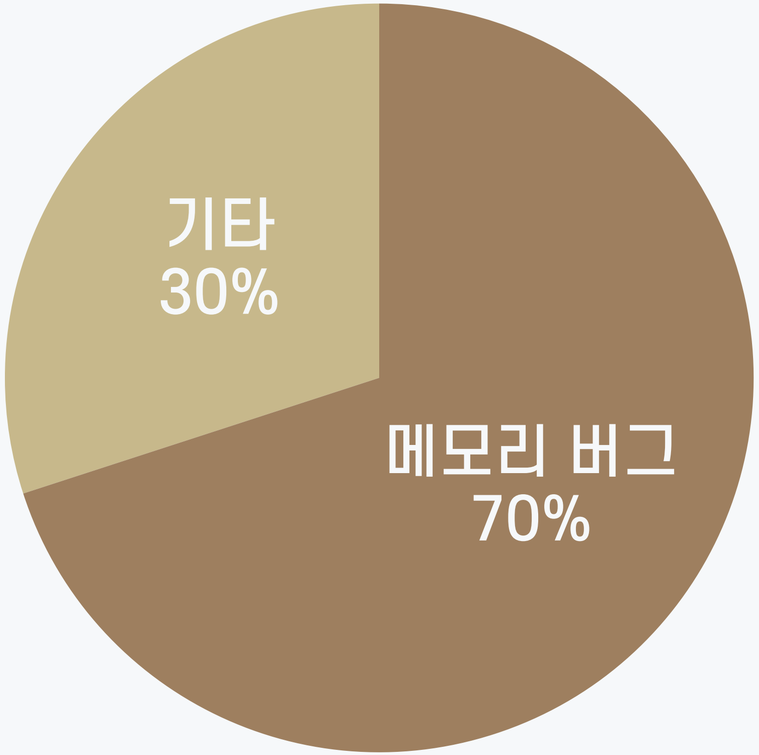
한국정보과학회 프로그래밍언어연구회 2026년 겨울학교

C를 러스트로 번역하기

분석으로, LLM으로, 아니면 함께?

홍재민(KAIST·예일대학교) | jaemin.hong@kaist.ac.kr

메모리 안전성을 보장하지 않는 C



마이크로소프트 코드베이스 보안 취약점의 원인

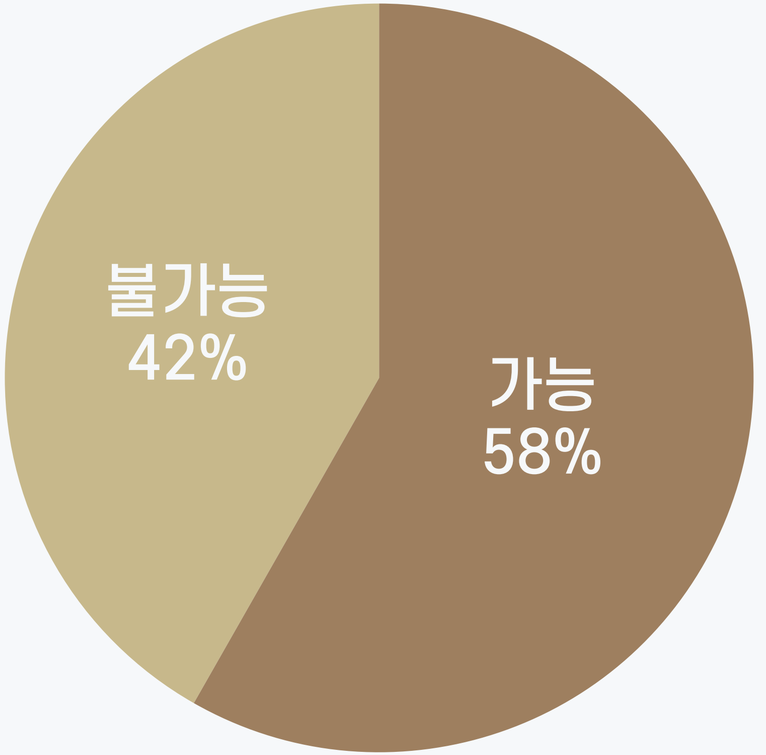
OpenSSL에서 발생한 Heartbleed 버그

Gavin Thomas, A Proactive Approach to More Secure Code.
Alex Hern, Heartbleed: Hundreds of Thousands of Servers at Risk from Catastrophic Bug.

메모리 안전성을 보장하는 러스트

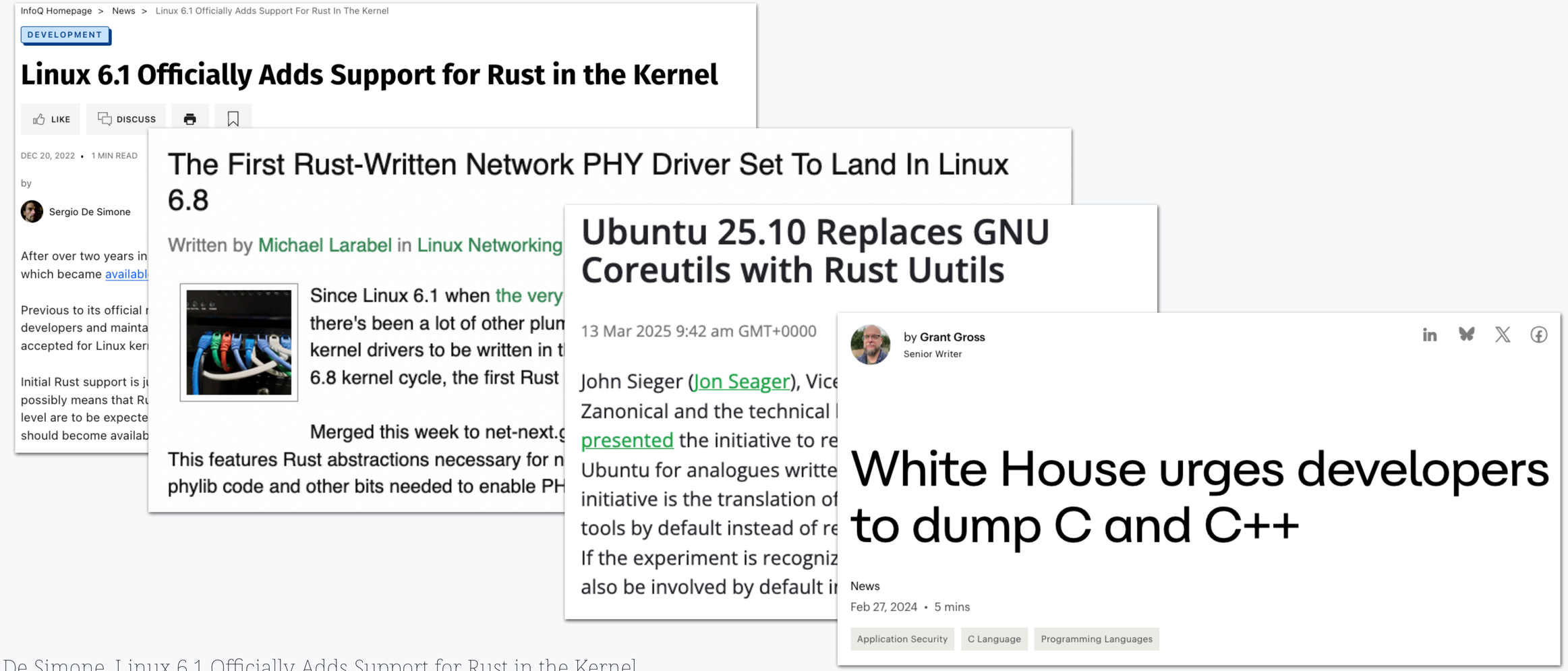
```rust
fn main() {
    let x = Box::new(0);
    foo(x);
    println!("{}", x);    borrow of moved value
}

fn foo(b: Box<i32>) {
    println!("{}", b);
}
```

러스트의 타입 검사



러스트가 얼마나 많은 cURL의 버그를 막을 수 있을까?

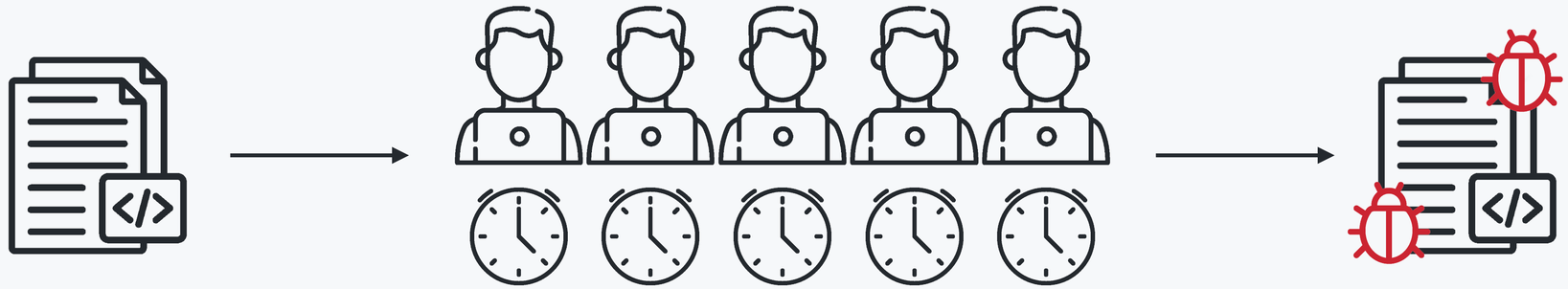
러스트를 통한 기존 시스템의 신뢰성 개선



Sergio De Simone, Linux 6.1 Officially Adds Support for Rust in the Kernel.
Michael Larabel, The First Rust-Written Network PHY Driver Set to Land in Linux 6.8.
Altus Intel, Ubuntu 25.10 Replaces GNU Coreutils with Rust Uutils.
Grant Gross, White House Urges Developers to Dump C and C++.

자동 번역기의 필요성

수작업 번역



자동 번역

자동 번역기의 필요성

xv6: a simple, Unix-like teaching operating system

Russ Cox Frans Kaashoek Robert Morris

August 31, 2020

때는 2021년...

Science of Computer Programming 238 (2024) 103152

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Taming shared mutable states of operating systems in Rust

Jaemin Hong [a,*], Sunghwan Shim [a], Sanguk Park [b], Tae Woo Kim [a], Jungwoo Kim [a], Junsoo Lee [a], Sukyoung Ryu [a], Jeehoon Kang [a]

[a] KAIST, 291 Daehak-ro, Yuseong-gu, 34141, Daejeon, Republic of Korea
[b] FuriosaAI, 145 Dosan-daero, Gangnam-gu, 06036, Seoul, Republic of Korea

ARTICLE INFO

Keywords:
Shared mutable state
Operating system
Rust

ABSTRACT

Operating systems (OSs) suffer from pervasive memory bugs. Their primary source is shared mutable states, crucial to low-level control and efficiency. The safety of shared mutable states is not guaranteed by C/C++, in which legacy OSs are typically written. Recently, researchers have adopted Rust into OS development to implement clean-slate OSs with fewer memory bugs. Rust ensures the safety of shared mutable states that follow the "aliasing XOR mutability" discipline via its type system. With the success of Rust in clean-slate OSs, the industry has become interested in rewriting legacy OSs in Rust. However, one of the most significant obstacles to this goal is shared mutable states that are *aliased AND mutable* (A&M). While they are essential to the performance of legacy OSs, Rust does not guarantee their safety. Instead, programmers have identified A&M states with the same reasoning principle dubbed an *A&M pattern* and implemented its modular abstraction to facilitate safety reasoning. This paper investigates modular abstractions for A&M patterns in legacy OSs. We present modular abstractions for six A&M patterns in the xv6 OS. Our investigation of Linux and clean-slate Rust OSs shows that the patterns are practical, as all of them are utilized in Linux, and the abstractions are original, as none of them are found in the Rust OSs. Using the abstractions, we implemented $xv6_{Rust}$, a complete rewrite of xv6 in Rust. The abstractions incur no run-time overhead compared to xv6 while reducing the reasoning cost of $xv6_{Rust}$ to the level of the clean-slate Rust OSs.

