability. We describe each pattern and its modular abstraction.

Self-Referential Values

Pattern Self-referential values contain pointers to their inner data and are utilized for various purposes in OSs. For example, Figure 6.3a depicts xv6's process manager, referred to as `Procs`, containing an array of `Proc` values. Each `Proc` comprises a process control block (PCB) that describes the process, specifically including a pointer to its parent process. This pointer is utilized, e.g., when a process terminates, to wake up its parent because it may be sleeping in a `wait` system call.

A mutable self-referential value is A&M because it is always aliased: its self-referenced field is accessible both through direct field access and via the self-referencing pointer. Additionally, the lifetime of a self-referencing pointer is dynamically determined, persisting until the enclosing value is destroyed. Consequently, self-referential values are typically implemented using raw pointers.

Furthermore, to ensure safety, self-referential values must be *immovable*, i.e., they must not change their memory addresses. Generally, a value must not move when there is a pointer referencing it. Moving to a new memory address renders the original address obsolete, yet the pointer does not automatically update, leading to dangling pointers. By definition, a self-referential value always includes a pointer to itself. Hence, it must not move. As illustrated in Figure 6.3b, moving a self-referential value makes its self-referencing pointers dangling.

Abstraction In Safe Rust, the compiler prevents the move of immovable values. Pointers in Safe Rust can only be references, and their existence is tracked by the compiler. When a reference to a value exists, the compiler prohibits any move of the value, ensuring that dangling pointers cannot be created in Safe Rust.

Unfortunately, in Unsafe Rust, immovable values may be inadvertently moved, particularly when

```
1 struct Mutex<T> { data: UnsafeCell<T>, lock: Lock }
2 struct Guard<'a, T> { ptr: &'a Mutex<T> }
3 impl<T> Mutex<T> {
4     fn lock(&self) -> Guard<'_, T> { ... }
5 }
6 impl<T> Guard<'_, T> {
7     fn deref_mut(&mut self) -> &mut T { ... }
8 }
```

Listing 6.2: Modular abstraction of lock-protected value

using raw pointers. Unlike references, the compiler does not track raw pointers, which means that a value can be moved even when there is a raw pointer pointing to it. Consequently, self-referential values, which are A&M and require the use of raw pointers, are not guaranteed by the compiler to be immovable.

To enforce immovability, we use the `Pin` type [28]. A *pinned reference* that refers to a value of type `T` has type `Pin<&mut T>` or `Pin<&T>`. Since it is just a pointer, its run-time representation is the same as a normal reference. A key invariant of a pinned reference is that its referent must not move. Thus, a pinned reference serves as evidence that its referent resides in the same location until it is removed.

To maintain this invariant, a pinned reference does not expose a mutable reference pointing to the value through a safe method. Note that mutable references are always eligible for moves: when variables `x` and `y` have type `&mut T`, one can swap the referents with `mem::replace(x, y)`, which is considered a safe operation. Instead, a pinned reference provides unsafe methods that convert it to a mutable reference and vice versa. To perform such conversions, one should ensure that the mutable reference is not used for moves.

Listing 6.1 shows that `Procs`'s method takes a pinned reference (line 4) to ensure that `Procs` values do not move.

Apart from `Pin`, developers have proposed other abstractions for self-referential values [4, 139, 143, 3]. They restrict the use of self-referencing pointers while keeping self-referencing values movable.

Lock-Protected Values

Pattern Locks are a classical synchronization scheme commonly used to protect a value shared among multiple threads. Only the thread holding the lock can read or mutate the value, thus preventing data races.

A lock-protected value is A&M because it is aliased among multiple threads, and a thread holding the lock can mutate it. A thread acquires a lock at the beginning of a lexical scope and releases it when the scope finishes. Since possible access to the value is statically determined, lock-protected values are implemented with interior mutability.

Abstraction Listing 6.2 shows that `Mutex` couples a lock with a lock-protected value [39] (line 1). `Mutex`es are shared among multiple threads as shared references. A thread can mutate the value in a `Mutex` when it acquires the lock, so the value must be in an `UnsafeCell`.

A `Guard` is evidence of holding the lock of the referenced mutex (line 2). Its API encapsulates the `Mutex`'s A&M pattern as follows:

- A `Mutex` issues a guard to a thread when it acquires the lock (line 4).

- The thread can then access the value with a mutable reference provided by the guard (line 7).
- When the guard goes out of scope, its destructor releases the lock, allowing other threads to acquire it.

While the concept of `Guard` originates from C++'s `std::lock_guard` [41], Rust's abstraction offers a much stronger safety guarantee than C++'s. In C++, a thread acquires the lock when the guard is constructed and releases it upon the guard's destruction by going out of scope. This mechanism prevents the lock from not being released after the use of the protected value. However, C++'s `std::mutex` [42] is merely a lock and not coupled with a value. Threads access the protected value directly, not through the guard. Thus, they can access the protected value without holding the lock despite the use of guards.

6.2.4 Raw Pointer or Interior Mutability?

A&M patterns with dynamically determined lifetimes, such as `Rc` in Rust's standard library, should be implemented through raw pointers. Only raw pointers facilitate mutation without any restrictions on lifetimes because interior mutability permits mutation via references during statically determined lifetimes.

Conversely, A&M patterns with statically determined lifetimes, such as `Mutex` in Rust's standard library, should be implemented through interior mutability. Interior mutability helps modular reasoning by encapsulating A&M-ness within types. Consider the following type `T`, where `y` is intended to be A&M:

```
struct T { x: X, y: Y }
```

With interior mutability, it suffices to change only the type of `y`:

```
struct T { x: X, y: UnsafeCell<Y> }
```

This modification allows `y` to be mutated even with a shared reference to `T`. The A&M-ness of `y` is safely confined within `T` and remains invisible externally.

However, with raw pointers, the only way to make `y` A&M is to use raw pointers to `T`, as shared references do not permit mutating `y`. This approach compromises modularity by delegating the reasoning about its safety to the users of `T`. Moreover, it restricts optimization opportunities. Even if `x` still follows A⊕M, the compiler considers `x` mutable.

This guideline is supported by observations of the Rust standard library. `Rc`, `Arc`, `LinkedList`, and `BTreeMap` are implemented with raw pointers, whose lifetimes are dynamically determined, allowing them to point to on-heap values that persist indefinitely. In contrast, `Cell`, `OnceCell`, `SyncOnceCell`, `RefCell`, `RwLock`, and `Mutex` employ interior mutability with pointers of statically determined lifetimes. They can be placed on stack and thus do not necessitate heap allocation. Therefore, pointers to these values are used in a lexically scoped manner.

6.3 A&M Patterns in OSs

To discover A&M patterns in OSs, we rewrite the xv6 OS [74] entirely in Rust. The result is xv6$_{Rust}$, which is functionally equivalent to xv6 except that it supports a few more system calls (e.g., `getppid`, which returns the process ID of the parent) and ARMv8 in addition to RISC-V. It is available online [103] and open-source.

We completely identify A&M patterns in xv6$_{Rust}$ and design their modular abstractions. We systematically identify A&M patterns in xv6$_{Rust}$ by drawing a dependency graph among types and finding

```
1  struct OwnedData { cwd: RcInode, ... }
2  struct Proc { owned: UnsafeCell<OwnedData>, ... }
3  struct CurrentProc<'a> { ptr: &'a Proc }
4  unsafe fn current_proc() -> CurrentProc<'_> { ... }
5  unsafe fn user_trap() {
6      let proc = unsafe { current_proc() };
7      ...
8  }
9  impl CurrentProc<'_> {
10     fn owned_mut(&mut self) -> &mut OwnedData { ... }
11 }
```

Listing 6.3: Modular abstraction of process-owned value

all the reachable nodes from `UnsafeCell` and raw pointer types, which are the only sources of A&M-ness in Rust. However, some of them are not A&M patterns; they just depend on other types providing A&M-ness. For this reason, we exclude each type whose path from `UnsafeCell` or a raw pointer type contains a type for an A&M pattern. Then, we additionally exclude already-known patterns, such as `Mutex`. As a result, we discover six A&M patterns whose modular abstractions have not yet been proposed and design a modular abstraction for each pattern.

In the rest of the section, we explain each of the six A&M patterns in xv6$_{Rust}$ and its abstraction.

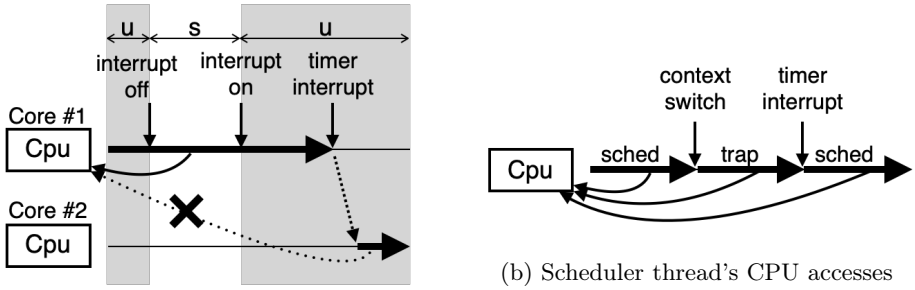
6.3.1 Process-Owned Value

Pattern A process-owned value is shared among threads without being protected by a lock. For each process-owned value, only a thread handling a certain process's system call accesses the value. In xv6, whose process consists of only one thread, PCBs have process-owned values, e.g., `cwd`. Section 6.1 explains the pattern in detail.

Abstraction We propose a modular abstraction for process-owned values. Note that thread-local storage (TLS) [80] cannot be utilized for process-owned values. TLS creates per-thread variables by storing values in a private memory area of each thread, preventing access to thread-local variables of other threads. In contrast, PCBs are stored in the shared memory area of the kernel. Although any thread can access PCBs, process-owned values are exclusively accessed by the thread that handles a system call of a specific process. This guarantee of exclusive accesses is derived from the behavior of the kernel, not where process-owned values are stored, and our abstraction leverages this behavior.

Listing 6.3 shows modular abstraction for process-owned values. We aggregate process-owned values of `Proc` in the type `OwnedData` (line 1) and put `OwnedData` in an `UnsafeCell` (line 2). `Proc` values are stored in global variables, so none of the threads own `Proc`s, and each thread has only shared references to `Proc`. The use of `UnsafeCell` allows mutation of `OwnedData` by the owning process while the enclosing `Proc` is shared by multiple threads as shared references.

Providing a safe method to mutate `OwnedData` is challenging. A naïve approach would be a method that takes `&Proc` as an argument and returns `&mut OwnedData`. But, it is unsound because it allows creating multiple mutable references to the same `OwnedData`, breaking the invariant of process-owned values and Rust's aliasing model.



(a) CPU accesses and interrupts

(b) Scheduler thread's CPU accesses

where `u`/`s` for unsafe/safe accesses to `Cpu`, × for inaccessible,
`sched` for a scheduler thread, and `trap` for a trap-handling thread

Figure 6.4: CPU accesses from threads

As a solution, we introduce `CurrentProc`, which represents the current process. It wraps a shared reference to a `Proc` (line 3). `CurrentProc` has two invariants: (1) the inner reference must point to the `Proc` of the process for which the current thread works; and (2) at most one `CurrentProc` value exists in each thread. To satisfy the second invariant, `current_proc`, which creates a `CurrentProc` value, is an unsafe function that requires a caller to ensure the absence of a `CurrentProc` (line 4). The condition is easily validated because it is called only in `user_trap`, a kernel's entry point where every trap (including system calls, CPU exceptions, and hardware interrupts) from the user space arrives (line 6). The `kernel_trap` function, an entry point for kernel-mode traps, does not call `current_proc` because the kernel does not need to access PCBs while handling kernel-mode traps. `CurrentProc` has a method `owned_mut`, which returns a mutable reference to the inner `OwnedData` (line 10). The invariant and API of `CurrentProc` ensure a process's `OwnedData` is exclusively accessed from the unique thread that handles a system call for the process.

While every process has a single thread in xv6$_{Rust}$, we can still apply the same abstraction to OSs with multi-threaded processes. In such OSs, *thread control blocks* (TCBs) have *thread-owned values*, each of which is accessed only by a thread handling a system call of a certain user thread. We can use the same strategy to implement thread-owned values in TCBs because each user thread triggers at most one system call at each time.

6.3.2 CPU-Owned Value

Pattern A value is CPU-owned if it is accessed only by the thread running on a specific CPU core. CPU-owned values are similar to process-owned values in that they are shared among multiple threads without locks. However, they are different form process-owned values because unique access is guaranteed by that at most one thread runs on a CPU core each moment, instead of that at most one thread handles a certain system call. We aggregate the values owned by a CPU core in a `Cpu`. A `Cpu` has, e.g., a pointer to the process running on the core. When the kernel runs on a multi-core CPU, the kernel has multiple `Cpu` values.