1. Introduction

Operating systems (OSs) suffer from pervasive memory bugs such as use-after-free and buffer overflow. For example, memory bugs are the source of two-thirds of the vulnerabilities in Linux [39] and about 70% of the CVE-assigned vulnerabilities in Microsoft's codebase [86].

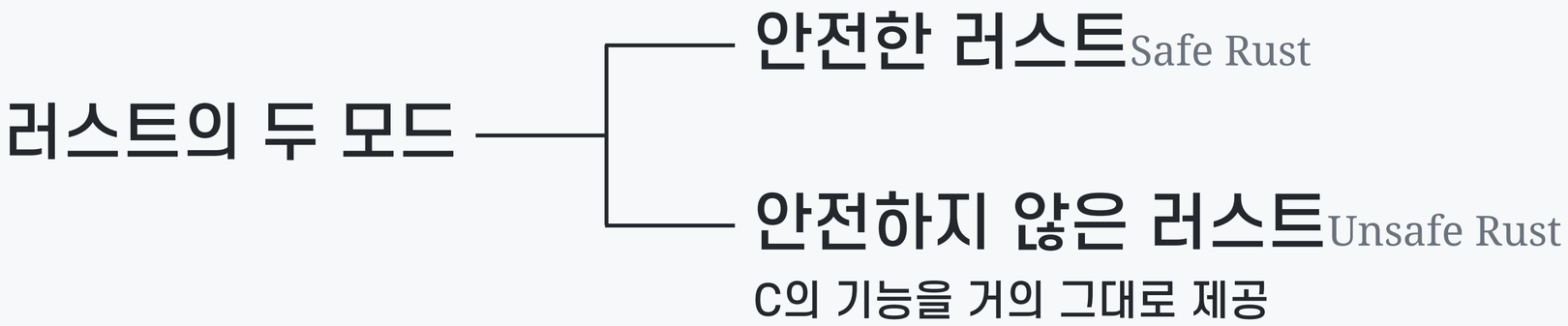
To reduce the number of memory bugs in OSs, various approaches have been proposed. Fuzzing, which generates random inputs to a program, has succeeded in finding bugs in OSs [58,63,83,89]. Static analysis approximates the behavior of a program without execution. Several static analysis tools target OSs [30,71]. Formal verification techniques allow verifying the correctness of a program with the help of proof assistants [32,77]. Recent improvement in the techniques realizes the verification of even an entire OS [26,50,65,76].

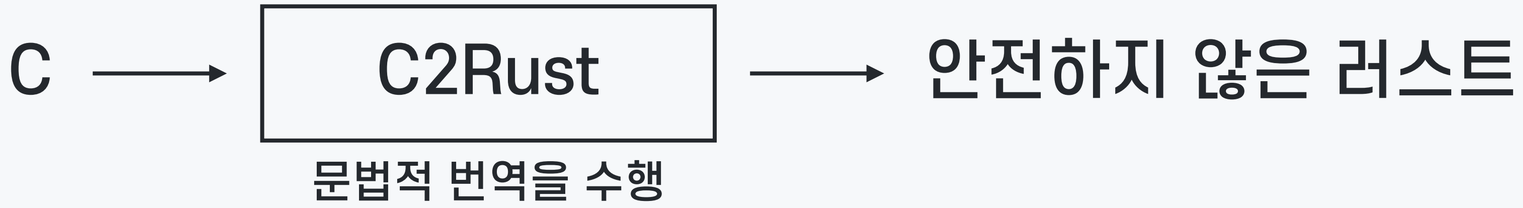
* Corresponding author.
E-mail addresses: jaemin.hong@kaist.ac.kr (J. Hong), sunghwan.shim@kaist.ac.kr (S. Shim), efenniht@furiosa.ai (S. Park), taewoo.kim99@kaist.ac.kr (T. Kim), jungwoo.kim@kaist.ac.kr (J. Kim), junsoo.lee97@kaist.ac.kr (J. Lee), sryu.cs@kaist.ac.kr (S. Ryu), jeehoon.kang@kaist.ac.kr (J. Kang).

https://doi.org/10.1016/j.scico.2024.103152
Received 10 January 2024; Received in revised form 15 April 2024; Accepted 24 May 2024
Available online 27 May 2024

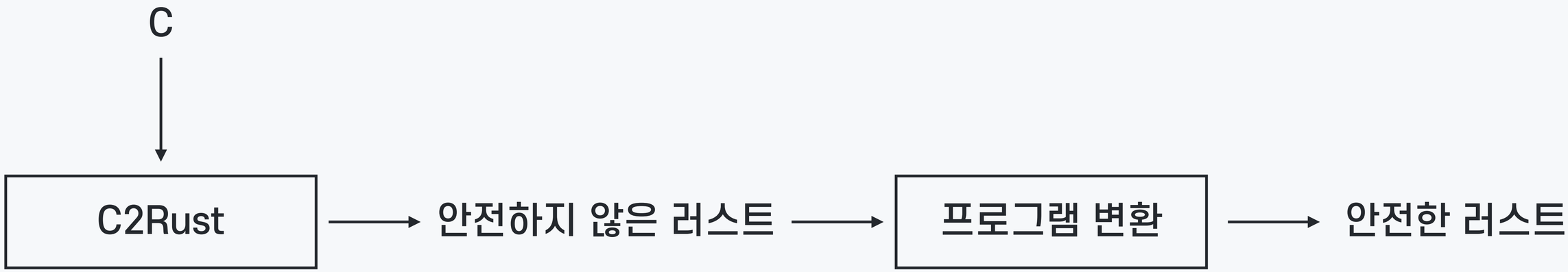
Hong et al., Taming Shared Mutable States of Operating Systems in Rust, SCICO 2024.

기존 번역기의 한계

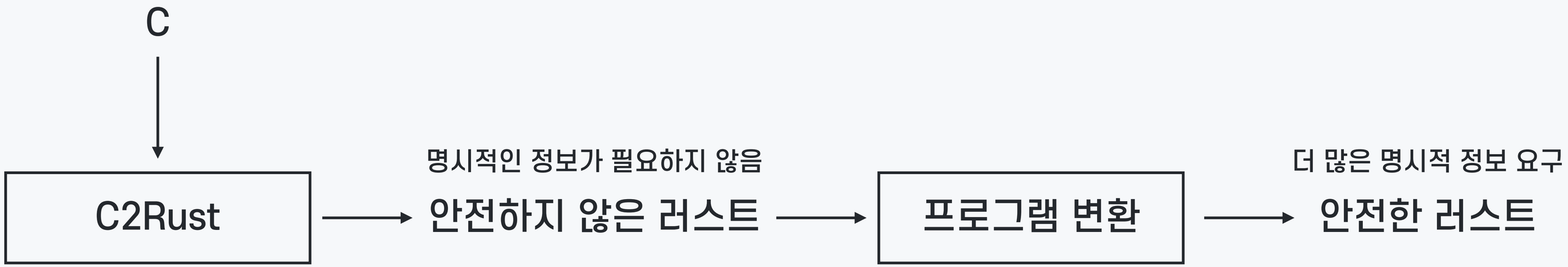

러스트의 두 모드
안전한 러스트Safe Rust
안전하지 않은 러스트Unsafe Rust
C의 기능을 거의 그대로 제공


C
C2Rust
안전하지 않은 러스트
문법적 번역을 수행

정적 분석을 통한 번역 개선

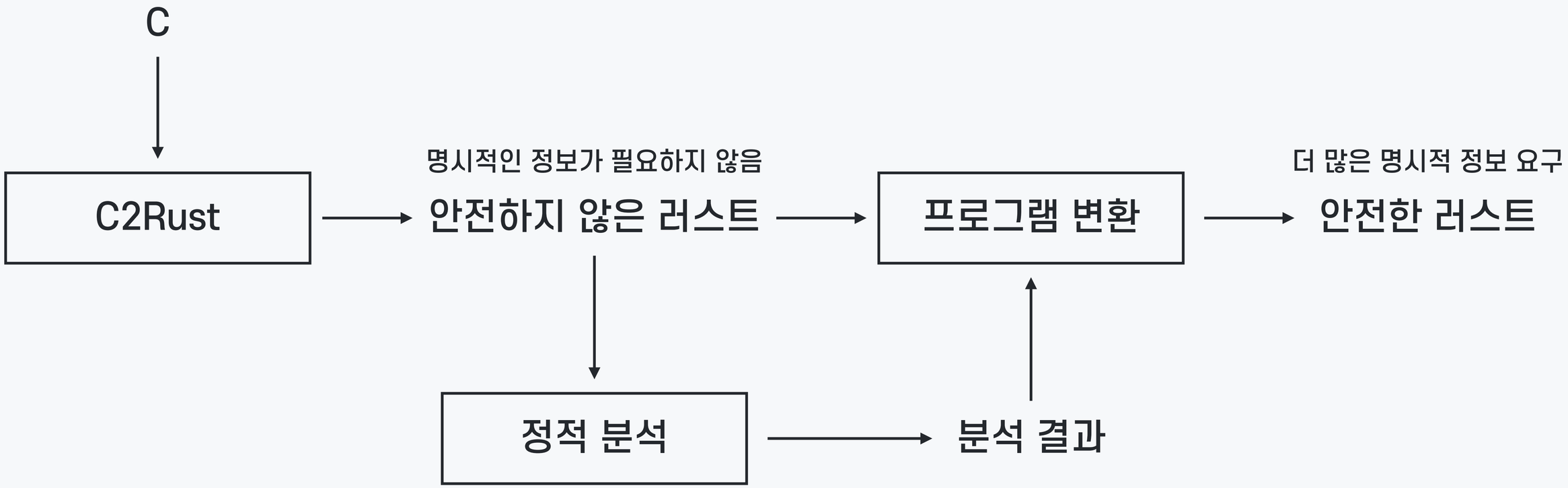


정적 분석을 통한 번역 개선



프로그램의 성질을 파악해 명시적인 정보를 알아내야 변환 가능

정적 분석을 통한 번역 개선



프로그램의 성질을 파악해 명시적인 정보를 알아내야 변환 가능

C의 포인터

`int *` 소유권ownership에 상관없이 같은 타입

메모리 할당을 해제deallocate할 권한

C

안전하지 않은 러스트Unsafe Rust

`*mut i32` 보호받지 않는 포인터raw pointer 타입

C의 포인터

```
fn foo(q: *mut i32) {
    *q += 1;

}

let p: *mut i32 = malloc(4);
*p = 0;
foo(p);
*p += 1;
free(p);
```

```
fn bar(q: *mut i32) {
    *q += 1;
    free(q);
}

let p: *mut i32 = malloc(4);
*p = 0;
bar(p);
```

소유권ownership에 상관없이 같은 타입

C의 포인터

```
fn foo(q: *mut i32) {
    *q += 1;

}

let p: *mut i32 = malloc(4);
*p = 0;
foo(p);
*p += 1;
free(p);  // memory leak
```

```
fn bar(q: *mut i32) {
    *q += 1;
    free(q);
}

let p: *mut i32 = malloc(4);
*p = 0;
bar(p);
*p += 1;  // use-after-free
free(p);  // double-free
```

소유권ownership에 상관없이 같은 타입

러스트의 포인터

`Box<i32>` 메모리를 소유한 포인터owning pointer 타입

`&mut i32` 메모리를 잠시 빌린 포인터borrowing pointer 타입

소유권ownership에 따라 서로 다른 타입

러스트의 포인터

```rust
fn foo(q: &mut i32) {
    *q += 1;

}

let p: Box<i32> = Box::new(0);
foo(&mut p);
*p += 1;
drop(p);
```

```rust
fn foo(q: Box<i32>) {
    *q += 1;
    drop(q);
}

let p: Box<i32> = Box::new(0);
foo(p);
```