Modular abstraction of CPU-owned values has two challenges. First, a thread may move to another core after a context switch incurred by a timer interrupt. As Figure 6.4a illustrates, when a thread moves to another core after an interrupt, any existing pointers to the previous `Cpu` must no longer be dereferenced. The thread must newly acquire a pointer to the `Cpu` where it is now located. It implies

```
1 struct HeldInterrupts;
2 impl Cpu {
3     fn push_off(&self) -> HeldInterrupts { ... }
4     fn pop_off(&self, intr: HeldInterrupts) { ... }
5 }
6 static mut CPUS: [Cell<Cpu>; NCPU] = ...;
7 fn current_cpu(intr: &HeldInterrupts) -> &Cell<Cpu> { ... }
```

Listing 6.4: Modular abstraction of CPU-owned value

that a pointer to a `Cpu` is guaranteed to remain dereferenceable only while interrupts are disabled: when interrupts are enabled, a context switch might have happened already, so pointers to a `Cpu` are unreliable. In fact, it is enough to disable only preemption or inter-CPU task migration to make pointers to a `Cpu` reliable. Although disabling all the interrupts is inefficient, this work follows xv6's implemenation. One can easily improve the performance by disabling only preemption or inter-CPU task migration.

Second, the scheduler may reference `Cpu`. The kernel creates a scheduler thread for each core while booting. As in Figure 6.4b, each scheduler thread obtains a pointer to the `Cpu` where it runs and retains the pointer until the kernel terminates. At the same time, a thread handling a trap also requires, at least temporarily, a pointer to the `Cpu` where it runs to get a pointer to the current process and create a `CurrentProc`.

Abstraction Listing 6.4 presents a modular abstraction of CPU-owned values. To address the first challenge, we adopt, from the Tifflin kernel [98], the idea of `HeldInterrupts` type that serves as evidence of disabling interrupts (line 1). `HeldInterrupts` has one invariant: interrupts must be disabled if a `HeldInterrupts` value exists. A `HeldInterrupts` value is created by a function `push_off`, which disables interrupts of the current core (line 3). Calling a function `pop_off` is the only way to enable interrupts again (line 4). As their names imply, `push_off` and `pop_off` work in a stack-like manner: they push and pop "off" onto an imaginary stack, respectively, and interrupts are enabled when the stack is empty. Since each call to `pop_off` consumes a `HeldInterrupts` value, a `HeldInterrupts` exists only when interrupts are disabled. `HeldInterrupts` is a zero-sized type and does not introduce any run-time overhead.

For the second challenge, we put `Cpu`s in `Cell`s [34] (line 6). `Cell` is a standard library type with interior mutability that provides getter and setter (but not a reference) for the interior value via shared references. This API prevents memory bugs even when multiple shared references are used to mutate the same `Cell`'s interior value. `Cell` itself does not have a synchronization mechanism, but it is safe for CPU-owned value because at most one thread runs on each core at a time.

Putting it all together, the function `current_cpu` takes `&HeldInterrupts` as an argument and returns a reference to the current `Cpu` with the same lifetime as the argument (line 7). It ensures interrupts are disabled while the `Cpu` is in use.

Our abstraction facilitates a better understanding of CPU-owned values, an important A&M pattern in xv6, while not being completely novel. It utilizes two well-known concepts in the Rust community: guards of lock-protected values and `Cell`. Our work provides a better view of this pattern by decomposing it into well-known patterns.

6.3.3 Memory Pool

Pattern Memory pool maintains a chunk of memory consisting of multiple entries of the same type. Like heap, memory pool provides dynamic allocation, but it incurs less run-time overhead than heap because, among other things, it does not suffer from fragmentation. For example, memory pool is used to maintain (in-memory) inodes, each consisting of an inode number and a copy of on-disk inode data. In this work, we focus on memory pools with reference-counted entries because many use cases of memory pools involve reference counting.

When the kernel needs a specific inode, it invokes the `find_or_alloc` method of the memory pool. The method takes an inode number for a key and an initial value as arguments. If the memory pool already has an entry with the same key, it increases the entry's reference count by one and returns a pointer to the entry. Otherwise, the memory pool finds an unoccupied entry, initializes it, sets the reference count to one, and returns a pointer. We call such a pointer to an inode in a memory pool `RcInode`, which is reference-counting and thus read-only.

A memory pool must ensure the absence of duplicated keys. Otherwise, processes may refer to different in-memory copies of a single on-disk inode, leading the processes to corrupt data and incurring functionality (and possibly safety) bugs.

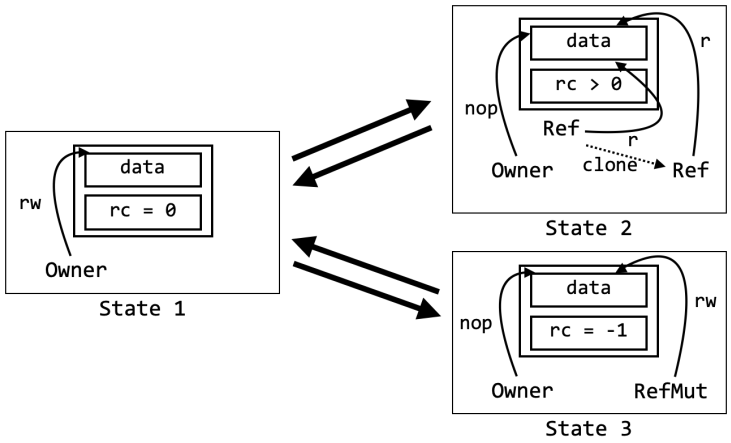
`RcInode`s are cloned and destroyed—and their corresponding reference counters are increased and decreased, respectively—in system calls. For example, when a process forks, the parent's working directory, represented as an `RcInode`, is cloned and used for the child's working directory, and when a process terminates, its working directory is destroyed. When a reference count becomes zero, a pre-registered finalizer is called to destruct the value in the entry. The finalizer of an `RcInode` checks any remaining file-system link to the inode. If no such link exists, then the file denoted by the inode has been deleted from the file system already, and thus the finalizer removes the content of the file from the disk.

The example shows three functionalities of each entry in a memory pool. First, an entry allows the memory pool to mutate its content when its reference count is zero for the initialization of `find_or_alloc`. Second, an entry provides multiple read-only cloneable pointers returned by `find_or_alloc`. Third, when a pointer to an entry is unique, it can mutate the referent. It is essential for the finalization of an entry, which happens when the last pointer goes out of scope because the finalization may need to mutate the value in the entry.

Abstraction We design memory pool as an array of entries. To avoid data races among multiple threads, we protect the entire array with a single lock and allow only one thread to traverse the array at a time. An alternative design would protect each entry with per-entry lock, but this allows multiple threads to simultaneously allocate distinct entries with the same key, breaking memory pool's functional requirement.

Each memory pool entry is reference-counted, but none of the reference-counted types provided in the Rust standard library satisfies memory pool's functional requirement. The Rust standard library has three reference-counted types: `Rc` [36], `Arc` [37], and `RefCell` [35]. `Rc` and `Arc`—`Rc`'s thread-safe version—introduce pointer indirection, which defeats the purpose of memory pool as allocator. `RefCell`'s lifetime is statically determined, but that of memory pool entries is dynamically determined. For example, the life cycle of in-memory inodes is dynamically determined by system calls from user processes and, therefore, cannot be expressed by static lifetimes. In addition, `RefCell` is not thread-safe, but memory pools are shared among multiple threads.

For memory pool entries, we propose `ArcCell`, a reference-counted type that serves memory pools'



where r for read-only and rw for read & write

Figure 6.5: Three states of `RefCell` and `ArcCell`

```
struct ArcCell<T> { data: T, rc: AtomicUsize }
struct Ref<T> { ptr: *mut ArcCell<T> }
struct RefMut<T> { ptr: *mut ArcCell<T> }
impl<T> ArcCell<T> {
    fn try_borrow(self: Pin<&mut Self>) -> Option<Ref<T>> { ... }
}
impl<T> Drop for ArcCell<T> {
    fn drop(&mut self) { while self.rc != 0 {} }
}
impl<T> Ref<T> {
    fn try_into_mut(self) -> Result<RefMut<T>, Self> { ... }
}
```

Listing 6.5: Modular abstraction of `ArcCell`

purposes. It is atomic, has dynamic lifetime like `Arc`, and does not introduce pointer indirection like `RefCell`. `ArcCell` provides two sorts of pointer: read-only `Ref` and read-write `RefMut`.

Our design of `ArcCell` is inspried by the design of `RefCell`. Figure 6.5 illustrates how `RefCell` and `ArcCell` operate. We discuss the operation of `RefCell` first and then how `ArcCell` differs from `RefCell`.

In State 1, which is the initial state, `RefCell` does not have pointers, and its reference count is zero. Since only the owner can access the `RefCell`, it can read and mutate the data. For memory pool, this state represents an unoccupied entry that can be initialized by the memory pool. In State 2, one or more `Refs` exist, and the reference count equals the number of the `Refs`. This state is reachable from State 1 by creating a new `Ref`. Since each `Ref` is read-only, multiple `Refs` may coexist, and the owner can create `Refs` repeatedly. It is also possible to create new `Refs` by cloning existing `Refs`. When every `Ref` goes out of scope, the `RefCell` returns to State 1. In State 3, a single `RefMut` exists, and the reference count is a special value -1 indicating the presence of a `RefMut`. This state is reachable from State 1 by creating a new `RefMut`. Since a `RefMut` can mutate the data, it should be unique. Thus, a `RefMut` disallows cloning. In this state, the owner cannot create a new pointer. Destroying the `RefMut` changes the state to State 1.

We now discuss `ArcCell` presented in Listing 6.5. Just like `RefCell`, `ArcCell` has four invariants: (1)

```
struct StrongPinMut<'a, T> { ptr: *mut T }
impl<T> ArcCell<T> {
    fn try_borrow(self: StrongPinMut<'_, Self>) -> Option<Ref<T>> { ... }
}
```

Listing 6.6: Modular abstraction of `StrongPinMut`

if its reference count equals -1, then a single `RefMut` refers to the `ArcCell`; (2) if its reference count is n, which is not -1, then n `Ref`s refer to the `ArcCell`; (3) A `RefMut` can mutate both the data and the reference count; and (4) A `Ref` can read the data and mutate the reference count. However, `ArcCell` has three characteristics distinct from `RefCell`.

First, for thread safety, we use an `AtomicUsize` for the reference counter (line 1). `AtomicUsize` is the type of an integer that allows atomic operations such as `compare_and_swap`.

Second, `ArcCell` allows transitions from State 2 to State 3. In memory pool, when the last remaining pointer to an entry is destroyed, the pointer is used for the finalization of the entry. To mutate the value during the finalization, the pointer must become a `RefMut`. To this end, we add a method `try_into_mut` to `Ref`, which converts a `Ref` to a `RefMut` if possible (line 11). When the reference count is one, it consumes a given `Ref` and returns a `RefMut`. Otherwise, it returns the `Ref` back.

Third, `Ref` and `RefMut` do not have lifetime parameters because their lifetimes are not statically determined (lines 2 and 3). Thus, we implement them with raw pointers instead of `UnsafeCell`. Since we use raw pointers, we need to prevent the possibility of dangling pointers. To this end, (1) we disallow the move of an `ArcCell` value by using `Pin`. Recall that `Pin` prevents values from moving. By making the methods of `ArcCell` take pinned references, we prohibit `ArcCell`s from moving after the creation of pointers (line 5); (2) we ensure an `ArcCell` value is not destructed in the presence of remaining pointers by inserting dynamic checking into the destructor of `ArcCell` (line 8).

StrongPinMut `ArcCell` presented so far, however, is not fully safe because it violates Rust's aliasing model. According to Stacked Borrows, the state-of-the-art aliasing model for (Unsafe) Rust [114], a mutable reference and raw pointers should not coexist: when a mutable reference is created, any existing raw pointers to the same referent become invalidated even if the address itself is still valid. Dereferencing such invalidated pointers is considered a UB because the compiler optimizes programs assuming that the mutable reference is unique. For `ArcCell`, each of `Ref`s and `RefMut`s contains a raw pointer to an `ArcCell`, and thus they are valid only until a mutable reference is created, and `Pin<&mut ArcCell>` contains a mutable reference inside it. Thus, if one creates a `Ref` from an `ArcCell`, creates a `Pin<&mut ArcCell>` pointer, and reads the value through the `Ref`, then it is a UB.

To resolve this problem, we propose `StrongPinMut`, presented in Listing 6.6, that strengthens `Pin<&mut>`. `StrongPinMut` is a type of a mutable pointer with three invariants: (1) the referent must not move; (2) there must be at most one `StrongPinMut` to a certain referent; and (3) mutable references to the same referent must not coexist with `StrongPinMut`. To this end, a `StrongPinMut` consists of a raw pointer, so it can coexist with raw pointers without invalidating them (line 1). We make `ArcCell`'s methods take `StrongPinMut`s instead of pinned references (line 3). For example, to create a `Ref` or `RefMut` from an `ArcCell`, the owner should give a `StrongPinMut` to the `ArcCell`, preventing the created references from being invalidated by mutable references.

Concurrently with our development of `StrongPinMut`, the Rust community also has discussed the

```
impl<T> Mutex<T> {
    fn lock(self: Pin<&Self>) -> Guard<'_,T> { ... }
}
impl<T> Guard<'_, T> {
    fn deref_mut(&mut self) -> Pin<&mut T> { ... }
}
```

Listing 6.7: Modular abstraction of lock-protected immovable value

same problem of Stacked Borrows being too restrictive to immovable values [113]. While the discussion is still ongoing, a temporary solution exempting the uniqueness requirement for mutable references to immovable values [112] has been proposed and applied to Miri [8], a Rust interpreter detecting UBs. Although it changes the aliasing model of Rust, its impact on existing Rust code has not been systematically studied yet. At the same time, there are proposals of new types that avoid the invalidation of raw pointers without any changes to the aliasing model, just like `StrongPinMut`, as alternative solutions. However, their designs have not been thoroughly investigated yet. We believe `StrongPinMut` is a strong candidate for solving this problem because it does not change the aliasing model of Rust and has proven its practicality by being used in xv6$_{Rust}$.

6.3.4 Lock-Protected Immovable Value

Pattern OSs protect various immovable values with locks. For example, `Disk`, the interface type of a single disk, is immovable due to its array for direct memory access (DMA) of its target disk. When a `Disk` is initialized, it informs the disk about the array's address, and the disk reads and modifies the content of the array during I/O. In addition, each `Disk` must be protected with a lock because multiple threads may simultaneously try to initiate disk operations.

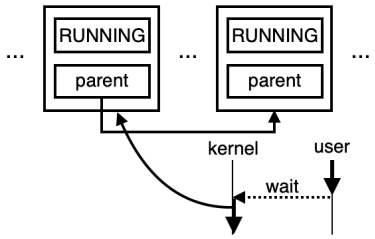
The standard implementation of `Mutex` does not work well with immovable values for two reasons. First, when a `Mutex` moves, the inner value moves together, breaking the invariant of the inner immovable value. Second, a `Mutex` exposes a mutable reference to the inner immovable value via `Guard`, breaking the same invariant again.

Abstraction Listing 6.7 shows a modular abstraction of `Mutex` that supports immovable values by disallowing the inner value from moving. It has two invariants: (1) the inner value must be accessed only by the thread holding the lock; and (2) the inner value must not move. To satisfy the second invariant, we make two changes: (1) to acquire a lock, a thread requires a pinned reference instead of a shared reference (line 2), enforcing the immovability of `Mutex` acquired at least once; (2) `Guard` provides a pinned reference instead of a mutable reference (line 5), preventing exposing a mutable reference to the inner value. Although the abstraction is straightforward, we help understand this real-world pattern by decomposing it into two well-known types, `Pin` and `Mutex`.

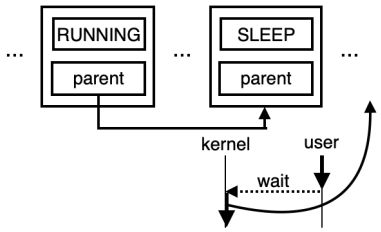
6.3.5 Lock-Protected Separated Value

Pattern While the standard `Mutex` protects only a single value with a lock, there is a need for protecting multiple separated values with a lock. An example is `Proc`'s `parent`.

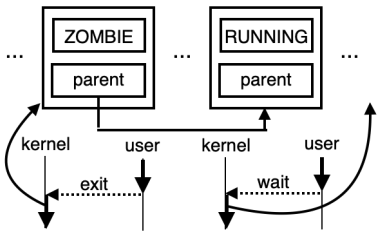
The `parent` field is shared among multiple threads, e.g., to handle a `wait` system call, as illustrated



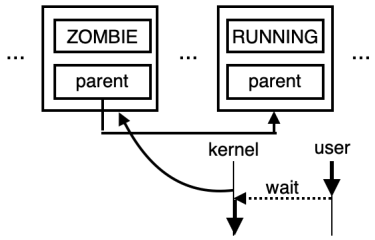
(a) A child is running when the parent checks the state.



(b) The parent reaches the end of the array and sleeps.

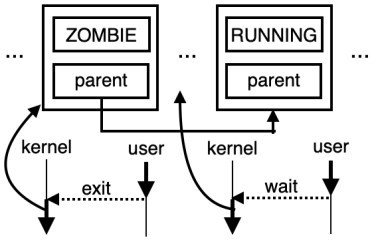


(c) The child terminates and wakes up the parent.

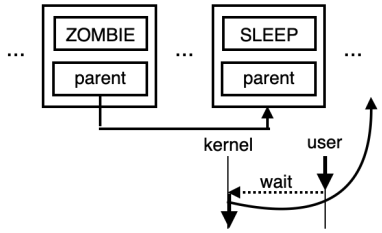


(d) The parent restarts the iteration and reaps the child.

Figure 6.6: How a `wait` system call is handled



(a) A child ends while its parent in a loop, failing wake-up.



(b) The parent reaches the array end and sleeps forever.

Figure 6.7: Problem of using per-process locks

in Figure 6.6. When a process handles `wait`, it iterates through the all-process array and compares each process's `parent` with itself to determine if the process is its child. If so, it checks the child's state (Figure 6.6a). If no child is a zombie and the process reaches the end of the array, it starts sleeping (Figure 6.6b). When a child terminates, it changes its state to `ZOMBIE`, and wakes up its parent (Figure 6.6c). Then, the parent re-iterates through the array and finds the zombie child (Figure 6.6d).

For functional correctness of `wait`, the `parent` fields of all `Procs` should be protected together by a single lock, as shown in Figure 6.7. If each `parent` were protected with a per-process lock, then a child might terminate while its parent is still in the loop. In that case, the child wakes up the parent before it sleeps, losing the wake-up (Figure 6.7a). When the parent reaches the end of the array, it sleeps forever unless it has another child (Figure 6.7b). To avoid the problem, we require a process to hold a single lock for all `parent` fields while (1) iterating the all-process array to handle `wait`; or (2) waking up its parent.

Abstraction Listing 6.8 shows a naïve implementation of `wait`. A `Procs` has `wait_lock`, which is a lock coupled with a value of type `()` (line 1). Since `()` is a zero-sized type, the lock is meaningless per se. Instead, it protects the separated `parent`s of the `Procs`. Each `Proc` stores a pointer to its parent in an

```
1  struct Procs { wait_lock: Mutex<()>, ... }
2  struct Proc { parent: UnsafeCell<*mut Proc>, ... }
3  struct WaitGuard<'a> { inner: Guard<'a, ()> }
4  impl Procs {
5      fn proc(&self, index: usize) -> &Proc { ... }
6      fn wait_guard(&self) -> WaitGuard<'_> { ... }
7  }
8  impl Proc {
9      fn parent_mut(&self, guard: &mut WaitGuard<'_>) -> &mut *mut Proc { ... }
10 }
11 fn wait(procs: &Procs, ...) {
12     let mut guard = procs.wait_guard();
13     for i in 0..NPROC {
14         let p = procs.proc(i);
15         let parent = p.parent_mut(&mut guard);
16         ...
17     }
18 }
```

Listing 6.8: Naïve abstraction of `wait` without branded types

`UnsafeCell` (line 2). A thread must acquire `wait_lock` to obtain a reference to the `parent`s.

A `WaitGuard`, which wraps a `Guard`, is evidence of holding `wait_lock` (line 3). When a thread acquires `wait_lock` of a `Procs`, the `Procs` returns a `WaitGuard` (line 6). Each `Proc` provides a mutable reference to its `parent` when a `WaitGuard` is given (line 9). The `wait` function, which handles a `wait` system call, first acquires `wait_lock` of a given `Procs` (line 12) and uses `WaitGuard` to access the `parent` of each `Proc` in the `Procs` (line 15).

The implementation so far, however, is unsafe because it admits data races. Consider two `Procs`es, `procs1` and `procs2`. One can acquire two `WaitGuard`s, one from `procs1` and the other from `procs2`. Since two `WaitGuard`s are indistinguishable with their types, both can be used to mutably access the `parent` of a `Proc` in `procs1`, which may incur data race.

To distinguish `WaitGuard`s from different `Procs`es, we employ *branded types*, types decorated with *brands*, which differentiate types that are otherwise the same. In Haskell, the `ST` monad [129] uses branded types. In Rust, branded types enable implementation of safe array indexing without run-time overhead [63], and `GhostCell`, a thread-safe cell with interior mutability without synchronization overhead [202]. As a new application of branded types, we brand locks and lock-protected separated values to ensure safety without overhead.

Listing 6.9 shows `wait` implementation with branded types. We make the following changes compared to the previous implementation without branded types:

- We brand `&Procs`. `ProcsRef<'id, 'a>` is a branded version of `&Procs` with a brand parameter `'id` (line 1). The brand parameter, as in [63, 202], has a form of a lifetime but its purpose is to distinguish different `&Procs`: for each brand `'id`, `ProcsRef<'id, 'a>` is a singleton type.
- We brand `&Proc` and `WaitGuard` to annotate in their types `ProcsRef` from which they originate. `ProcRef` is a branded version of `&Proc` (line 2). The brand of a `ProcRef` equals the brand of a

```
1  struct ProcsRef<'id, 'a> { inner: &'a Procs }
2  struct ProcRef<'id, 'a> { inner: &'a Proc }
3  struct WaitGuard<'id, 'a> { inner: Guard<'a, ()> }
4  impl<'id> ProcsRef<'id, '_> {
5      fn proc(&self, index: usize) -> ProcRef<'id, '_> { ... }
6      fn wait_guard(&self) -> WaitGuard<'id, '_> { ... }
7      fn new<'a, F: for<'id> FnOnce(ProcsRef<'id, 'a>)>(procs: &'a Procs, f: F) {
8          ...
9      }
10 }
11 impl<'id> ProcRef<'id, '_> {
12     fn parent_mut(&self, guard: &mut WaitGuard<'id, '_>) -> &mut *mut Proc {
13         ...
14     }
15 }
16 fn wait<'id>(procs: &ProcsRef<'id, _>, ...) { ... }
```

Listing 6.9: Modular abstraction of `wait` with branded types

`ProcsRef` it belongs to. The `proc` method, which is now a method of `ProcsRef`, returns a `ProcRef` of the same brand (line 5). `WaitGuard` now also has a brand (line 3). The method `wait_guard` returns a `WaitGuard` of the same brand (line 6). `ProcRef` and `WaitGuard` have the same invariant: each can be created only from a `ProcsRef` of the same brand.

- We make the `parent_mut` method aware of brands. It is now a method of `ProcRef` and requires a `WaitGuard` of the same brand as an argument (line 12). To access the `parent` of a `ProcRef` from a certain `ProcsRef`, a thread needs a `WaitGuard` originated from the same `ProcsRef`. Consider the previous example again where one uses a `WaitGuard` from `procs2` to access the `parent` of a `ProcRef` from `procs1`. It fails to pass type checking because the `WaitGuard` and the `ProcRef` have different brands.

- The function `ProcsRef::new` creates a `ProcsRef` using a `&Procs` and a closure that takes a `ProcsRef` as an argument (line 7). For example, the following code creates a `ProcsRef` with the given `&Procs` and applies the closure to the `ProcsRef`:

  ```
  ProcsRef::new(procs, |p| wait(p, ...));
  ```

  Since the closure is universally quantified over `'id`, the lifetime is opaque, which ensures that two different `ProcsRef`s always have different brands.

6.3.6 Asynchronous Ownership Transfer for DMA

Pattern DMA allows peripheral devices to read from or write to memory directly. It is preferred over programmed I/O because it enables data transfers between memory and devices without involving the CPU, allowing efficient data movement by freeing up the CPU to perform other tasks. For DMA, OSs need to send resource ownership to hardware and back. For example, OSs transfer a buffer to a disk to read from or write on the disk.