소유권ownership에 따라 서로 다른 타입

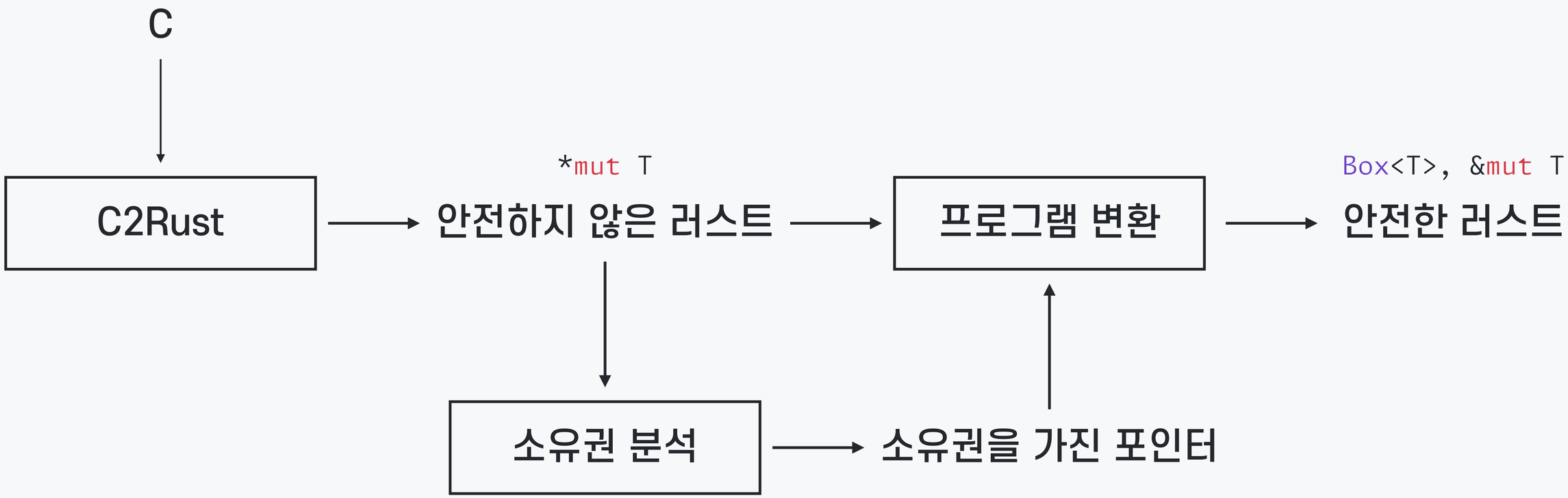
러스트의 포인터

```rust
fn foo(q: &mut i32) {
    *q += 1;

}

let p: Box<i32> = Box::new(0);
foo(&mut p);
*p += 1;
drop(p);  // drop inserted
```

```rust
fn foo(q: Box<i32>) {
    *q += 1;
    drop(q);
}

let p: Box<i32> = Box::new(0);
foo(p);
*p += 1;  // compile error
drop(p);  // compile error
```

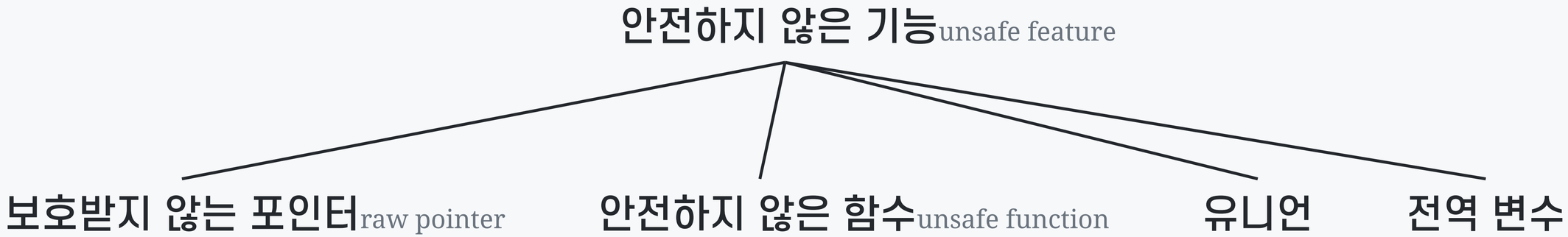
소유권ownership에 따라 서로 다른 타입

포인터 변환을 위한 소유권 분석

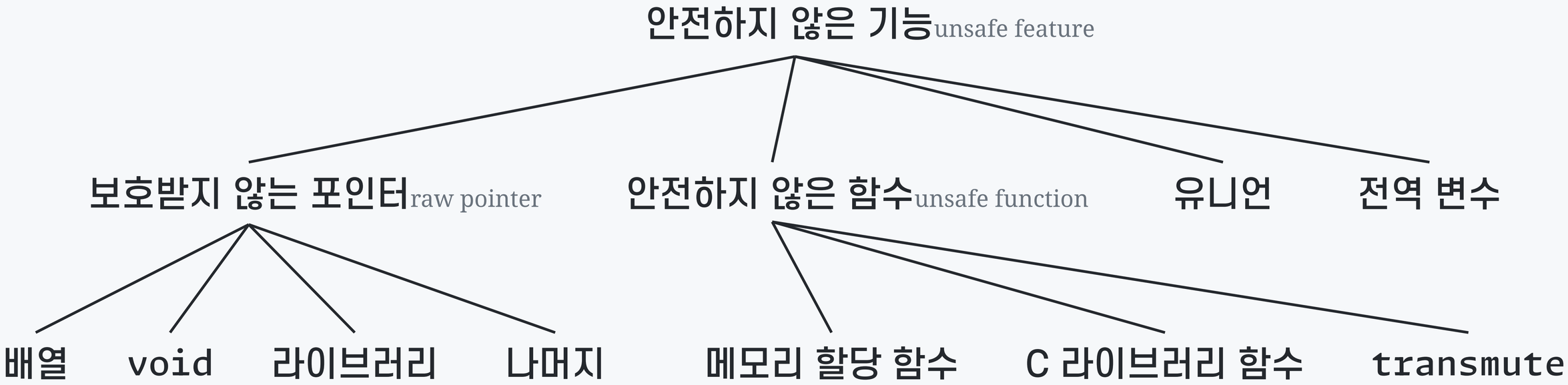


Zhang et al., Ownership Guided C to Rust Translation, CAV 2023.

안전하지 않은 기능들

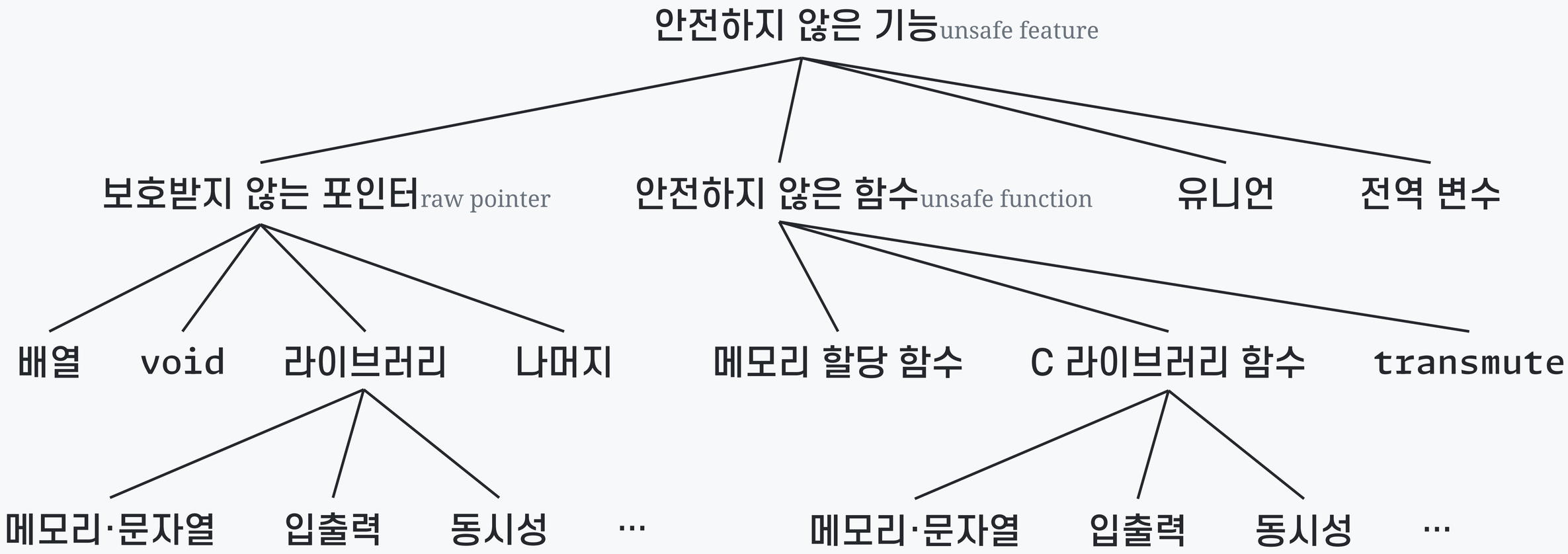

안전하지 않은 기능unsafe feature
보호받지 않는 포인터raw pointer
안전하지 않은 함수unsafe function
유니언
전역 변수

안전하지 않은 기능들



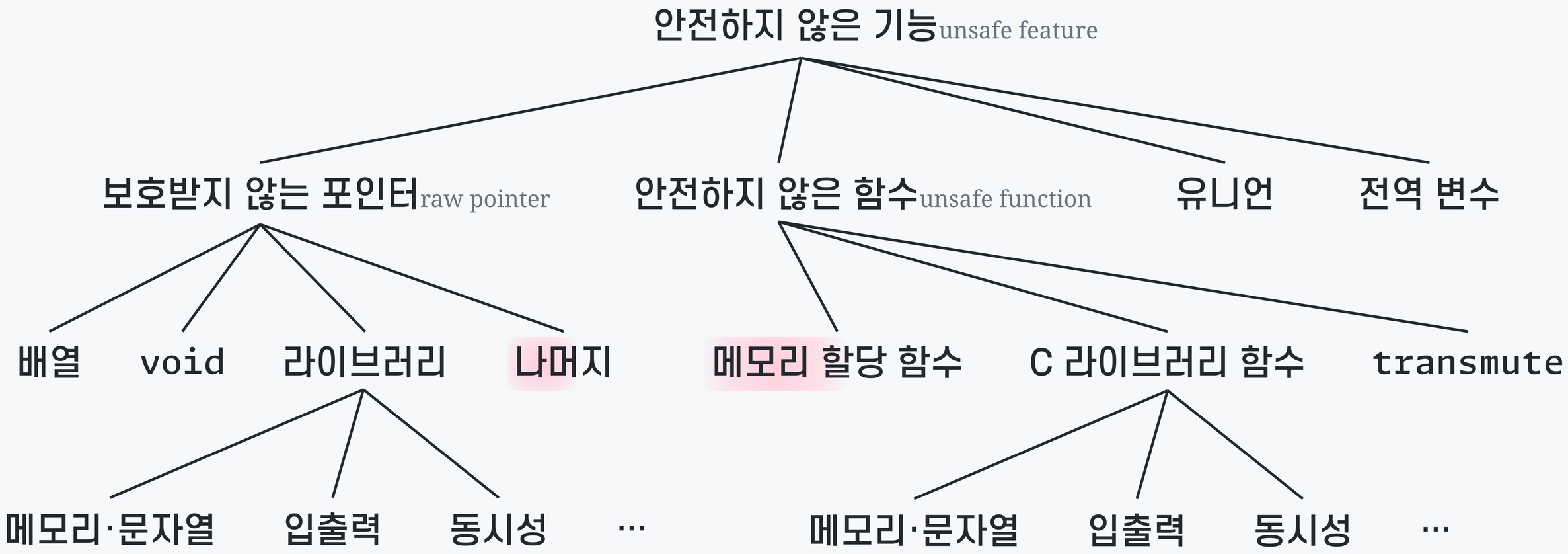
Emre et al., Translating C to Safer Rust, OOPSLA 2021.