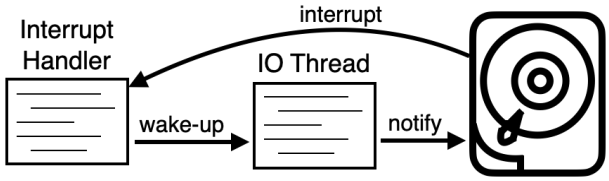


Figure 6.8: Ownership transfer of buffer in disk operations

```
1  struct Disk { inflight: [InflightInfo; NUM], ... }
2  struct InflightInfo { b: *mut Buf, ... }
3  impl Disk {
4      fn rw(&mut self, b: &mut Buf) {
5          ...
6          self.inflight[idx].b = b as *mut Buf;
7          unsafe { MmioRegs::notify_queue(0); }
8          b.wait_channel.sleep();
9          ...
10     }
11     fn intr(&mut self) {
12         ...
13         unsafe { (*self.inflight[idx].b).wait_channel.wakeup(); }
14     }
15 }
```

Listing 6.10: Modular abstraction of ownership transfer to disk

Due to the asynchronous nature of I/O, its ownership transfer is fairly different from that of normal Rust function calls. For example, Figure 6.8 illustrates how a buffer's ownership transfers between kernel and disk. An I/O thread sends a buffer to a disk with memory-mapped I/O (MMIO) registers by writing the buffer's address in an MMIO register and then a special value in another MMIO register to notify the disk. After the notification, the disk owns the buffer, and the thread starts sleeping not to access the buffer. When the disk finishes the operation, it raises a hardware interrupt that passes the buffer from the disk to the kernel's interrupt handler. Finally, the handler wakes up the sleeping thread to complete the job, giving the buffer back to the thread.

Abstraction Listing 6.10 shows `Disk`, a modular abstraction for asynchronous ownership transfer for disk operations. `Disk` has two methods `rw` and `intr`, for the I/O thread and the interrupt handler, respectively. The `rw` method takes a mutable reference to a buffer as an argument and initiates disk operations. During the execution of `rw`, `Disk` has an invariant: the mutable reference to the buffer has a unique owner. The reference to the buffer is converted to a raw pointer and written in `inflight`, which is accessible from the disk (line 6). The disk is notified to perform its operation with the buffer (line 7), which is safe because the thread immediately sleeps (line 8). After the disk operation finishes, the interrupt handler calls `intr` that wakes up the thread sleeping in `rw` (line 13), which effectively transfers the buffer back from disk to `intr` and then `rw`. In doing so, the dereference in `intr` is safe because it temporarily owns the buffer on line 13.

Table 6.1: Manual analysis of the A&M patterns in other OSs

	Process-owned	CPU-owned	Memory pool	Lock, immovable	Lock, separated	Asynch transfer
Linux	●	●	●	●	●	●
`xv6-rust`	●[n]	●[n]	●	○	●[n]	●[n]
Tock	○	○	○	○	○	○
RustyHermit	○	●[n]	○	○	○	○
Theseus	○	○	●	○	○	○
RedLeaf	○	●	●	○	○	○
Redox	○	●	●	○	○	○

n: non-modular abstraction

6.4 Evaluation

We evaluate the identified patterns and the proposed modular abstractions with the following three research questions:

- RQ1. Existence of abstractions: Do the modular abstractions of the A&M patterns exist in legacy OSs and Rust OSs? (Section 6.4.1)
- RQ2. Effectiveness in reducing unsafe code: Do the modular abstractions effectively reduce the number of unsafe lines? (Section 6.4.2)
- RQ3. Impact on performance: Do the modular abstractions incur only modest overhead at run time? (Section 6.4.3)

6.4.1 RQ1: Existence of Abstractions

To show that the proposed abstractions are practical and original, we investigate whether the six A&M patterns and their abstractions in xv6 are utilized in Linux, a representative legacy OS, Qi et al.'s `xv6-rust` [170], another port of xv6 to Rust, and five clean-slate Rust OSs: Tock [131], RustyHermit [128], Theseus [67], RedLeaf [156], and Redox [10].

Table 6.1 shows the result. Linux uses all six patterns. Thus, our abstractions are practical in that they can help rewrite legacy OSs in Rust. `xv6-rust` utilizes the same A&M patterns as xv6 but does not provide modular abstractions for most patterns. It directly accesses A&M values using unsafe operations, similar to xv6, instead of encapsulating each value within a modular abstraction and accessing them through a set of API calls. This is because `xv6-rust` aims not at designing modular abstractions for A&M values to reduce the reasoning cost, but rather focuses on syntactically translating C to Rust and adopting some well-known Rust idioms, e.g., using iterators instead of index-based loops. Some clean-slate Rust OSs use CPU-owned values and memory pools, but their abstractions are different from ours. Since none of the existing OSs have proposed the same abstractions, our abstractions are original.

Process-Owned Value Linux uses process-owned values in `task_struct`, the type of PCBs. While `task_struct` protects many of its fields with locks, some fields including `journal_info` are accssed by a single thread and not protected by any locks.

```
struct task_struct { void *journal_info; ... }
```

Such fields are process-owned values and can benefit from our abstraction.

`xv6-rust` uses process-owned values without a modular abstraction. It employs `UnsafeCell` but always accesses the inner data with an unsafe method provided by `UnsafeCell`.

Tock does not use process-owned values. Its system calls are non-blocking; the kernel may have multiple ongoing system calls from a single process. Process-owned values can exist only when the kernel handles only one system call from a single process at a time. Therefore, although Tock provides *grants*, per-process kernel heap, they are not process-owned values. Since multiple kernel threads handling system calls from the process owning a particular grant may access the grant simultaneously, Tock protects each grant with run-time checks, like Rust's `RefCell`, but not like process-owned values.

The other Rust OSs utilize reference-counted or lock-protected values, instead of process-owned values, thereby requiring run-time checks even when the unique access is guaranteed. Note that xv6$_{Rust}$ can access process-owned values without any run-time checks. In RustyHermit, PCBs have the type `Task`, and `Rc` and `RefCell` protect each `Task`.

```
struct PerCoreScheduler { current_task: Rc<RefCell<Task>>, ... }
```

In Theseus, `Mutex` protects all the mutable values of `Task`.

```
struct Task { inner: Mutex<TaskInner>, ... }
```

In RedLeaf, values uniquely accessed by a system-call-handling thread are in `ThreadLocal`, which uses `Mutex`.

```
struct ThreadLocal<T> { values: Mutex<HashMap<u64, Option<T>>>, ... }
```

In Redox, the type of PCBs is `Context`, and `Context` protects each mutable field with `Arc` and `RwLock`.

```
struct Context { files: Arc<RwLock<Vec<Option<FD>>>>, ... }
```

CPU-Owned Value Linux uses CPU-owned values. For example, it has a run queue for each CPU core. Each run queue is modified only by the thread running on the corresponding core. The kernel disables preemption before modifying a run queue.

`xv6-rust` uses CPU-owned values without a modular abstraction. It creates a mutable reference to a global data structure containing CPU-owned values without ensuring that the reference is used only while interrupts are disabled.

Tock does not use CPU-owned values because it supports only single-core processors. There are no per-core data.

RustyHermit uses CPU-owned values, but its abstraction is unsafe. In RustyHermit, per-core schedulers are CPU-owned values. The following function can construct multiple mutable references to a specific core's scheduler, while aliasing of mutable references is UB:

```
fn core_scheduler() -> &'static mut PerCoreScheduler { ... }
```

Theseus does not use CPU-owned values. It has per-core run queues, but they are implemented with lock-protected values.

```
static RUNQUEUES: AtomicMap<u8, RwLock<RunQueue>>;
```

RedLeaf and Redox use CPU-owned values, but their abstractions are different from that of xv6$_{Rust}$. They rely on the ELF TLS ABI [80] by using the `#[thread_local]` attribute [44] for CPU-owned values. The actual implementation of TLS depends on the compiler, linker, and architecture. RedLeaf defines per-core schedulers with CPU-owned values.

```
#[thread_local]
static SCHED: RefCell<Scheduler>;
```

Redox defines CPU core numbers with CPU-owned values.

```
#[thread_local]
static CPU_ID: AtomicUsize;
```

Memory Pool Linux uses a slab allocator [66], which is effectively a group of multiple memory pools for different data types. While slab allocators are more complicated than memory pools, they share the same idea of reducing fragmentation. Some structs such as `inode` have reference counts, and they are managed manually by each struct.

```
struct inode { atomic_t i_count; ... }
```

`xv6-rust` uses memory pools, but its abstraction is much more inefficient than xv6 and xv6$_{Rust}$. It protects each cell in a memory pool with an individual lock in addition to a lock protecting the whole memory pool, thereby always acquiring a lock when accessing the data in each entry.

Tock does not use memory pools. It uses grants for dynamic memory allocation, and grants do not reserve a chunk of memory.

RustHermit does not use memory pools. It uses heap managed by a linked list of free blocks. It is less efficient for reducing fragmentation than memory pools.

Theseus, RedLeaf, and Redox use slab allocators. They use `Rc` and `Arc` in the standard library for reference counting.

Lock-Protected Immovable Value Linux has lock-protected immovable values. For example, `super_block` has `s_inodes`, which is a list entry for an intrusively linked list. Since `s_inodes` is referenced by adjacent nodes, `super_block` must not move. At the same time, `super_block` is protected by `s_inode_list_lock`.

```
struct super_block {
    spinlock_t s_inode_list_lock;
    struct list_head s_inodes;
    ...
}
```

Therefore, `super_block` is a lock-protected immovable value.

The Rust OSs do not have lock-protected immovable values. They do not use `Pin` at all or use `Pin` only without locks.

Lock-Protected Separated Value Linux uses lock-protected separated values. For example, `task_struct` has `parent`, a pointer to the parent process, and there is a global lock named `tasklist_lock` sychronizing accesses to `parent`.

```
struct task_struct {
    struct task_struct *parent;
    ...
}
rwlock_t tasklist_lock;
```

Table 6.2: Numbers of total and unsafe lines of code

OS	LOC	LOUC	Unsafe code percentage
xv6_{Rust}	11676	345	2.95%
`xv6-rust`	6579	339	5.15%
Tock	135347	1559	1.15%
RustyHermit	39860	1114	2.79%
Theseus	48886	930	1.90%
RedLeaf	29430	949	3.22%
Redox	56785	1306	2.30%

The parent process acquires `tasklist_lock` before iterating the task list to handle a `wait` system call, and the child process acquires `tasklist_lock` before notifying the parent process to handle an `exit` system call.

`xv6-rust` uses lock-protected separated values without a modular abstraction. Since it does not employ branded types, it cannot ensure whether a proper lock is acquired when accessing separated data.

The clean-slate Rust OSs do not use lock-protected separated values. None of them utilizes branded types. They uses lock-protected values instead by protecting the whole data structure, not individual fields. In RustHermit, the whole collection of waiting tasks are protected by a lock, and the lock must be held while accessing any fields of `Task` to handle a `join` system call.

```
static WAITING_TASKS: Mutex<BTreeMap<Id,Vec<Task>>>;
```

In Redox, the whole list of processes is protected by a lock, and accessing any fields of `Context` during a `waitpid` system call requires holding the lock.

```
static CONTEXTS: RwLock<ContextList>;
```

The use of lock-protected values for this purpose leads to the more frequent acqusition of locks, which degrades performance, compared to lock-protected separated values.

Asynchronous Ownership Transfer for DMA Linux uses asynchronous ownership transfer for DMA. It has DMA notified by interrupts. `xv6-rust` also uses asynchronous ownership transfer for DMA, but its ownership transfer is not explicitly described by a modular abstraction. On the other hand, the clean-slate Rust OSs use polling for DMA, unlike xv6_{Rust} and Linux.

6.4.2 RQ2. Effectiveness in Reducing Unsafe Code

To show that the proposed abstractions successfully mitigate the burden of manual safety reasoning about A&M states, we compare the lines of unsafe code (LOUC) of xv6_{Rust} with those of the other Rust OSs. LOUC is a reasonable proxy to estimate the burden for manual reasoning [128] because safe code is automatically validated by type checking. We exclude the unsafe lines of libraries used by each OS from LOUC because developers do not reason about the safety of the libraries. Developers usually trust libraries, assuming that their developers have reasoned about their safety.

We measure LOUC by counting the number of lines in unsafe blocks that are not in unsafe functions. In Rust, unsafe code must be placed in an unsafe block unless it resides in an unsafe function. Since most lines in unsafe blocks consist of unsafe code, counting these lines is a reasonable approach. In contrast, only a small portion of the lines in unsafe functions contain unsafe code because an unsafe function signifies that calling the function, but not the operations in its body, is unsafe. Therefore, counting the lines in unsafe functions leads to a significant overestimation of LOUC, whereas not counting them results in a slight underestimation of LOUC. If the `deny(unsafe_op_in_unsafe_fn)` attribute [25] is used in the codebase, unsafe code always requires an unsafe block, even in an unsafe function. Thus, using this attribute allows for precise LOUC measurement through counting the lines in unsafe blocks. However, since only xv6$_{Rust}$ employs this attribute, counting the lines in unsafe blocks would result in an unfair comparison for xv6$_{Rust}$. For this reason, we count the lines in unsafe blocks that are not in unsafe functions, allowing for a slight underestimation of every OS's LOUC. To measure LOC and LOUC, we implemented a line-counting tool using the Rust parser, which excludes comments and empty lines.