안전하지 않은 기능들


안전하지 않은 기능unsafe feature
보호받지 않는 포인터raw pointer
안전하지 않은 함수unsafe function
유니언
전역 변수
배열
void
라이브러리
나머지
메모리 할당 함수
C 라이브러리 함수
transmute
메모리·문자열
입출력
동시성
…
메모리·문자열
입출력
동시성
…

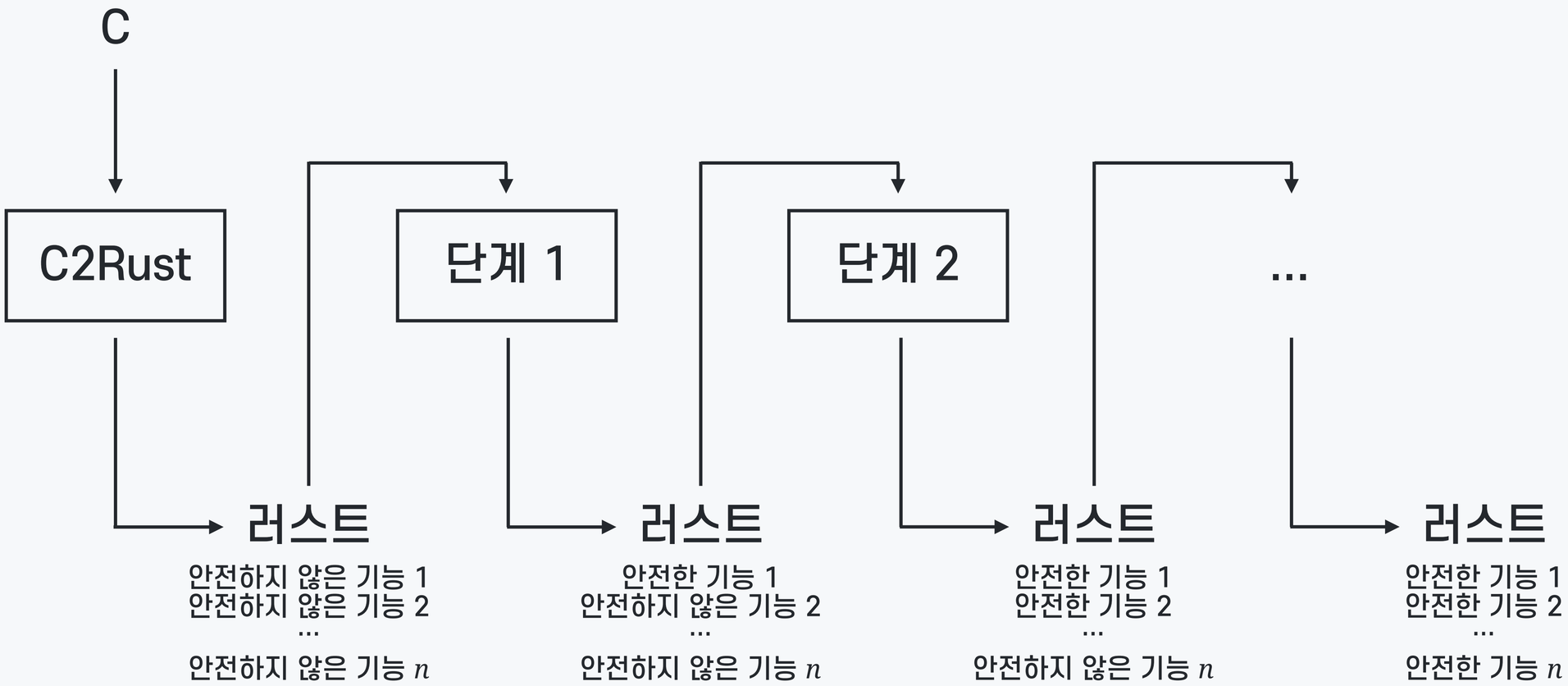
안전하지 않은 기능들

소유권 분석



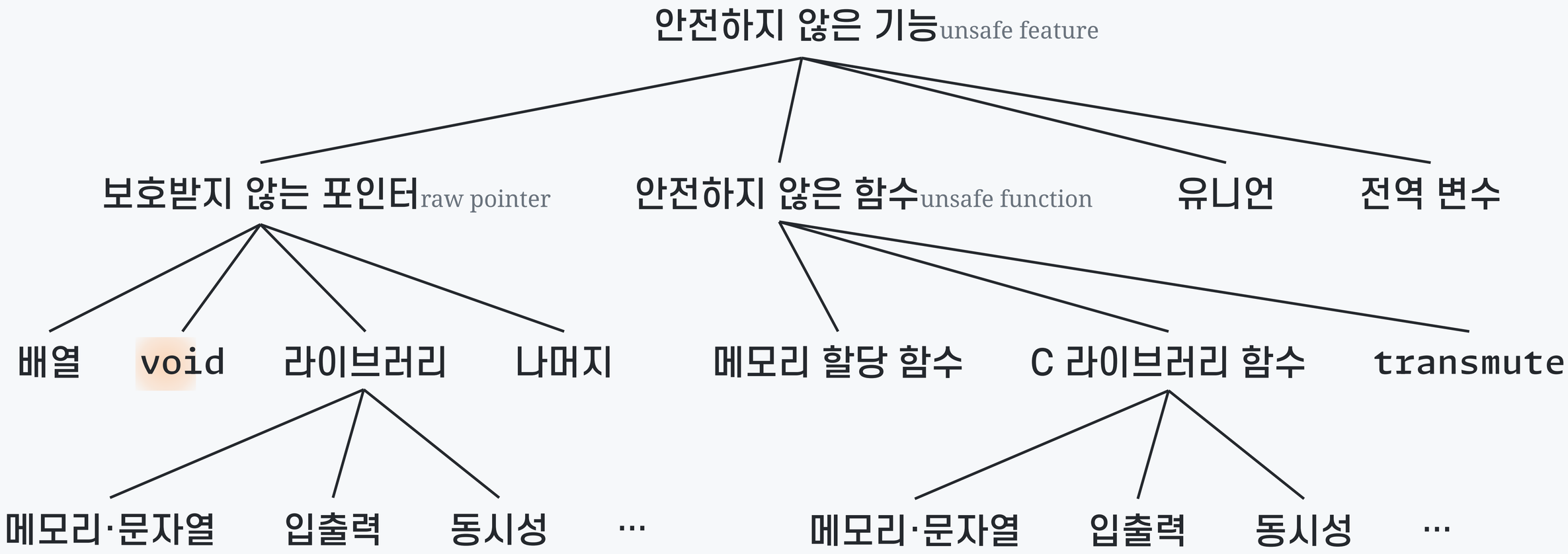
Zhang et al., Ownership Guided C to Rust Translation, CAV 2023.

여러 단계를 거치는 점진적 개선



안전하지 않은 기능들

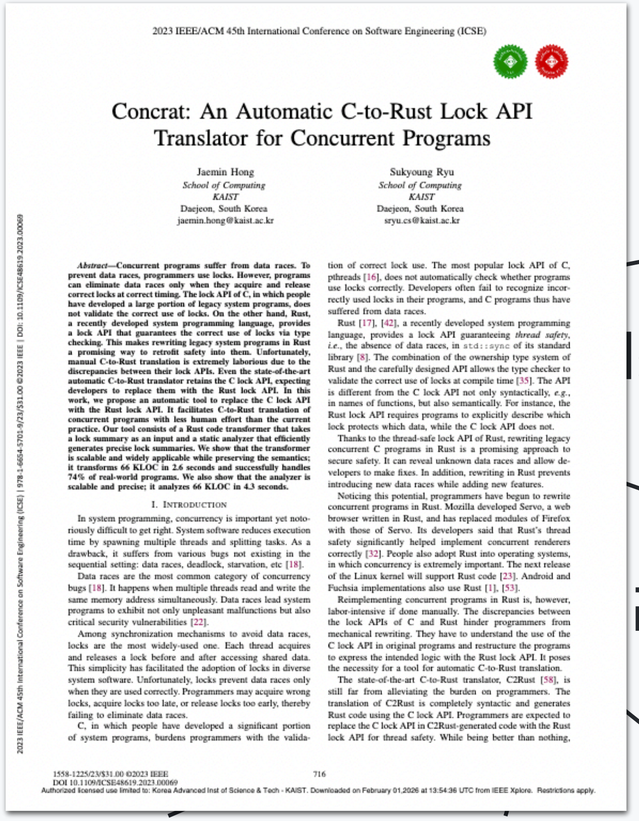
다형성 분석



Wu and Demsky, GenC2Rust: Towards Generating Generic Rust Code from C, ICSE 2025.

안전하지 않은 기능들

락 사용 분석



2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)

Concrat: An Automatic C-to-Rust Lock API Translator for Concurrent Programs

Jaemin Hong
School of Computing
KAIST
Daejeon, South Korea
jaemin.hong@kaist.ac.kr

Sukyoung Ryu
School of Computing
KAIST
Daejeon, South Korea
sryu.cs@kaist.ac.kr

***Abstract*—Concurrent programs suffer from data races. To prevent data races, programmers use locks. However, programs can eliminate data races only when they acquire and release correct locks at correct timing. The lock API of C, in which people have developed a large portion of legacy system programs, does not validate the correct use of locks. On the other hand, Rust, a recently developed system programming language, provides a lock API that guarantees the correct use of locks via type checking. This makes rewriting legacy system programs in Rust a promising way to retrofit safety into them. Unfortunately, manual C-to-Rust translation is extremely laborious due to the discrepancies between their lock APIs. Even the state-of-the-art automatic C-to-Rust translator retains the C lock API, expecting developers to replace them with the Rust lock API. In this work, we propose an automatic tool to replace the C lock API with the Rust lock API. It facilitates C-to-Rust translation of concurrent programs with less human effort than the current practice. Our tool consists of a Rust code transformer that takes a lock summary as an input and a static analyzer that efficiently generates precise lock summaries. We show that the transformer is scalable and widely applicable while preserving the semantics; it transforms 66 KLOC in 2.6 seconds and successfully handles 74% of real-world programs. We also show that the analyzer is scalable and precise; it analyzes 66 KLOC in 4.3 seconds.**

I. Introduction

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 [18].

Data races are the most common category of concurrency bugs [18]. 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 [22].

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, in which people have developed a significant portion of system programs, burdens programmers with the validation of correct lock use. The most popular lock API of C, pthreads [16], 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.

Rust [17], [42], a recently developed system programming language, provides a lock API guaranteeing *thread safety*, *i.e.*, the absence of data races, in `std::sync` of its standard library [8]. 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 [35]. 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.

Thanks to the thread-safe lock API of Rust, rewriting legacy concurrent C programs in Rust is a promising approach to secure safety. It can reveal unknown data races and allow developers to make fixes. In addition, rewriting in Rust prevents introducing new data races while adding new features.

Noticing this potential, programmers have begun to rewrite concurrent programs in Rust. Mozilla developed Servo, a web browser written in Rust, and has replaced modules of Firefox with those of Servo. Its developers said that Rust's thread safety significantly helped implement concurrent renderers correctly [32]. People also adopt Rust into operating systems, in which concurrency is extremely important. The next release of the Linux kernel will support Rust code [23]. Android and Fuchsia implementations also use Rust [1], [53].

Reimplementing concurrent programs in Rust is, however, labor-intensive if done manually. The discrepancies between the lock APIs of C and Rust hinder programmers from mechanical rewriting. They have to understand the use of the C lock API in original programs and restructure the programs to express the intended logic with the Rust lock API. It poses the necessity for a tool for automatic C-to-Rust translation.