Table 6.2 shows the result of each Rust OS. While both xv6$_{Rust}$ and `xv6-rust` port xv6 to Rust, `xv6-rust` has a higher ratio of LOUC to LOC than xv6$_{Rust}$. Since `xv6-rust` has not ported all the functionalities of xv6, the LOC of `xv6-rust` is much smaller than that of xv6$_{Rust}$, but its LOUC is already comparable to that of xv6$_{Rust}$. Using our abstractions, xv6$_{Rust}$ achieves a similar reasoning cost to `xv6-rust` while providing more functionalities. This demonstrates that the abstractions are effective in reducing the number of unsafe lines.

The proportion of unsafe code in xv6$_{Rust}$ is slightly higher than (but within the same order of magnitude as) the proportion of unsafe code in the clean-slate Rust OSs, except for RedLeaf. We believe that the design of xv6$_{Rust}$ unavoidably leads to more unsafe code. To utilize A&M patterns in legacy OSs, xv6$_{Rust}$ has to implement its own modular abstractions for the A&M patterns in its codebase with some amount of unsafe code. On the other hand, the other OSs rely only on well-known A&M patterns. Although such patterns also require unsafe code for implementation, their implementation resides in pre-existing libraries, not the codebase of the OSs. It allows them to complete their functionalities with less amount of unsafe code. In conclusion, our abstractions reduce the reasoning cost of xv6$_{Rust}$ to the level of the other Rust OSs despite the use of A&M patterns originated from legacy OSs.

6.4.3 RQ3: Impact on Performance

To show that the run-time overhead of the proposed abstractions is modest, we compare the performance of xv6$_{Rust}$ with that of xv6. We compile xv6 with GCC 9.3.0 and xv6$_{Rust}$ with Rust nightly-2021-06-19, both using the `-O3` optimization level. We run these OSs in the QEMU emulator [64] running on an Ubuntu 20.04 machine with AMD Ryzen 5900X (12 cores, 24 threads, 3.7GHz), 64GB DRAM, and SSD. We measure the execution cycles of xv6's `usertests` benchmark programs on xv6 and xv6$_{Rust}$ ten times. We attempted to include `xv6-rust` in this experiment as well because it is another port of xv6 to Rust. However, `xv6-rust` has not fully ported the functionality of xv6, preventing most programs in the `usertests` suite from running. Thus, we exclude `xv6-rust` from this experiment.

Figure 6.9 shows the average execution cycles of the benchmark programs on xv6$_{Rust}$ compared to xv6. Each vertical bar represents a program and its height is the ratio of its average execution time on xv6$_{Rust}$ to that on xv6. Table 6.3 describes each benchmark program. Surprisingly, xv6$_{Rust}$ performs 31.1% faster than xv6 on average. It turns out that our modular abstractions optimize the kernel although we only focused on safety reasoning while designing them. For example, xv6$_{Rust}$ calls `current_proc` only at the kernel entry point, but xv6 calls the equivalent function in multiple places, including loop bodies.

Table 6.3: Description of benchmarks

Benchmark	Description	Benchmark	Description
bsstest	execute a program	rmdot	remove directories
manywrites	write to a file concurrently	sharedfd	write to a file concurrently
truncate2	truncate and write to a file	exectest	fork and execute a program
writebig	write to and read a big file	sbrkfail	grow heap a lot
bigwrite	write to a big file	sbrkbasic	grow heap a bit
copyin	write to a file and a pipe	exitwait	fork many processes
writetest	write to a small file	sbrkmuch	grow and shrink heap
bigdir	create a big directory	twochildren	fork processes
iref	create and remove a directory and a file	bigargtest	execute a program
truncate1	truncate a file	reparent2	fork processes
concreate	create and remove a file concurrently	copyout	read a file and a pipe
rwsbrk	grow heap	sbrkbugs	shrink heap a lot
unlinkread	remove and read a file	forktest	forks a lot
createdelete	create and remove files concurrently	validatetest	create a symbolic link
fourfiles	write to files concurrently	kernmem	incur CPU exceptions
createtest	remove and read files	stacktest	overflow stack
fourteen	create and remove directories	argptest	read a file
linktest	create symolic links	execout	grow and shrink heap
dirtest	create a directory	reparent	fork processes
linkunlink	create and remove a file concurrently	badarg	execute a program
sbrkarg	grow heap and create a pipe	mem	grow and shrink heap
dirfile	create a directory and files	copyinstr2	open a file
iputtest	change the working directory	openiputtest	create, open, and remove a directory
subdir	create subdirectories	killstatus	kill a process
pgbug	grow heap and create a pipe	forkforkfork	fork processes
exitiputtest	create and remove a directory	pipe1	read and write to a pipe
truncate3	truncate and write to a file	copyinstr1	open a file
bigfile	write to and read a big file	copyinstr3	grow heap and open a file
opentest	open files	preempt	fork and kill processes
forkfork	fork concurrently		

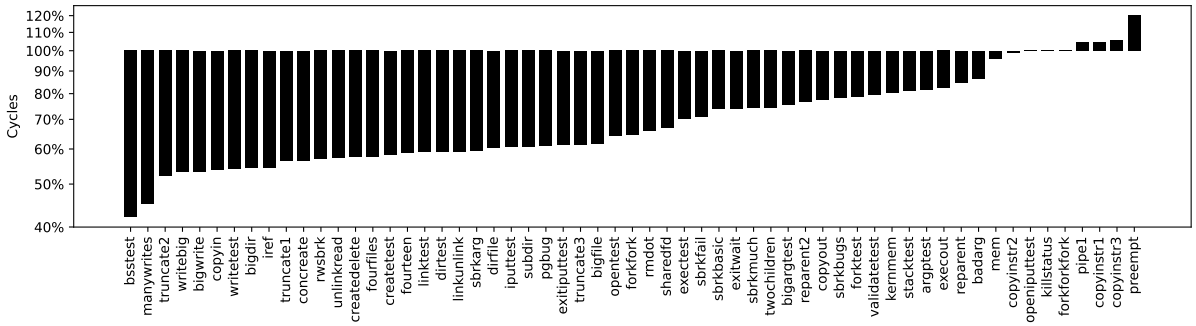


Figure 6.9: Execution cycles of each benchmark on xv6$_{Rust}$ compared to xv6

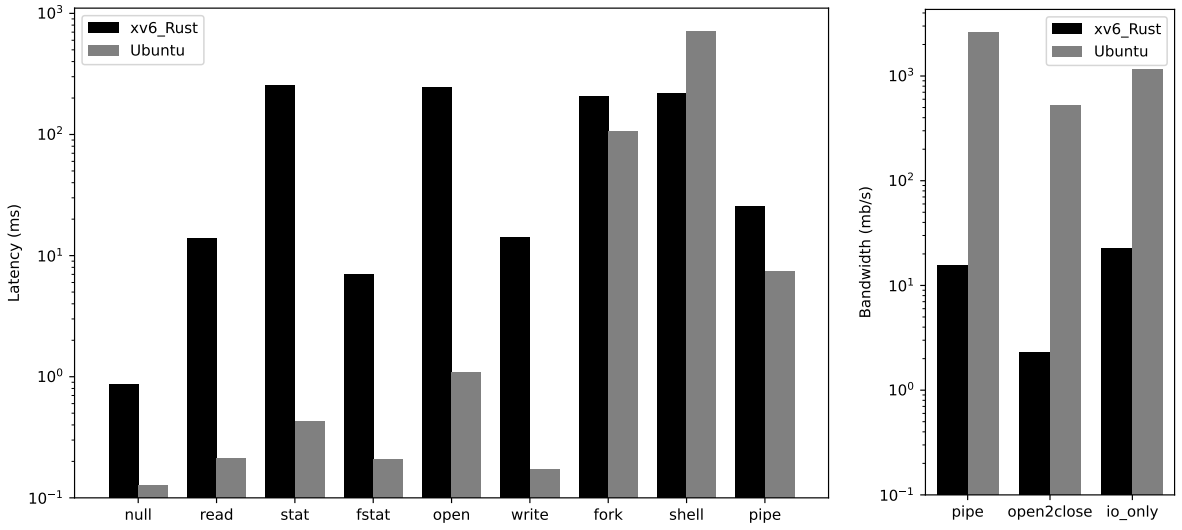
We could decrease the performance difference to 16.1% by applying the optimizations in xv6$_{Rust}$ to xv6. We anticipate that the remaining difference also originates from similar unintended optimizations or benefits from the compiler optimization based on the Rust aliasing model.

While xv6$_{Rust}$ performs 20% slower than xv6 for one benchmark program, our investigation shows that this slowdown is not attributable to the overhead of the proposed abstractions. The benchmark program creates three child processes via `fork`, `kill`s each child, and then `wait`s for each of them. If a context switch occurs due to a timer interrupt before the parent calls `wait`, the children handle the kill signals before the parent's `wait` call. This allows `wait` to find a zombie child in the first iteration and return immediately. Conversely, if a timer interrupt does not occur before calling `wait`, it reaches the end of the process list and makes the parent sleep until a child handles the kill signal, thereby significantly increasing the execution time, in our experiments, by more than twice. Therefore, the execution time largely depends on the timing of a timer interrupt. We find that timer interrupts occur more frequently before `wait` calls in xv6 than in xv6$_{Rust}$. Considering the results from other benchmarks, we believe that xv6$_{Rust}$ handles most system calls faster than xv6, increasing the likelihood of calling `wait` before a timer interrupt. This explains the observed slowdown, and thus, we do not consider the slowdown as evidence of the overhead incurred by the proposed abstractions.

Comparison with Linux To evaluate the performance of the abstractions, we additionally compare xv6$_{Rust}$ with the Linux kernel, specifically Ubuntu 18.04 [43]. We run both OSs on an ARMv8 machine with KVM virtualization [122]. We measure latency and bandwidth with 13 benchmark programs from LMbench [147]. We exclude the other benchmarks from our experiment because they cannot be executed on xv6$_{Rust}$. The excluded benchmarks fall into four categories: network-related, signal-related, `mmap`-related, and hardware-related.

- Network-related benchmarks include `bw_tcp`, `lat_tcp`, `bw_udp`, `lat_udp`, `bw_unix`, `lat_unix`, `lat_connect`, `lat_select`, and `lat_rpc`. To run these benchmarks, various system calls (e.g., `socket`, `bind`, `listen`) must be implemented, but neither xv6 nor xv6$_{Rust}$ incorporates a network stack. The integration of a network stack would present considerable challenges and likely necessitate new abstractions.

- A signal-related benchmark (`lat_sig`) requires the implementation of system calls to register signal handlers (`signal`, `sigaction`, `sigemptyset`, and `posix_kill`) and a mechanism for delivering signals to processes. This involves augmenting PCBs with signal-related metadata and their appropriate manipulation upon system calls and traps. We anticipate that these modifications would not present significant challenges nor require new abstractions.



(a) Latency

(b) Bandwidth

Figure 6.10: Performance of xv6$_{Rust}$ compared to Ubuntu 18.04

- `mmap`-related benchmarks (`bw_mmap_rd`, `lat_pagefault`, and `lat_mmap`) necessitate the implementation of the `mmap`, `munmap`, and `msync` system calls. This involves adding `mmap`-related metadata to PCBs and their appropriate manipulation upon system calls. We believe these changes would not present significant challenges nor necessitate new abstractions.
- Hardware-related benchmarks (`mhz`, `tlb`, `line`, `cache`, `stream`, `par_mem`, and `par_ops`) aim to measure hardware rather than kernel performance. They can be supported by increasing user-mode stack, heap, and file sizes, which does not require the introduction of new abstractions.

As shown in Figure 6.10, xv6$_{Rust}$ performs a few times to a few thousand times worse than Linux, depending on the benchmark program. We analyze the reasons for Linux outperforming xv6$_{Rust}$ as follows:

- Following the design of xv6, xv6$_{Rust}$ maintains only a single page table on the memory and swaps the page table at the start and end of each kernel trap. On the other hand, Linux always keeps page tables for both kernel and user processes on the memory. As a result, xv6$_{Rust}$ spends 15 clock cycles on swapping the page table, barrier instructions, and TLB cache flush in each system call, while Linux does not. Thus, short system calls in xv6$_{Rust}$ have a much longer latency than Linux.
- xv6$_{Rust}$ uses inefficient algorithms of xv6 without any change. For example, the `wakeup` function, which wakes up every sleeping process, acquires the lock of every existing process and incurs huge contention. On the other hand, Linux avoids such contention by managing a wait queue consisting only of sleeping processes.
- As in xv6, xv6$_{Rust}$ uses spin locks in many places. However, Linux replaces some of them with much cheaper locks, such as RCU read locks [54].
- Like xv6, xv6$_{Rust}$ disables interrupts when accessing CPU-owned values. Linux, in contrast, often employs more efficient strategies, e.g., disabling only preemption.

As our analysis shows, xv6$_{Rust}$'s performance issues originate from the design of xv6, which has little consideration for performance, not our abstractions for A&M patterns. We believe that we can make the performance of xv6$_{Rust}$ comparable to that of Linux by addressing the aforementioned inefficiencies.

Chapter 7. Related Work

Transforming C2Rust-Generated Code Several tools have been proposed to incorporate Rust's language features into C2Rust-generated code. However, they focuses only on syntactic improvement or removal of scalar pointers. CRustS [137] utilizes syntactic replacement rules and lacks the capability to perform intricate transformations that require a deeper understanding of semantics. Both Laertes [84, 83] and Crown [203] address the replacement of scalar pointers with safe references. Laertes relies on feedback from the Rust compiler to determine which raw pointers can be converted to references. In contrast, Crown conducts ownership analysis for pointers and can convert a wider range of pointers into references compared to Laertes.

C to Safe Languages Many safe substitutes for C and (semi-) automatic translation to those languages have been proposed [157, 93, 141, 81, 72, 89, 177, 57, 90, 91]. However, they guarantee only a limited form of safety. Necula et al. [157] proposed CCured, a language extending C with new kinds of pointer and the type system to statically verify or dynamically enforce the safety of each pointer. Grossman et al. [93] proposed Cyclone, which extends C with region-based memory management [190]. Cyclone's type system prevents dangling pointer dereference by annotating each pointer with a region to use the pointer. Machiry et al.'s **3C** tool [141] automatically translates C to Checked C [81], an extension of C with checked pointer types. Checked pointers guarantee the absence of null pointer dereference and out-of-bound accesses.

Static Analysis for Rust A few static analysis techniques have been proposed for Rust, primarily focusing on bug detection. Their goal is to analyze human-written Rust code, unlike this work, which deals with C2Rust-generated code. A notable study is MirChecker [135], which employs a static analysis based on the abstract interpretation framework. It specifically targets the detection of unique bug patterns in Rust programs, such as runtime panics and lifetime corruption, but does not introduce novel sensitivities. Additionally, Rudra [58] and SafeDrop [75] perform dataflow analyses to detect memory bugs.

Static Analysis for Concurrent Programs Various static analyzers for concurrent C programs have been proposed. Our analyzer used in Concrat analyzes concurrent Rust programs but assumes only C2Rust-generated code, which uses C features rather than Rust features and thus shares characteristics with C programs. While the goal of existing analyzers is to detect concurrency bugs, our goal is to efficiently generate lock summaries for code transformation. This different goal makes our design completely distinct from the others. Our analyzer targets the same precision as the Rust type checker by using context- and path-insensitive dataflow analyses. However, other analyzers perform context- or path-sensitive analyses to reduce false alarms. Goblint [192, 180] and others [155, 154, 153] are based on abstract interpretation [73]. RELAY [193] and SDRacer [196] utilize symbolic execution [61]. Locksmith [169] and Kahlon et al's tool [116] perform context-sensitive analyses.

A few static analyzers for concurrent Java programs [134, 65] exist. Because Java provides a syntactic `synchronized` block, a lock acquired by a certain function cannot be released by another function, which frequently happens in C.

Must-Points-To Analysis Researchers have studied must-points-to analysis, but their techniques differ from our analysis in Urcrat in purpose, method, and target language. Our analysis aims to precisely compute struct field values. In contrast, most studies use must-points-to analysis to enhance the precision of other analyses. Altucher and Landi [51] use must-points-to relations to compute def-use relations. Ma et al. [140] and Fink et al. [88] improve may-points-to relations using must-points-to relations, enabling more precise detection of null pointer dereferences and Java typestate checking, respectively. Nikolić and Spoto [162] use must-points-to analysis to improve other analyses, such as nullness and termination analyses. While our approach uses may-points-to analysis as a pre-analysis, some techniques compute may- and must-points-to relations simultaneously. Emami et al. [82] conduct interprocedural may- and must-points-to analysis for compiler optimizations and parallelizations. Sagiv et al. [178] propose a parametric framework that generates a family of shape analyses based on three-valued logic, expressing must-, must-not-, and may-points-to relations. Unlike most studies, including ours, which focus on imperative languages, Jagannathan et al. [109] propose must-points-to analysis for functional languages, with results used for optimizations such as closure conversion. Techniques applicable to general must-points-to analysis have also been explored. Balatsouras et al. [60] present a declarative model in Datalog that expresses a wide range of must-points-to analyses. Kastrinis et al. [118] propose an efficient data structure for pointer graphs.

Sensitivities Various sensitivities have been proposed in the abstract interpretation literature. While these sensitivities aim to improve the precision of static analysis in a general setting, our write set sensitivity and nullity sensitivity used in Nopcrat are specifically designed to enhance the precision of output parameter identification. Call-site sensitivity [181, 182, 150] distinguishes calls to the same function from different call sites. Since output parameters should act as output parameters regardless of the calling context, our approach does not employ call-site sensitivity. On the other hand, abstract call state sensitivity [181] is often utilized for bottom-up analyses to distinguish various abstract input states of functions. Our nullity sensitivity can be viewed as a restricted form of abstract call state sensitivity, focusing solely on nullity information within the abstract input state. Trace history sensitivity, or trace partitioning [146, 173, 96], distinguishes different execution trace histories. Our write set sensitivity can be considered a restricted form of trace history sensitivity, specifically concerned with the history of effective writes to parameters. Object sensitivity [151, 152, 184] is commonly employed in static analysis for object-oriented languages. However, as neither C nor Rust is object-oriented, object sensitivity is not a consideration in our work. For a formal understanding of sensitivities, refer to the study by Kim et al. [121].

Neural Machine Translation of Programming Languages Neural machine translation of programming languages has been extensively studied over the past decade. Most existing studies have focused on training models to translate code without considering the integration of additional information and guidance, which is the primary focus of this work. Supervised learning approaches, which rely on code translation data for training, have been applied to only a limited number of language pairs, such as Java and C# [160, 117, 161, 71]. To address this limitation, TransCoder [175] introduces unsupervised learning to programming language translation, enabling training with monolingual code bases. Further studies [126, 176, 188] enhance TransCoder's translation capabilities by utilizing obfuscated code, unit tests, and compilers' intermediate code representation during training. SDA-Trans [138] also employs unsupervised learning but leverages syntax structure and domain knowledge to allow effective

learning even with limited training data. TransCoder, its successors, and SDA-Trans primarily focus on translating small programs containing one or two functions. Consequently, the need for augmenting functions with callee signatures has not been motivated. There exist several LLMs capable of code translation [197, 70, 87, 95], which can be effectively leveraged by Tymcrat.

Only a few studies have focused on utilizing pre-trained LLMs for code translation. Notable among them is UniTrans [201], which, like this work, provides additional information and guidance to LLMs. UniTrans augments each function with generated unit tests before feeding it into the LLM and iteratively requests the LLM to fix the translated code if it does not pass the unit tests. Since UniTrans targets small programs without complex types, it does not aim to migrate types. Furthermore, because type errors rarely occur after translating such programs, UniTrans focuses on fixing semantics errors rather than type errors. Pan et al. [166] categorize incorrect code translation results produced by LLMs. They conducted experiments with multiple source and target programming languages. Their categorization covers various kinds of translation bugs, including both type errors and runtime errors. In contrast, this work focuses specifically on translating C to Rust and aims to reduce type errors. A promising future direction for this work would be to use a methodology similar to Pan et al.'s for classifying unresolved type errors. This would help develop techniques to fix each kind of error.

While we evaluate Tymcrat with the number of type errors, various metrics have been proposed. BLEU, which treats code as a token sequence and measures syntactic similarity between translated code and human translation, has been used by several studies [160, 117, 161, 71]. CodeBLEU [171], a recently introduced metric, enhances BLEU by considering the similarity of syntax trees and dataflow graphs. Roziere et al. [175] proposed computational accuracy, which verifies if the translated code preserves the semantics through the execution of unit tests. We do not empoly existing metrics due to the absence of human-translated Rust code for the benchmark programs and the presence of type errors preventing compilation and unit test execution.

Modular Abstractions for A&M Patterns Various A&M patterns and their modular abstractions in Rust have already been proposed outside the context of OSs. For example, Rust's standard library [26] and intrusive collection libraries [130, 77] provide modular abstractions for well-known A&M patterns. Several cells such as `GhostCell` [202], `QCell`, `TCell`, `TLCell`, and `LCell` [168] have been proposed, but they are different from `ArcCell`. The `spin` [191] and `parking_lot` [78] crates have proposed OS-independent synchronization primitives, but not lock-protected immovable values or separated values.

Existing clean-slate Rust OSs rarely propose new modular abstractions for A&M patterns. They primarily utilize A&M patterns in the standard library to guarantee safety straightforwardly. Theseus [67] supports live evolution—changing modules while the kernel is running—and fault recovery without introducing new A&M patterns. RedLeaf [156] also features fault isolation of modules but introduces a new A&M pattern, `RRef<T>`, for safe zero-copy communication between modules. `RRef<T>` is the same as `Box<T>` [33] or C++'s `unique_ptr<T>` [108, §23.10.2] except that it counts the number of borrows to defer its deallocation in case of its owner module's fault. `RRef<T>` is not necessary for xv6_{Rust} because it is a monolithic kernel without isolated modules.

Chapter 8. Conclusion

In this dissertation, we propose techniques to improve C-to-Rust translation using static analysis. For each C feature, we design (1) specialized static analysis to gather information on how the feature is used in C2Rust-generated code and (2) code transformation that replaces the feature with its Rust counterpart based on the analysis results. Specifically, we target three key features: locks, tagged unions, and output parameters. To replace C's locks with Rust's locks, we compute data-lock relations and flow-lock relations through bottom-up dataflow analysis and top-down data fact propagation. The analysis results allow us to merge each lock with the protected data and properly pass guards through function calls. To replace unions that are accompanied by tag values with tagged unions, we employ intraprocedural must-points-to analysis, which identifies the tag field for each union and the tag values associated with each union field. Using the analysis results, we update type definitions and apply two kinds of transformations, naïve and idiomatic, to the use sites of unions. To replace output parameters with tuples and the `Option`/`Result` types, we introduce write-set-sensitive and nullity-sensitive bottom-up dataflow analysis, which identifies output parameters and the return values signifying success or failure. The analysis results facilitate the replacement of must-output parameters with tuples and may-output parameters with `Option`/`Result`. Our evaluation shows that the proposed approaches are scalable, precise, and mostly correct.

Additionally, to complement the static-analysis-based translation, we propose type-migrating LLM-based translation. To promote type migration, we explicitly instruct the LLM to generate candidate signatures and translate the function using each candidate. To reduce type errors, we iteratively fix them using compiler feedback, leveraging compiler-generated fixes and providing the error messages from the compiler to the LLM. Our evaluation shows that the proposed techniques effectively promote type migration and reduce type errors.

Furthermore, to facilitate future development of static analysis targeting A&M patterns in OSs, we identify six A&M patterns and propose their modular abstractions. The six patterns are process-owned values, CPU-owned values, memory pools, lock-protected immovable values, lock-protected separated values, and asynchronous ownership transfer for DMA. Our evaluation shows that these patterns are all present in Linux, implying practicality, but the proposed modular abstractions do not currently exist in clean-slate Rust OSs, demonstrating originality.

Bibliography

[1] C2Rust. https://github.com/immunant/c2rust.

[2] Crate libc. https://docs.rs/libc.

[3] Crate reffers. https://docs.rs/reffers.

[4] Crate rental. https://docs.rs/rental.

[5] Fuchsia guides: Rust. https://fuchsia.dev/fuchsia-src/development/languages/rust.

[6] GNU package blurbs. https://www.gnu.org/manual/blurbs.html.

[7] Goblint documentation: configuring. https://goblint.readthedocs.io/en/latest/user-guide/configuring.

[8] Miri. https://github.com/rust-lang/miri.

[9] OpenAI documentation: Models. https://platform.openai.com/docs/models.

[10] Redox. https://redox-os.org/.

[11] Rust by example: if let. https://doc.rust-lang.org/rust-by-example/flow_control/if_let.html.

[12] Rust compiler development guide. https://rustc-dev-guide.rust-lang.org/.

[13] Rust compiler development guide: Dataflow analysis. https://rustc-dev-guide.rust-lang.org/mir/dataflow.html.

[14] Rust compiler development guide: The HIR. https://rustc-dev-guide.rust-lang.org/hir.html.

[15] Rust compiler development guide: The MIR. https://rustc-dev-guide.rust-lang.org/mir/index.html.

[16] The Rust programming language. http://rust-lang.org/.

[17] The Rust programming language: Defining an enum. https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html.

[18] The Rust programming language: Pattern syntax. https://doc.rust-lang.org/book/ch18-03-pattern-syntax.html.

[19] The Rust programming language: Unsafe Rust. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.

[20] The Rust reference: Behavior considered undefined. https://doc.rust-lang.org/reference/behavior-considered-undefined.html.

[21] The Rust reference: Lifetime elision. https://doc.rust-lang.org/reference/lifetime-elision.html.

[22] The Rust reference: Static items—mutable statics. https://doc.rust-lang.org/reference/items/static-items.html#mutable-statics.