The state-of-the-art C-to-Rust translator, C2Rust [58], is still far from alleviating the burden on programmers. The translation of C2Rust is completely syntactic and generates Rust code using the C lock API. Programmers are expected to replace the C lock API in C2Rust-generated code with the Rust lock API for thread safety. While being better than nothing,

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00069

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) | 978-1-6654-5701-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE48619.2023.00069

안전하지 않은 기능unsafe feature

raw pointer

안전하지 않은 함수unsafe function

유니언

전역 변수

리

나머지

메모리 할당 함수

C 라이브러리 함수

transmute

메모리·문자열

입출력

동시성

…

메모리·문자열

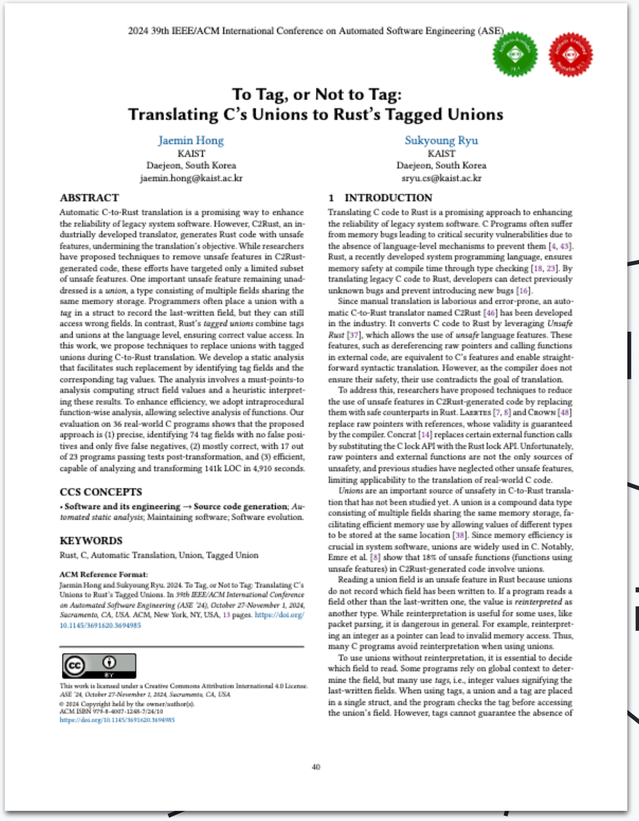
입출력

동시성

…

안전하지 않은 기능들

태그 분석



2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)

To Tag, or Not to Tag:
Translating C's Unions to Rust's Tagged Unions

Jaemin Hong
KAIST
Daejeon, South Korea
jaemin.hong@kaist.ac.kr

Sukyoung Ryu
KAIST
Daejeon, South Korea
sryu.cs@kaist.ac.kr

ABSTRACT

Automatic C-to-Rust translation is a promising way to enhance the reliability of legacy system software. However, C2Rust, an industrially developed translator, generates Rust code with unsafe features, undermining the translation's objective. While researchers have proposed techniques to remove unsafe features in C2Rust-generated code, these efforts have targeted only a limited subset of unsafe features. One important unsafe feature remaining unaddressed is a *union*, a type consisting of multiple fields sharing the same memory storage. Programmers often place a union with a *tag* in a struct to record the last-written field, but they can still access wrong fields. In contrast, Rust's *tagged unions* combine tags and unions at the language level, ensuring correct value access. In this work, we propose techniques to replace unions with tagged unions during C-to-Rust translation. We develop a static analysis that facilitates such replacement by identifying tag fields and the corresponding tag values. The analysis involves a must-points-to analysis computing struct field values and a heuristic interpreting these results. To enhance efficiency, we adopt intraprocedural function-wise analysis, allowing selective analysis of functions. Our evaluation on 36 real-world C programs shows that the proposed 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 post-transformation, and (3) efficient, capable of analyzing and transforming 141k LOC in 4,910 seconds.

CCS CONCEPTS

• **Software and its engineering** → **Source code generation**; *Automated static analysis*; Maintaining software; Software evolution.

KEYWORDS

Rust, C, Automatic Translation, Union, Tagged Union

ACM Reference Format:

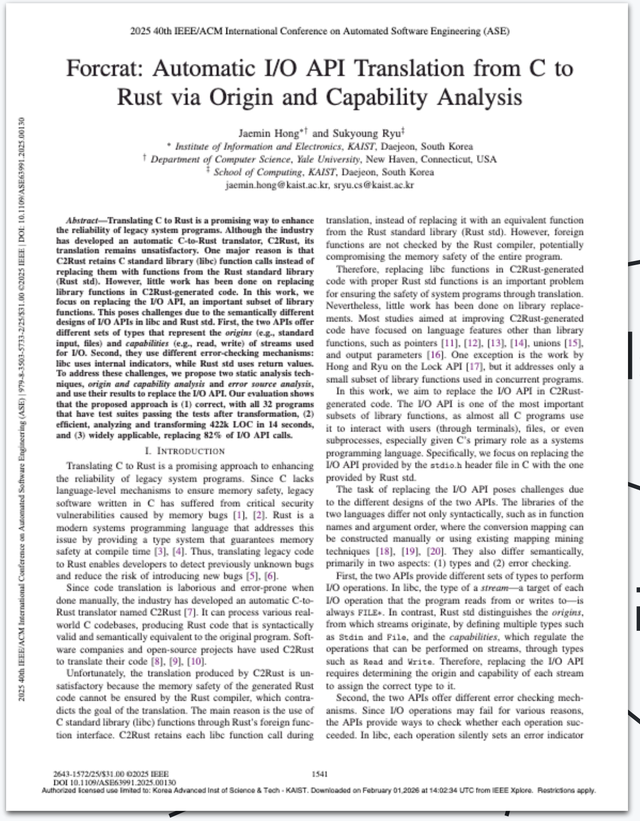
Jaemin Hong and Sukyoung Ryu. 2024. To Tag, or Not to Tag: Translating C's Unions to Rust's Tagged Unions. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3694985

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3694985

Hong and Ryu, To Tag, or Not to Tag: Translating C's Unions to Rust's Tagged Unions, ASE 2024.

안전하지 않은 기능들

스트림 분석



2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)

Forcrat: Automatic I/O API Translation from C to Rust via Origin and Capability Analysis

Jaemin Hong*† and Sukyoung Ryu†
* *Institute of Information and Electronics, KAIST*, Daejeon, South Korea
† *Department of Computer Science, Yale University*, New Haven, Connecticut, USA
‡ *School of Computing, KAIST*, Daejeon, South Korea
jaemin.hong@kaist.ac.kr, sryu.cs@kaist.ac.kr

***Abstract*—Translating C to Rust is a promising way to enhance the reliability of legacy system programs. Although the industry has developed an automatic C-to-Rust translator, C2Rust, its translation remains unsatisfactory. One major reason is that C2Rust retains C standard library (libc) function calls instead of replacing them with functions from the Rust standard library (Rust std). However, little work has been done on replacing library functions in C2Rust-generated code. In this work, we focus on replacing the I/O API, an important subset of library functions. This poses challenges due to the semantically different designs of I/O APIs in libc and Rust std. First, the two APIs offer different sets of types that represent the *origins* (e.g., standard input, files) and *capabilities* (e.g., read, write) of streams used for I/O. Second, they use different error-checking mechanisms: libc uses internal indicators, while Rust std uses return values. To address these challenges, we propose two static analysis techniques, *origin and capability analysis* and *error source analysis*, and use their results to replace the I/O API. Our evaluation shows that the proposed approach is (1) correct, with all 32 programs that have test suites passing the tests after transformation, (2) efficient, analyzing and transforming 422k LOC in 14 seconds, and (3) widely applicable, replacing 82% of I/O API calls.**

I. Introduction

Translating C to Rust is a promising approach to enhancing the reliability of legacy system programs. Since C lacks language-level mechanisms to ensure memory safety, legacy software written in C has suffered from critical security vulnerabilities caused by memory bugs [1], [2]. Rust is a modern systems programming language that addresses this issue by providing a type system that guarantees memory safety at compile time [3], [4]. Thus, translating legacy code to Rust enables developers to detect previously unknown bugs and reduce the risk of introducing new bugs [5], [6].

Since code translation is laborious and error-prone when done manually, the industry has developed an automatic C-to-Rust translator named C2Rust [7]. It can process various real-world C codebases, producing Rust code that is syntactically valid and semantically equivalent to the original program. Software companies and open-source projects have used C2Rust to translate their code [8], [9], [10].

Unfortunately, the translation produced by C2Rust is unsatisfactory because the memory safety of the generated Rust code cannot be ensured by the Rust compiler, which contradicts the goal of the translation. The main reason is the use of C standard library (libc) functions through Rust's foreign function interface. C2Rust retains each libc function call during translation, instead of replacing it with an equivalent function from the Rust standard library (Rust std). However, foreign functions are not checked by the Rust compiler, potentially compromising the memory safety of the entire program.

Therefore, replacing libc functions in C2Rust-generated code with proper Rust std functions is an important problem for ensuring the safety of system programs through translation. Nevertheless, little work has been done on library replacements. Most studies aimed at improving C2Rust-generated code have focused on language features other than library functions, such as pointers [11], [12], [13], [14], unions [15], and output parameters [16]. One exception is the work by Hong and Ryu on the Lock API [17], but it addresses only a small subset of library functions used in concurrent programs.

In this work, we aim to replace the I/O API in C2Rust-generated code. The I/O API is one of the most important subsets of library functions, as almost all C programs use it to interact with users (through terminals), files, or even subprocesses, especially given C's primary role as a systems programming language. Specifically, we focus on replacing the I/O API provided by the `stdio.h` header file in C with the one provided by Rust std.

The task of replacing the I/O API poses challenges due to the different designs of the two APIs. The libraries of the two languages differ not only syntactically, such as in function names and argument order, where the conversion mapping can be constructed manually or using existing mapping mining techniques [18], [19], [20]. They also differ semantically, primarily in two aspects: (1) types and (2) error checking.

First, the two APIs provide different sets of types to perform I/O operations. In libc, the type of a *stream*—a target of each I/O operation that the program reads from or writes to—is always `FILE*`. In contrast, Rust std distinguishes the *origins*, from which streams originate, by defining multiple types such as `Stdin` and `File`, and the *capabilities*, which regulate the operations that can be performed on streams, through types such as `Read` and `Write`. Therefore, replacing the I/O API requires determining the origin and capability of each stream to assign the correct type to it.