[23] The Rust reference: Unions. https://doc.rust-lang.org/reference/items/unions.html.

[24] The Rust RFC book: 2094-nll. https://rust-lang.github.io/rfcs/2094-nll.html.

[25] The Rust RFC book: 2585-unsafe-block-in-unsafe-fn. https://rust-lang.github.io/rfcs/2585-unsafe-block-in-unsafe-fn.html.

[26] The Rust standard library. https://doc.rust-lang.org/std.

[27] The Rust standard library: Module std::option. https://doc.rust-lang.org/std/option/.

[28] The Rust standard library: Module std::pin. https://doc.rust-lang.org/std/pin/index.html.

[29] The Rust standard library: Module std::result. https://doc.rust-lang.org/std/result/index.html.

[30] The Rust standard library: Module std::sync. https://doc.rust-lang.org/stable/std/sync/index.html.

[31] The Rust standard library: Primitive type never. https://doc.rust-lang.org/std/primitive.never.html.

[32] The Rust standard library: Primitive type tuple. https://doc.rust-lang.org/std/primitive.tuple.html.

[33] The Rust standard library: Struct std::boxed::Box. https://doc.rust-lang.org/std/boxed/struct.Box.html.

[34] The Rust standard library: Struct std::cell::Cell. https://doc.rust-lang.org/std/cell/struct.Cell.html.

[35] The Rust standard library: Struct std::cell::RefCell. https://doc.rust-lang.org/std/cell/struct.RefCell.html.

[36] The Rust standard library: Struct std::rc::Rc. https://doc.rust-lang.org/std/rc/struct.Rc.html.

[37] The Rust standard library: Struct std::sync::Arc. https://doc.rust-lang.org/std/sync/struct.Arc.html.

[38] The Rust standard library: Struct std::sync::Condvar. https://doc.rust-lang.org/stable/std/sync/struct.Condvar.html.

[39] The Rust standard library: Struct std::sync::Mutex. https://doc.rust-lang.org/std/sync/struct.Mutex.html.

[40] The Rust standard library: Struct std::sync::RwLock. https://doc.rust-lang.org/stable/std/sync/struct.RwLock.html.

[41] std::lock_guard. https://en.cppreference.com/w/cpp/thread/lock_guard.

[42] std::mutex. https://en.cppreference.com/w/cpp/thread/mutex.

[43] Ubuntu 18.04 LTS (Bionic Beaver). https://releases.ubuntu.com/18.04.

[44] The Rust unstable book: thread_local. https://doc.rust-lang.org/beta/unstable-book/language-features/thread-local.html, 2015.

[45] Netstack3 - a Fuchsia owned rust based netstack. https://fuchsia.dev/fuchsia-src/contribute/roadmap/2021/netstack3, 2021.

[46] Introducing ChatGPT. https://openai.com/blog/chatgpt, 2022.

[47] GPT-4o mini: advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/, 2024.

[48] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. Concurrency bugs in open source software: a case study. *Journal of Internet Services and Applications*, 8(1), April 2017.

[49] Toufique Ahmed and Premkumar Devanbu. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[50] Eyad Alkassar, Mark A. Hillebrand, Wolfgang J. Paul, and Elena Petrova. Automated verification of a small hypervisor. In *Proceedings of the Third International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE'10, page 40–54, Berlin, Heidelberg, 2010. Springer-Verlag.

[51] Rita Z. Altucher and William Landi. An extended form of must alias analysis for dynamic allocation. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 74–84, New York, NY, USA, 1995. Association for Computing Machinery.

[52] Lars Ole Andersen. Program analysis and specialization for the C programming language. *PhD Thesis, University of Copenhagen*, 1994.

[53] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. Engineering the Servo web browser engine using Rust. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, page 81–89, New York, NY, USA, 2016. Association for Computing Machinery.

[54] Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar Sarma. Using read-copy-update techniques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*, pages 297–309. USENIX, 2003.

[55] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use Unsafe Rust? *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[56] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging Rust types for modular specification and verification. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[57] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 290–301, New York, NY, USA, 1994. Association for Computing Machinery.

[58] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding memory safety bugs in Rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 84–99, New York, NY, USA, 2021. Association for Computing Machinery.

[59] Xiaolong Bai, Luyi Xing, Min Zheng, and Fuping Qu. iDEA: Static analysis on the security of Apple kernel drivers. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1185–1202, New York, NY, USA, 2020. Association for Computing Machinery.

[60] George Balatsouras, Kostas Ferles, George Kastrinis, and Yannis Smaragdakis. A Datalog model of must-alias analysis. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2017, page 7–12, New York, NY, USA, 2017. Association for Computing Machinery.

[61] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), May 2018.

[62] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, and Chetan Murthy. The Coq proof assistant reference manual: Version 6.1. https://hal.inria.fr/inria-00069968, 1997.

[63] Alexis Kenneth Beingessner. You can't spell trust without Rust. Master's thesis, Carleton University, 1125 Colonel By Drive, Ottawa, ON, K1S 5B6, Canada, 2016.

[64] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

[65] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. RacerD: compositional static race detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[66] Jeff Bonwick. The slab allocator: an object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, page 6, USA, 1994. USENIX Association.

[67] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: An experiment in operating system structure and state management. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[68] Kyle Bradshaw. Google is officially releasing its Fuchsia OS, starting w/ first-gen Nest Hub. https://9to5google.com/2021/05/25/google-releases-fuchsia-os-nest-hub, 2021.

[69] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, New York, NY, USA, 2011. Association for Computing Machinery.

[70] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. https://arxiv.org/abs/2107.03374, 2021.

[71] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 2552–2562, Red Hook, NY, USA, 2018. Curran Associates Inc.

[72] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proceedings of the 16th European Symposium on Programming*, ESOP'07, page 520–535, Berlin, Heidelberg, 2007. Springer-Verlag.

[73] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[74] Russ Cox, M Frans Kaashoek, and Robert Morris. xv6, a simple Unix-like teaching operating system. http://pdos.csail.mit.edu/6.828/2012/xv6.html, 2011.

[75] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. SafeDrop: Detecting memory deallocation bugs of Rust programs via static data-flow analysis. *ACM Trans. Softw. Eng. Methodol.*, 32(4), may 2023.

[76] Al Danial. cloc. https://github.com/AlDanial/cloc.

[77] Amanieu d'Antras. Crate intrusive_collections. https://docs.rs/intrusive-collections.

[78] Amanieu d'Antras. Crate parking_lot. https://docs.rs/parking_lot.

[79] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via ChatGPT. https://arxiv.org/abs/2304.07590, 2024.

[80] Ulrich Drepper. ELF handling for thread-local storage. https://www.akkadia.org/drepper/tls.pdf.

[81] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked C: Making C safe by extension. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 53–60, 2018.

[82] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 242–256, New York, NY, USA, 1994. Association for Computing Machinery.

[83] Mehmet Emre, Peter Boyland, Aesha Parekh, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Aliasing limits on translating C to safe Rust. *Proc. ACM Program. Lang.*, 7(OOPSLA1), apr 2023.

[84] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Translating C to safer Rust. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[85] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1469–1481, Melbourne, Victoria, Australia, 2023. IEEE Press.

[86] Mazdak Farrokhzad. Tracking issue for eRFC 2497, "if- and while-let-chains, take 2". https://github.com/rust-lang/rust/issues/53667, 2018.

[87] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. https://arxiv.org/abs/2002.08155, 2020.

[88] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), may 2008.

[89] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, November 2006.

[90] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 313–323, New York, NY, USA, 1998. Association for Computing Machinery.

[91] David Gay and Alex Aiken. Language support for regions. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, page 70–80, New York, NY, USA, 2001. Association for Computing Machinery.

[92] Manish Goregaokar. Fearless concurrency in Firefox Quantum. https://blog.rust-lang.org/2017/11/14/Fearless-Concurrency-In-Firefox-Quantum.html, 2017.

[93] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, page 282–293, New York, NY, USA, 2002. Association for Computing Machinery.

[94] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: an extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 653–669, USA, 2016. USENIX Association.

[95] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training code representations with data flow. https://arxiv.org/abs/2009.08366, 2021.

[96] Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In Giorgio Levi, editor, *Static Analysis*, pages 200–214, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[97] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, page 226–238, New York, NY, USA, 2009. Association for Computing Machinery.

[98] John Hodge. Tifflin experimental kernel. https://github.com/thepowersgang/rust_os/blob/master/Kernel/Core/arch/mod.rs.

[99] Jaemin Hong and Sukyoung Ryu. Tymcrat: Type-migrating C-to-Rust automatic translator. https://github.com/kaist-plrg/simcrat.

[100] Jaemin Hong and Sukyoung Ryu. Concrat: An automatic C-to-Rust lock API translator for concurrent programs (artifact). https://doi.org/10.5281/zenodo.7573490, January 2023.

[101] Jaemin Hong and Sukyoung Ryu. Don't write, but return: Replacing output parameters with algebraic data types in C-to-Rust translation (artifact). https://doi.org/10.5281/zenodo.10795858, March 2024.

[102] Jaemin Hong and Sukyoung Ryu. To tag, or not to tag: Translating C's unions to Rust's tagged unions (artifact). https://doi.org/10.5281/zenodo.13373683, August 2024.

[103] Jaemin Hong, Sunghwan Shim, Sanguk Park, Taewoo Kim, Jungwoo Kim, Junsoo Lee, Sukyoung Ryu, and Jeehoon Kang. $xv6_{Rust}$, a complete rewrite of xv6 in Rust. https://github.com/kaist-cp/rv6.

[104] White House. Back to the building blocks: A path toward secure and measurable software. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf, February 2024.

[105] Tim Hutt. Would Rust secure cURL? https://blog.timhutt.co.uk/curl-vulnerabilities-rust/, 2021.

[106] IEEE. IEEE standard for information technology–portable operating system interface (POSIX(TM)) base specifications, issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pages 1–3951, 2018.

[107] Ayooluwa Isaiah. Rewriting the GNU Coreutils in Rust. https://lwn.net/Articles/857599, 2021.

[108] ISO. Programming languages — C++. Standard ISO/IEC 14882:2020, International Organization for Standardization, Chemin de Blandonnet 8, CP 401 - 1214 Vernier, Geneva, Switzerland, 2020.

[109] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: must-alias analysis for higher-order languages. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, page 329–341, New York, NY, USA, 1998. Association for Computing Machinery.

[110] Sujay Jayakar. Rewriting the heart of our sync engine. https://dropbox.tech/infrastructure/rewriting-the-heart-of-our-sync-engine, 2020.

[111] Dave Jones. Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity.

[112] Ralf Jung. Exclude mutable references to !Unpin types from uniqueness guarantees. https://github.com/rust-lang/miri/pull/1952.

[113] Ralf Jung. Stacked Borrows vs self-referential structs. https://github.com/rust-lang/unsafe-code-guidelines/issues/148.

[114] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked Borrows: An aliasing model for Rust. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[115] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[116] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, page 13–22, New York, NY, USA, 2009. Association for Computing Machinery.

[117] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, page 173–184, New York, NY, USA, 2014. Association for Computing Machinery.

[118] George Kastrinis, George Balatsouras, Kostas Ferles, Nefeli Prokopaki-Kostopoulou, and Yannis Smaragdakis. An efficient data structure for must-alias analysis. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, page 48–58, New York, NY, USA, 2018. Association for Computing Machinery.

[119] Uday P. Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009.

[120] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid fuzzing on the Linux kernel. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[121] Se-Won Kim, Xavier Rival, and Sukyoung Ryu. A theoretical foundation of sensitivity in an abstract interpretation framework. *ACM Trans. Program. Lang. Syst.*, 40(3), aug 2018.

[122] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. *Proceedings of the Linux symposium*, 1(8):225–230, 2007.

[123] Aleksey Kladov. Spinlocks considered harmful. https://matklad.github.io/2020/01/02/spinlocks-considered-harmful.html, 2020.

[124] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[125] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2024. Curran Associates Inc.

[126] Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. DOBF: a deobfuscation pre-training objective for programming languages. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21, Red Hook, NY, USA, 2024. Curran Associates Inc.

[127] Stefan Lankes, Jens Breitbart, and Simon Pickartz. Exploring Rust for unikernel development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, PLOS '19, page 8–15, New York, NY, USA, 2019. Association for Computing Machinery.

[128] Stefan Lankes, Jonathan Klimt, Jens Breitbart, and Simon Pickartz. RustyHermit: A scalable, Rust-based virtual execution environment. In *High Performance Computing: ISC High Performance 2020 International Workshops, Frankfurt, Germany, June 21–25, 2020, Revised Selected Papers*, page 331–342, Berlin, Heidelberg, 2020. Springer-Verlag.

[129] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp Symb. Comput.*, 8(4):293–341, December 1995.