Second, the two APIs offer different error checking mechanisms. Since I/O operations may fail for various reasons, the APIs provide ways to check whether each operation succeeded. In libc, each operation silently sets an error indicator

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00130

2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-5733-2/25/$31.00 ©2025 IEEE | DOI: 10.1109/ASE63991.2025.00130

안전하지 않은 기능unsafe feature

raw pointer

안전하지 않은 함수unsafe function

유니언

전역 변수

리

나머지

메모리 할당 함수

C 라이브러리 함수

`transmute`

메모리·문자열

입출력

동시성

…

메모리·문자열

입출력

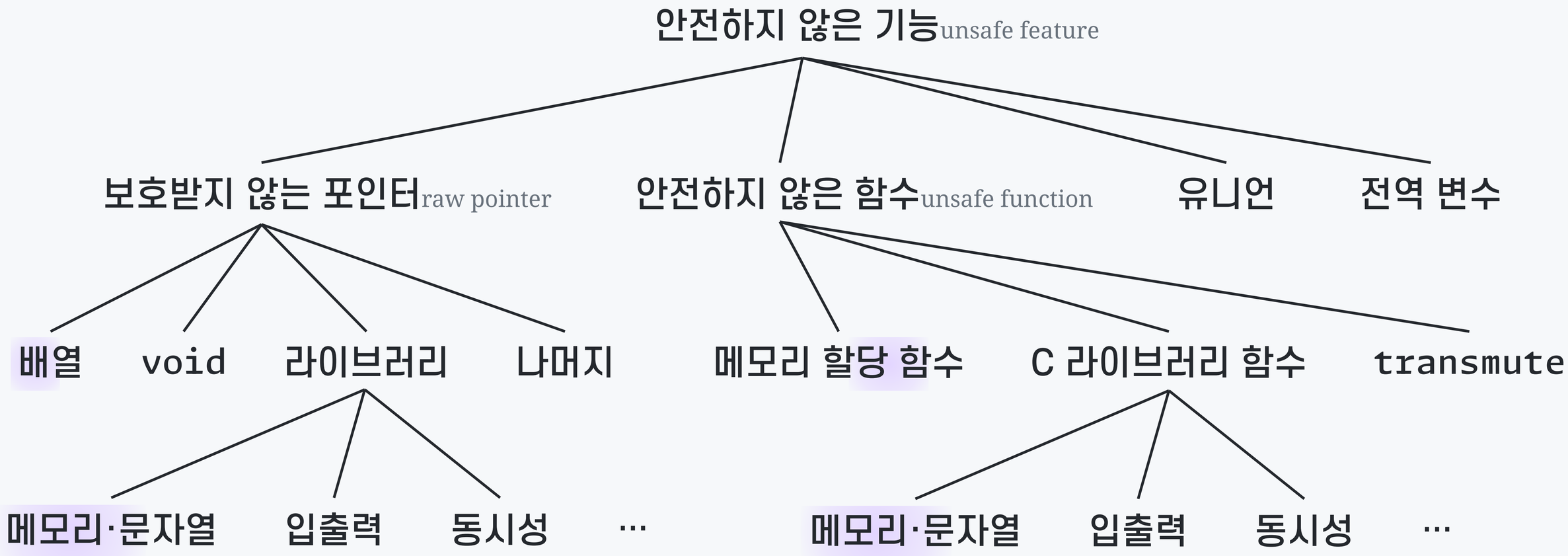
동시성

…

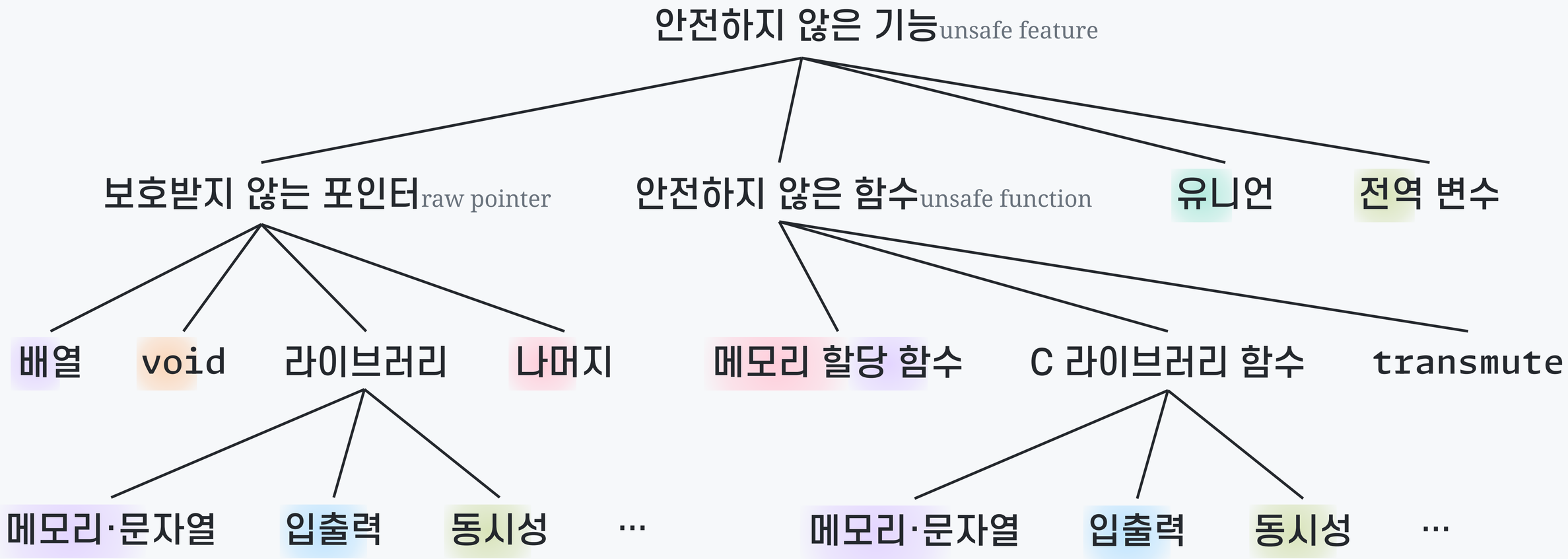
안전하지 않은 기능들

배열 분석

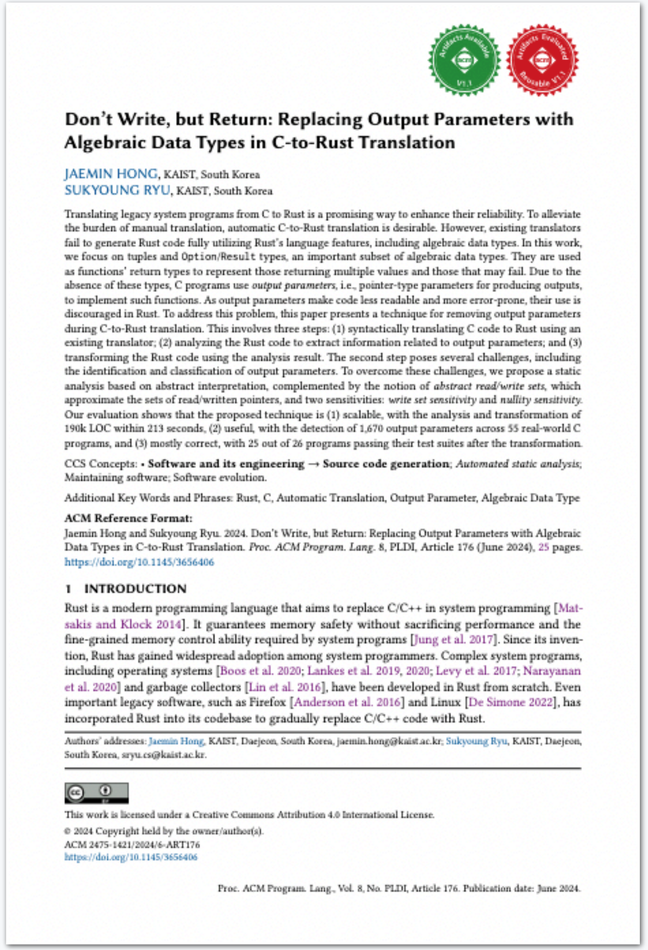
(연구 진행 중)



안전하지 않은 기능들


안전하지 않은 기능unsafe feature
보호받지 않는 포인터raw pointer
안전하지 않은 함수unsafe function
유니언
전역 변수
배열
void
라이브러리
나머지
메모리 할당 함수
C 라이브러리 함수
transmute
메모리·문자열
입출력
동시성
…
메모리·문자열
입출력
동시성
…

자연스럽지 않은 패턴들



Don't Write, but Return: Replacing Output Parameters with Algebraic Data Types in C-to-Rust Translation

JAEMIN HONG, KAIST, South Korea
SUKYOUNG RYU, KAIST, South Korea

Translating legacy system programs from C to Rust is a promising way to enhance their reliability. To alleviate the burden of manual translation, automatic C-to-Rust translation is desirable. However, existing translators fail to generate Rust code fully utilizing Rust's language features, including algebraic data types. In this work, we focus on tuples and `Option`/`Result` types, an important subset of algebraic data types. They are used as functions' return types to represent those returning multiple values and those that may fail. Due to the absence of these types, C programs use *output parameters*, i.e., pointer-type parameters for producing outputs, to implement such functions. As output parameters make code less readable and more error-prone, their use is discouraged in Rust. To address this problem, this paper presents a technique for removing output parameters during C-to-Rust translation. This involves three steps: (1) syntactically translating C code to Rust using an existing translator; (2) analyzing the Rust code to extract information related to output parameters; and (3) transforming the Rust code using the analysis result. The second step poses several challenges, including the identification and classification of output parameters. To overcome these challenges, we propose a static analysis based on abstract interpretation, complemented by the notion of *abstract read/write sets*, which approximate the sets of read/written pointers, and two sensitivities: *write set sensitivity* and *nullity sensitivity*. Our evaluation shows that the proposed technique is (1) scalable, with the analysis and transformation of 190k LOC within 213 seconds, (2) useful, with the detection of 1,670 output parameters across 55 real-world C programs, and (3) mostly correct, with 25 out of 26 programs passing their test suites after the transformation.

CCS Concepts: • **Software and its engineering** → **Source code generation**; *Automated static analysis*; Maintaining software; Software evolution.

Additional Key Words and Phrases: Rust, C, Automatic Translation, Output Parameter, Algebraic Data Type

ACM Reference Format:
Jaemin Hong and Sukyoung Ryu. 2024. Don't Write, but Return: Replacing Output Parameters with Algebraic Data Types in C-to-Rust Translation. *Proc. ACM Program. Lang.* 8, PLDI, Article 176 (June 2024), 25 pages. https://doi.org/10.1145/3656406

1 INTRODUCTION

Rust is a modern programming language that aims to replace C/C++ in system programming [Matsakis and Klock 2014]. It guarantees memory safety without sacrificing performance and the fine-grained memory control ability required by system programs [Jung et al. 2017]. Since its invention, Rust has gained widespread adoption among system programmers. Complex system programs, including operating systems [Boos et al. 2020; Lankes et al. 2019, 2020; Levy et al. 2017; Narayanan et al. 2020] and garbage collectors [Lin et al. 2016], have been developed in Rust from scratch. Even important legacy software, such as Firefox [Anderson et al. 2016] and Linux [De Simone 2022], has incorporated Rust into its codebase to gradually replace C/C++ code with Rust.

Authors' addresses: Jaemin Hong, KAIST, Daejeon, South Korea, jaemin.hong@kaist.ac.kr; Sukyoung Ryu, KAIST, Daejeon, South Korea, sryu.cs@kaist.ac.kr.

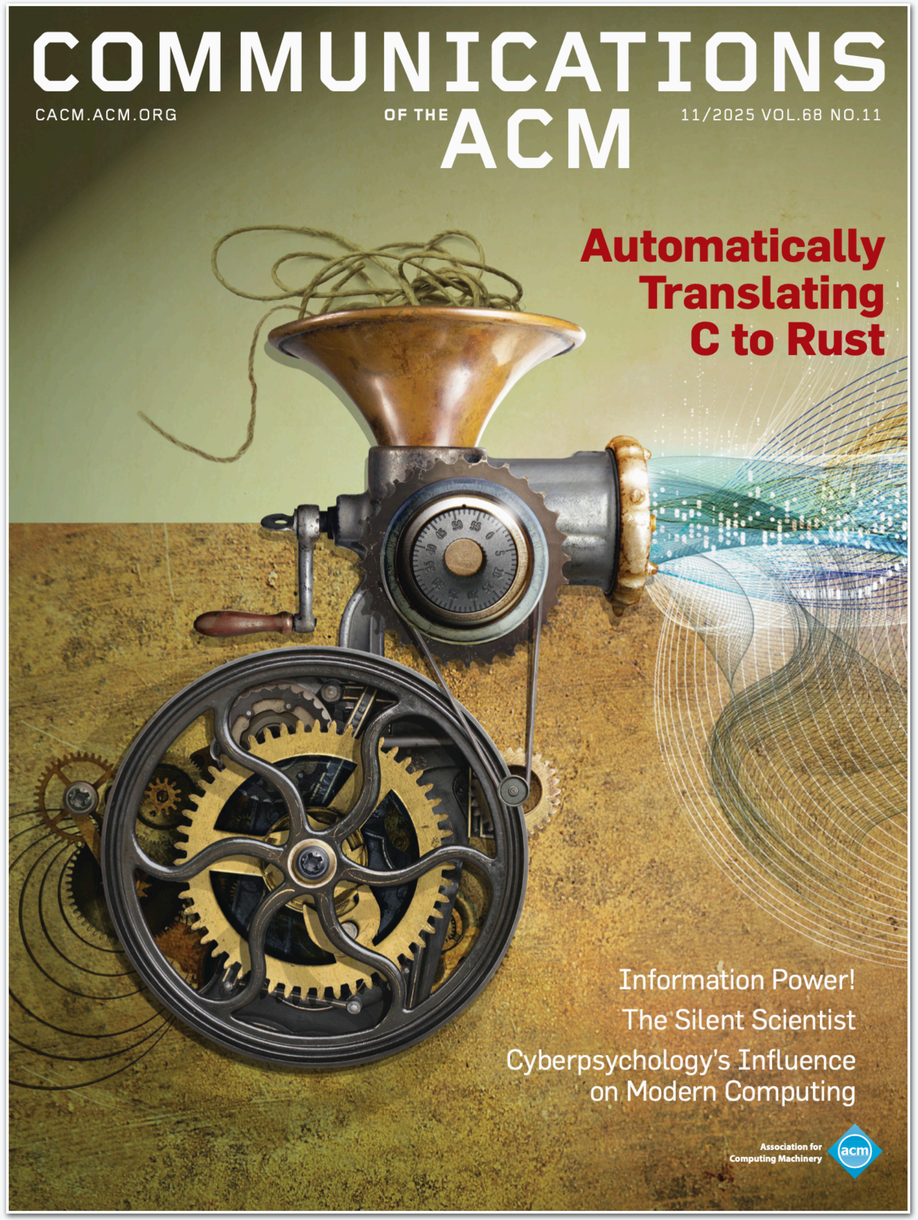
ACM 2475-1421/2024/6-ART176
https://doi.org/10.1145/3656406

Proc. ACM Program. Lang., Vol. 8, No. PLDI, Article 176. Publication date: June 2024.

```rust
fn div(n: i32, d: i32, r: *mut i32) -> i32 {
    *r = n % d; return n / d;
}
```

```rust
fn div(n: i32, d: i32) -> (i32, i32) {
    return (n / d, n % d);
}
```

Hong and Ryu, Don't Write, but Return: Replacing Output Parameters with Algebraic Data Types in C-to-Rust Translation, PLDI 2024.



DOI:10.1145/3737696

Automatic C-to-Rust translation tools are helpful, but they produce unsafe and unidiomatic code. What can be done to address these issues?

BY JAEMIN HONG AND SUKYOUNG RYU

Automatically Translating C to Rust

IN THE SOFTWARE industry, legacy systems developed in older languages often evolve by being reimplemented in newer languages that offer modern language features. For example, Twitter migrated from Ruby to Scala to enhance performance and reliability;[29] Dropbox rewrote its Python backends in Go to leverage better concurrency support and faster execution;[18] and banking systems originally written in Cobol have evolved to Java or C# for easier maintenance and integration with modern infrastructures.[3]

One of the most critical language migrations needed today is the shift from C to newer languages, which can improve the reliability of important system programs. Despite its popularity in systems programming, C is infamous for its limited language-level safety mechanisms. C programs are prone to memory bugs, such as buffer overflow and use-after-free, which can lead to severe security vulnerabilities—exemplified by the Heartbleed bug[a] in OpenSSL. A large portion of vulnerabilities in legacy systems arise from memory bugs; for example, approximately 70% of the vulnerabilities in Microsoft's codebase are due to memory bugs.[28] Acknowledging these risks, even the White House recently recommended discouraging the use of C.[24]

The most promising migration target for C is Rust,[21] which provides strong safety guarantees while still allowing low-level memory control and high performance. Its safety guarantees are enabled by its ownership type system, which ensures the absence of memory bugs in programs that pass type checking.[15] Due to this advantage, Rust has been widely adopted in systems programming, as demonstrated by the development of clean-slate system programs such as garbage collectors,[20] Web browsers,[1] and operating systems.[2,16,19,22]

Recognizing Rust's potential, the industry has shown significant interest in migrating legacy systems from C to Rust. Such migration allows developers to detect previously unknown

a https://heartbleed.com/

key insights

- **Migrating legacy systems developed in C to Rust is a promising way to enhance reliability, thanks to Rust's strong safety guarantees.**
- **Automatic translators can facilitate migration from C to Rust, but existing translators generate unsatisfactory code by relying on language features whose safety is not validated by the compiler and code patterns considered unidiomatic by Rust developers.**
- **Carefully designed static analyses and code transformations can improve automatic translation by replacing unsafe features and unidiomatic patterns with safe and idiomatic alternatives.**
- **Although LLMs are another promising approach to C-to-Rust translation, they often produce code with type errors or behavior different from the original code. Combining LLMs with static analyses is a potential future research direction.**

ILLUSTRATION BY VICTOR KOEN

Hong and Ryu, Automatically Translating C to Rust, CACM 2025.

분석 정확도

분석이 아주 정확하지 않아도 괜찮다

1. 100개의 경보alarm 중 10개가 진짜 경보true alarm vs 100개의 포인터 중 10개를 변환

2. 타입 검사기만큼만 정확해도 충분하다

C와 러스트의 입출력 스트림

`*mut FILE` 능력capability에 상관없이 같은 타입

C, 안전하지 않은 러스트Unsafe Rust

안전한 러스트Safe Rust

`Read`
`Write` 능력capability에 따라 서로 다른 타입
`Seek`

스트림 능력 분석

안전하지 않은 러스트 Unsafe Rust

```
fn foo(fp: *mut FILE) {
    fgetc(fp);
    fseek(fp, …, …);
}
```

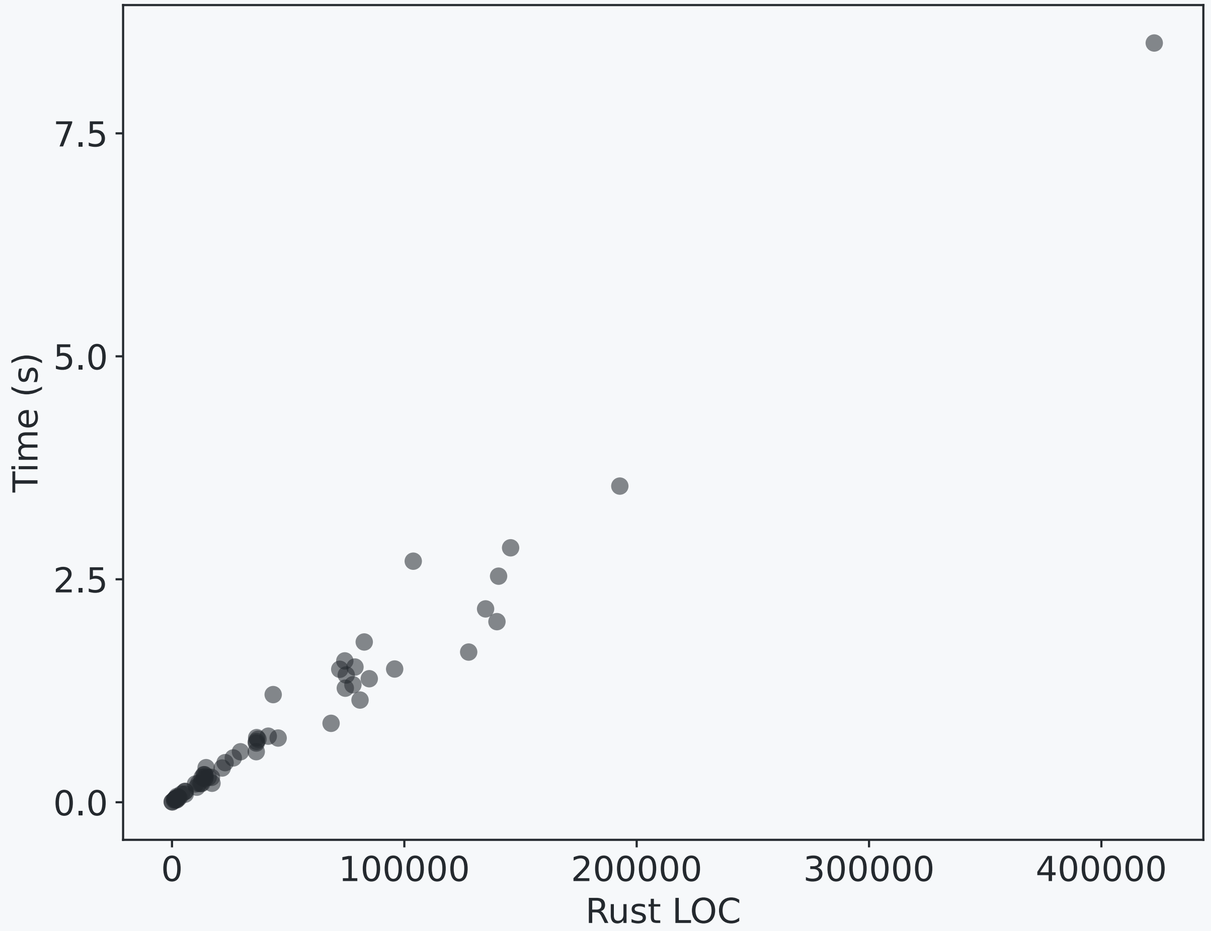
안전한 러스트 Safe Rust

```
fn foo<T: Read + Seek>(fp: T) {
    fp.read(…);
    fp.seek(…);
}
```

실행 흐름을 고려하지 않는 분석flow-insensitive analysis**으로도 충분하다**

Hong and Ryu, Forcrat: Automatic I/O API Translation from C to Rust via Origin and Capability Analysis, ASE 2025.

스트림 능력 분석



이론상 $O(n^3)$

82%의 입출력 API 호출 변환

32/32 테스트 통과

(62개의 실제 C 프로그램 대상 실험)

Hong and Ryu, Forcrat: Automatic I/O API Translation from C to Rust via Origin and Capability Analysis, ASE 2025.

C와 안전하지 않은 러스트의 락

```
static n: i32 = 0;
static m: pthread_mutex_t = …;

pthread_mutex_lock(&mut m);
n += 1;
pthread_mutex_unlock(&mut m);
```

1. 락-데이터 관계가 명시적이지 않음

2. 락-흐름 관계가 명시적이지 않음

안전한 러스트의 락

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

let guard;
guard = m.lock();
*guard += 1;
drop(guard);
```

1. 락-데이터 관계가 명시적

2. 락-흐름 관계가 명시적

락 사용 분석

```
static n: i32 = 0;
static m: pthread_mutex_t = …;

pthread_mutex_lock(&mut m);
n += 1;
pthread_mutex_unlock(&mut m);
```

{}
{m}
{m}
{}

실행 흐름을 고려하는 분석flow-sensitive analysis**이 필요하다**

Hong and Ryu, Concrat: An Automatic C-to-Rust Lock API Translator for Concurrent Programs, ICSE 2023.

락 사용 분석

```
static n: i32 = 0;
static m: pthread_mutex_t = …;

if c { pthread_mutex_lock(&mut m); }
if c { n += 1; }
if c { pthread_mutex_unlock(&mut m); }
```