[130] Keunhong Lee, Jeehoon Kang, Wonsup Yoon, Joongi Kim, and Sue Moon. Enveloping implicit assumptions of intrusive data structures within ownership type system. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, PLOS '19, page 16–22, New York, NY, USA, 2019. Association for Computing Machinery.

[131] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kB computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 234–251, New York, NY, USA, 2017. Association for Computing Machinery.

[132] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. https://arxiv.org/abs/2305.06599, 2023.

[133] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. AceCoder: Utilizing existing code to enhance code generation. https://arxiv.org/abs/2303.17780, 2023.

[134] Yanze Li, Bozhen Liu, and Jeff Huang. SWORD: a scalable whole program race detector for Java. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ICSE '19, page 75–78, Montreal, Quebec, Canada, 2019. IEEE Press.

[135] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. MirChecker: Detecting bugs in Rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2183–2196, New York, NY, USA, 2021. Association for Computing Machinery.

[136] Yi Lin, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. Rust as a language for high performance GC implementation. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, page 89–98, New York, NY, USA, 2016. Association for Computing Machinery.

[137] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R. Cordy, and Ahmed E. Hassan. In Rust we trust: A transpiler from unsafe C to safer Rust. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ICSE '22, page 354–355, New York, NY, USA, 2022. Association for Computing Machinery.

[138] Fang Liu, Jia Li, and Li Zhang. Syntax and domain aware model for unsupervised program translation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 755–767, 2023.

[139] Marvin Löbel. Crate owning_ref. https://docs.rs/owning_ref.

[140] Xiaodong Ma, Ji Wang, and Wei Dong. Computing must and may alias to detect null pointer dereference. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation*, pages 252–261, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[141] Aravind Machiry, John Kastner, Matt McCutchen, Aaron Eline, Kyle Headley, and Michael Hicks. C to Checked C by 3C. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022.

[142] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR. Checker: a soundy analysis for Linux kernel drivers. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 1007–1024, USA, 2017. USENIX Association.

[143] Joshua Maros. Crate ouroboros. https://docs.rs/ouroboros.

[144] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson, 1st edition, Aug 2008.

[145] Nicholas D. Matsakis and Felix S. Klock. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery.

[146] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of the 14th European Conference on Programming Languages and Systems*, ESOP'05, page 5–20, Berlin, Heidelberg, 2005. Springer-Verlag.

[147] Larry McVoy and Carl Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, page 23, USA, 1996. USENIX Association.

[148] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Roziere, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. Augmented language models: a survey. https://arxiv.org/abs/2302.07842, 2023.

[149] Jan Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3), jun 2012.

[150] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 305–315, New York, NY, USA, 2010. Association for Computing Machinery.

[151] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, page 1–11, New York, NY, USA, 2002. Association for Computing Machinery.

[152] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, jan 2005.

[153] Antoine Miné. Static analysis of run-time errors in embedded critical parallel C programs. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 398–418, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[154] Antoine Miné. Relational thread-modular static value analysis by abstract interpretation. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 39–58, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[155] Antoine Miné, Laurent Mauborgne, Xavier Rival, Jerome Feret, Patrick Cousot, Daniel Kästner, Stephan Wilhelm, and Christian Ferdinand. Taking static analysis to the next level: proving the absence of run-time errors and data races with Astrée. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[156] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[157] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, page 128–139, New York, NY, USA, 2002. Association for Computing Machinery.

[158] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 252–269, New York, NY, USA, 2017. Association for Computing Machinery.

[159] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 457–468, New York, NY, USA, 2014. Association for Computing Machinery.

[160] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 651–654, New York, NY, USA, 2013. Association for Computing Machinery.

[161] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Divide-and-conquer approach for multi-phase statistical migration for source code. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, page 585–596, Lincoln, Nebraska, 2015. IEEE Press.

[162] Đurica Nikolić and Fausto Spoto. Definite expression aliasing analysis for Java bytecode. In Abhik Roychoudhury and Meenakshi D'Souza, editors, *Theoretical Aspects of Computing – ICTAC 2012*, pages 74–89, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[163] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[164] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 229–238, New York, NY, USA, 2012. Association for Computing Machinery.

[165] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2024. Curran Associates Inc.

[166] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[167] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst.*, 30(1):4–es, nov 2007.

[168] Jim Peters. Crate qcell. https://docs.rs/qcell.

[169] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, page 320–331, New York, NY, USA, 2006. Association for Computing Machinery.

[170] ChengXiang Qi, Yu Chen, and Fengjie Li. xv6-rust. https://github.com/Ko-oK-OS/xv6-rust.

[171] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: a method for automatic evaluation of code synthesis. https://arxiv.org/abs/2009.10297, 2020.

[172] H. Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.

[173] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5):26–es, aug 2007.

[174] Stephan Roth. *Clean C++20: Sustainable Software Development Patterns and Best Practices*. Apress, 2 edition, Jun 2021.

[175] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[176] Baptiste Rozière, Jie Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[177] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. Achieving safety incrementally with Checked C. In Flemming Nielson and David Sands, editors, *Principles of Security and Trust*, pages 76–98, Cham, 2019. Springer International Publishing.

[178] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, page 105–118, New York, NY, USA, 1999. Association for Computing Machinery.

[179] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 167–182, USA, 2017. USENIX Association.

[180] Michael Schwarz, Simmo Saan, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. Improving thread-modular abstract interpretation. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings*, page 359–383, Berlin, Heidelberg, 2021. Springer-Verlag.

[181] Micha Sharir and Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*, chapter 7. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[182] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, page 164–174, New York, NY, USA, 1988. Association for Computing Machinery.

[183] Sergio De Simone. Linux 6.1 officially adds support for Rust in the kernel. https://www.infoq.com/news/2022/12/linux-6-1-rust, 2022.

[184] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 17–30, New York, NY, USA, 2011. Association for Computing Machinery.

[185] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Proceedings of the 6th International Conference on Compiler Construction*, CC '96, page 136–150, Berlin, Heidelberg, 1996. Springer-Verlag.

[186] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 32–41, New York, NY, USA, 1996. Association for Computing Machinery.

[187] Jeff Vander Stoep and Stephen Hines. Rust in the Android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html, 2021.

[188] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. Code translation with compiler representations. https://arxiv.org/abs/2207.03578, 2023.

[189] Gavin Thomas. A proactive approach to more secure code. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code, 2019.

[190] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[191] Mathijs van de Nes and Joshua Barretto. Crate spin. https://docs.rs/spin.

[192] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 391–402, New York, NY, USA, 2016. Association for Computing Machinery.

[193] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, page 205–214, New York, NY, USA, 2007. Association for Computing Machinery.

[194] Dmitry Vyukov. syzkaller - kernel fuzzer. https://github.com/google/syzkaller.

[195] Dan Wallach. Translating all C to Rust (Tractor). https://www.darpa.mil/program/translating-all-c-to-rust, 2024.

[196] Yu Wang, Linzhang Wang, Tingting Yu, Jianhua Zhao, and Xuandong Li. Automatic detection and validation of race conditions in interrupt-driven embedded software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 113–124, New York, NY, USA, 2017. Association for Computing Machinery.

[197] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. https://arxiv.org/abs/2109.00859, 2021.

[198] Frances Wingerter. C2Rust is back. https://immunant.com/blog/2022/06/back/, 2022.

[199] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1482–1494, Melbourne, Victoria, Australia, 2023. IEEE Press.

[200] Zhen Yang, Jacky Wai Keung, Zeyu Sun, Yunfei Zhao, Ge Li, Zhi Jin, Shuo Liu, and Yishu Li. Improving domain-specific neural code generation with few-shot meta-learning. *Information and Software Technology*, 166:107365, 2024.

[201] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. Exploring and unleashing the power of large language models in automated code translation. *Proc. ACM Softw. Eng.*, 1(FSE), jul 2024.

[202] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. GhostCell: Separating permissions from data in Rust. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.

[203] Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang. Ownership guided C to Rust translation. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 459–482, Cham, 2023. Springer Nature Switzerland.

[204] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 195–204, New York, NY, USA, 2010. Association for Computing Machinery.

Acknowledgments in Korean

가장 먼저 대학원 기간 동안 저를 지도해 주신 류석영 교수님께 깊은 감사의 인사를 올립니다. 교수님과의 인연은 2015년, 제가 고등학교 3학년이던 시절로 거슬러 올라갑니다. 교수님께서 학교에 방문해 즐겁게 강연하시는 모습에 이끌려 프로그래밍 언어 연구자의 길을 걷게 되었습니다. 그때부터 지금까지 언제나 열정적으로 지도해 주셨기에 제가 여기까지 올 수 있었습니다. 교수님께서 가르쳐 주신 지식, 조언해 주신 내용 하나하나가 제 성장의 밑거름이 되었습니다. 이 학위 논문 역시 교수님께서 지도해 주신 덕분에 탄생할 수 있었으며, 논문에 미흡한 부분이 있다 해도 이는 제가 부족한 탓입니다.

KAIST 프로그래밍 언어 연구실의 모든 동료에게 감사드립니다. 먼저 졸업한 선배님들 그리고 지금 저와 함께 지내고 있는 동기, 선후배 모두에게 감사합니다. 행복하게 지낼 수 있는 연구실 문화를 만들고 계속 유지해 준 덕에 어려운 대학원 생활을 즐겁게 헤쳐 나갈 수 있었습니다. 날마다 연구실에서 동료들과 나눈 다양한 연구 관련 이야기와 수많은 잡담이 연구를 하는 데 있어 좋은 길잡이가 되었습니다.

저를 낳고 지금까지 길러 주신 부모님께 감사드립니다. 어렸을 적부터 부모님께서 다양한 경험의 기회를 주신 덕분에 수학과 과학 그리고 전산학에 흥미를 가지게 되었습니다. 언제나 제 의견을 존중해 주시고 제가 하고 싶은 일을 하면서 살 수 있도록 이끌어 주셨기에 프로그래밍 언어 연구자가 될 수 있었습니다.

언제나 제 곁에서 저를 믿어 주고 지지해 주는 아내 임효진에게 감사합니다. 항상 저를 배려해 주고 많은 사랑을 아낌없이 베풀어 주기에 하루하루가 매일 행복하고 그 어떤 힘든 일도 이겨낼 수 있습니다.

마지막으로 가장 친한 친구인 현우 형에게 감사합니다. 고등학교와 학부 시절을 같이 보내며 즐거울 때도 슬플 때도 늘 함께하면서 힘이 되어 주었기에 제가 이만큼 성장할 수 있었습니다.

Curriculum Vitae in Korean

이 름: 홍 재 민

생 년 월 일: 1998년 09월 01일

학 력

2013. 3. – 2016. 2. 한국과학영재학교

2016. 3. – 2020. 2. 한국과학기술원 전산학부 (학사)

2020. 3. – 2025. 2. 한국과학기술원 전산학부 (박사)

경 력

2020. 3. – 2023. 8. 한국과학기술원 전산학부 조교

연 구 업 적

1. **Jaemin Hong**, Sunghwan Shim, Sanguk Park, Tae Woo Kim, Jungwoo Kim, Junsoo Lee, Sukyoung Ryu, and Jeehoon Kang. Taming shared mutable states of operating systems in Rust. *Science of Computer Programming*, 238:103152, 2024.
2. **Jaemin Hong** and Sukyoung Ryu. To tag, or not to tag: Translating C's unions to Rust's tagged unions. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 40–52, New York, NY, USA, 2024. Association for Computing Machinery.
3. **Jaemin Hong** and Sukyoung Ryu. Type-migrating C-to-Rust translation using a large language model. *Empirical Software Engineering*, 30(1), October 2024.
4. **Jaemin Hong** and Sukyoung Ryu. Don't write, but return: Replacing output parameters with algebraic data types in C-to-Rust translation. *Proc. ACM Program. Lang.*, 8(PLDI), jun 2024.
5. **Jaemin Hong**. Improving automatic C-to-Rust translation with static analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, page 273–277, 2023.
6. **Jaemin Hong** and Sukyoung Ryu. Concrat: An automatic C-to-Rust lock API translator for concurrent programs. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 716–728, Melbourne, Victoria, Australia, 2023. IEEE Press.
7. Jihee Park, Sungho Lee, **Jaemin Hong**, and Sukyoung Ryu. Static analysis of JNI programs via binary decompilation. *IEEE Transactions on Software Engineering*, 49(5):3089–3105, May 2023.

8. Jihee Park, **Jaemin Hong**, and Sukyoung Ryu. Semantic transformation framework for rewriting rules. In *Proceedings of the 2023 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation*, PEPM 2023, page 1–13, New York, NY, USA, 2023. Association for Computing Machinery.

9. Gyunghee Park, **Jaemin Hong**, Guy L. Steele Jr., and Sukyoung Ryu. Polymorphic symmetric multiple dispatch with variance. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

10. **Jaemin Hong**, Jihyeok Park, and Sukyoung Ryu. Path dependent types with path-equality. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, Scala 2018, page 35–39, New York, NY, USA, 2018. Association for Computing Machinery.