```
{}
{}
{}
{}
```

실행 경로를 고려하지 않는 분석path-insensitive analysis**은 부정확하다**

Hong and Ryu, Concrat: An Automatic C-to-Rust Lock API Translator for Concurrent Programs, ICSE 2023.

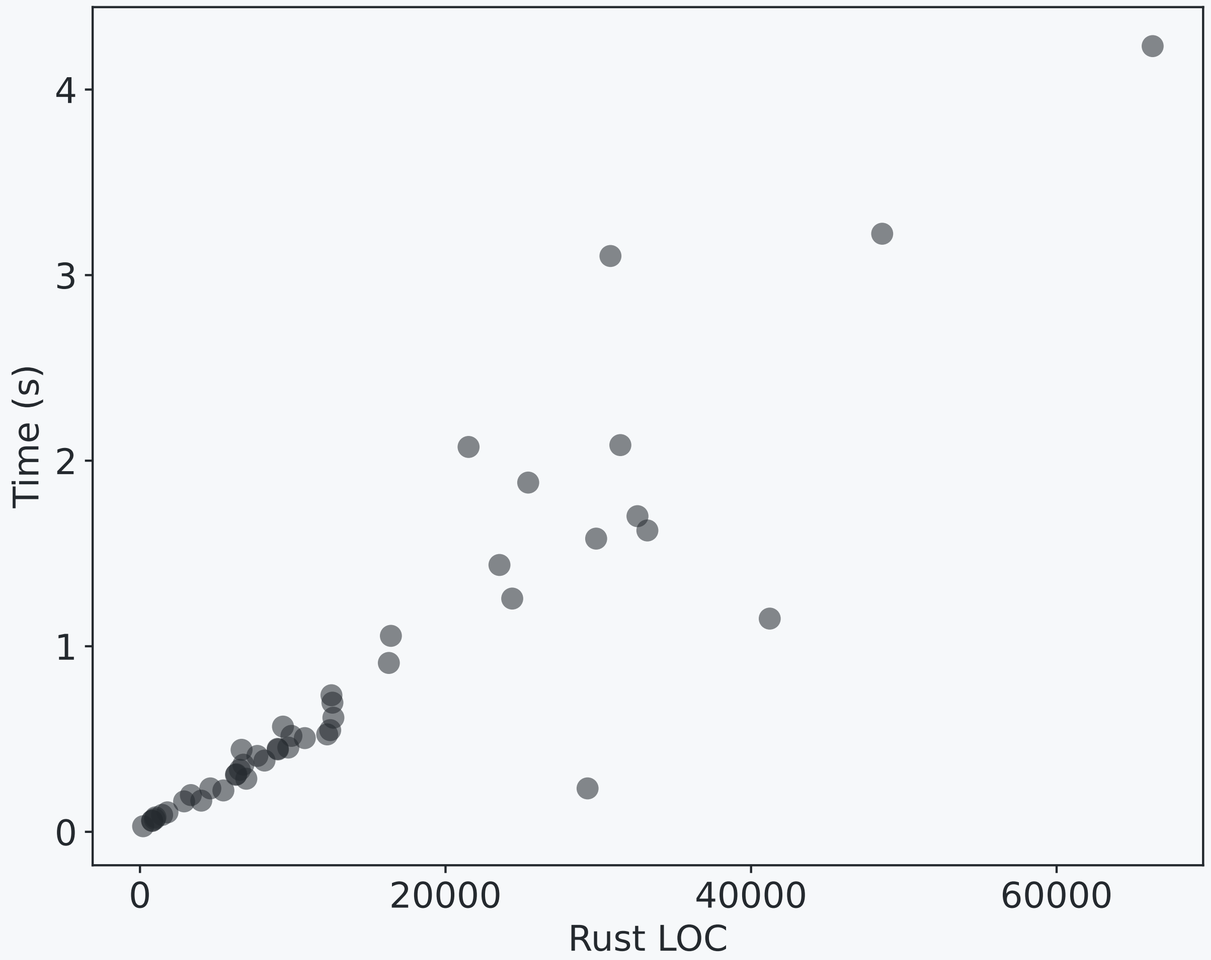
락 사용 분석

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

let guard;
if c { guard = m.lock(); }
if c { *guard += 1; }  // compile error
if c { drop(guard); }  // compile error
```

실행 경로를 고려하지 않는 분석path-insensitive analysis**으로도 충분하다**

Hong and Ryu, Concrat: An Automatic C-to-Rust Lock API Translator for Concurrent Programs, ICSE 2023.

락 사용 분석



이론상 $O(n^2 \cdot l^2)$

34/46 컴파일 가능

17/18 테스트 통과

(46개의 실제 C 프로그램 대상 실험)

Hong and Ryu, Concrat: An Automatic C-to-Rust Lock API Translator for Concurrent Programs, ICSE 2023.

프로그램 변환의 어려움

```
x = y;
```

	y: Box<i32>	y: &mut i32
x: Box<i32>	x = y;	불가능
x: &mut i32	x = &mut y;	x = y;

프로그램 변환의 어려움

```
x = y;
```

```
*mut i32    &mut i32    Option<&mut i32>    &mut [i32]    Option<&mut [i32]>

Box<i32>    Option<Box<i32>>    Box<[i32]>    Option<Box<[i32]>>
```

$$9 \times 9 = 81$$

프로그램 변환의 어려움

fgetc

ftell

rewind

fputc

puts

fflush

vfprintf

fputs

perror

fprintf

fseek

fgets

fscanf

remove

fread

getline

fopen

fwrite

rename

getdelim

프로그램 변환의 어려움

[E2BIG]
Argument list too long.
[EACCES]
Permission denied.
[EADDRINUSE]
Address in use.
[EADDRNOTAVAIL]
Address not available.
[EAFNOSUPPORT]
Address family not supported.
[EAGAIN]
Resource unavailable, try again (may be the same
[EALREADY]
Connection already in progress.
[EBADF]
Bad file descriptor.
[EBADMSG]
Bad message.
[EBUSY]
Device or resource busy.
[ECANCELED]
Operation canceled.
[ECHILD]
No child processes.
[ECONNABORTED]
Connection aborted.
[ECONNREFUSED]
Connection refused.
[ECONNRESET]
Connection reset.
[EDEADLK]
Resource deadlock would occur.
[EDESTADDRREQ]
Destination address required.
[EDOM]
Mathematics argument out of domain of function.
[EDQUOT]
Reserved.
[EEXIST]
File exists.
[EFAULT]
Bad address.
[EFBIG]
File too large.
[EHOSTUNREACH]
Host is unreachable.

[EIDRM]
Identifier removed.
[EILSEQ]
Illegal byte sequence.
[EINPROGRESS]
Operation in progress.
[EINTR]
Interrupted function.
[EINVAL]
Invalid argument.
[EIO]
I/O error.
[EISCONN]
Socket is connected.
[EISDIR]
Is a directory.
[ELOOP]
Too many levels of symbolic links.
[EMFILE]
File descriptor value too large.
[EMLINK]
Too many hard links.
[EMSGSIZE]
Message too large.
[EMULTIHOP]
Reserved.
[ENAMETOOLONG]
Filename too long.
[ENETDOWN]
Network is down.
[ENETRESET]
Connection aborted by network.
[ENETUNREACH]
Network unreachable.
[ENFILE]
Too many files open in system.
[ENOBUFS]
No buffer space available.
[ENODEV]
No such device.
[ENOENT]
No such file or directory.
[ENOEXEC]
Executable file format error.
[ENOLCK]
No locks available.

[ENOLINK]
Reserved.
[ENOMEM]
Not enough space.
[ENOMSG]
No message of the desired type.
[ENOPROTOOPT]
Protocol not available.
[ENOSPC]
No space left on device.
[ENOSYS]
Functionality not supported.
[ENOTCONN]
The socket is not connected.
[ENOTDIR]
Not a directory or a symbolic link to a direc
[ENOTEMPTY]
Directory not empty.
[ENOTRECOVERABLE]

State not recoverable.
[ENOTSOCK]
Not a socket.
[ENOTSUP]
Not supported (may be the same value as
[ENOTTY]
Inappropriate I/O control operation.
[ENXIO]
No such device or address.
[EOPNOTSUPP]
Operation not supported on socket (may be
[EOVERFLOW]
Value too large to be stored in data type.
[EOWNERDEAD]
Previous owner died.
[EPERM]
Operation not permitted.
[EPIPE]
Broken pipe.
[EPROTO]
Protocol error.
[EPROTONOSUPPORT]

Protocol not supported.

[EPROTOTYPE]
Protocol wrong type for socket.
[ERANGE]
Result too large.
[EROFS]
Read-only file system.
[ESOCKTNOSUPPORT]

Socket type not supported.
[ESPIPE]
Invalid seek.
[ESRCH]
No such process.
[ESTALE]
Reserved.
[ETIMEDOUT]
Connection timed out.
[ETXTBSY]
Text file busy.
[EWOULDBLOCK]
Operation would block (may be th
[EXDEV]
Improper hard link.

올바른 번역이란?

A pointer to an object or incomplete type may be converted to a pointer to a different object or incomplete type. If the resulting pointer is not correctly aligned[57] for the pointed-to type, the behavior is undefined. Otherwise, when converted back again, the

(C99 § 6.3.2.3)

```
* BROTLI_BUILD_NO_UNALIGNED_READ_FAST forces off the fast-unaligned-read
  optimizations (mainly for testing purposes)
```

```
#if defined(BROTLI_BUILD_NO_UNALIGNED_READ_FAST)
#define BROTLI_UNALIGNED_READ_FAST (!!0)
#elif defined(BROTLI_TARGET_X86) || defined(BROTLI_TARGET_X64) ||       \
    defined(BROTLI_TARGET_ARMV7) || defined(BROTLI_TARGET_ARMV8_ANY) || \
    defined(BROTLI_TARGET_RISCV64) || defined(BROTLI_TARGET_LOONGARCH64)
/* These targets are known to generate efficient code for unaligned reads
 * (e.g. a single instruction, not multiple 1-byte loads, shifted and or'd
 * together). */
#define BROTLI_UNALIGNED_READ_FAST (!!1)
```

(구글 Brotli)

올바른 번역이란?

Translate the following C code into equivalent Safe Rust code:

```
int main() {
    int x;
    char c;
    scanf("%d %c", &x, &c);
    printf("%d %c\n", x, c);
}
```

```rust
fn main() {
    let mut input = String::new();
    io::stdin().read_to_string(&mut input).unwrap();
    let mut iter = input.split_whitespace();
    let x: i32 =
        iter.next().unwrap().parse().unwrap();
    let c: char =
        iter.next().unwrap().chars().next().unwrap();
    println!("{} {}", x, c);
}
```

올바른 번역이란?

Translate the following C code into equivalent Safe Rust code:

```
int main() {
    int x;
    char c;
    scanf("%d %c", &x, &c);
    printf("%d %c\n", x, c);
}
```

```
fn main() {
    let mut input = String::new();
    io::stdin().read_to_string(&mut input).unwrap();
    let mut iter = input.split_whitespace();
    let x: i32 =
        iter.next().unwrap().parse().unwrap();
    let c: char =
        iter.next().unwrap().chars().next().unwrap();
    println!("{} {}", x, c);
}
```

```
1 x
```

```
thread 'main' panicked at a.rs:12:47:
called `Result::unwrap()` on an `Err` value: ParseIntError { kind: InvalidDigit }
```

입력: 1x

LLM을 통한 번역

https://hjaem.info/c-to-rust-papers

Type-migrating C-to-Rust Translation Using a Large Language Model

Towards Translating Real-World Code with LLMs: A Study of Translating to Rust

Context-aware Code Segmentation for C-to-Rust Translation Using Large Language Models

Syzygy: Dual Code-Test C to (Safe) Rust Translation Using LLMs and Dynamic Analysis

Optimizing Type Migration for LLM-Based C-to-Rust Translation: A Data Flow Graph Approach

C2RustTV: An LLM-based Framework for C to Rust Translation and Validation

VERT: Polyglot Verified Equivalent Rust Transpilation with Large Language Models

RustAssure: Differential Symbolic Testing for LLM-Transpiled C-to-Rust Code

RustMap: Towards Project-Scale C-to-Rust Migration via Program Analysis and LLM

A Systematic Exploration of C-to-Rust Code Translation Based on Large Language Models: Prompt Strategies and Automated Repair

C2SaferRust: Transforming C Projects into Safer Rust with NeuroSymbolic Techniques

PR2: Peephole Raw Pointer Rewriting with LLMs for Translating C to Safer Rust

LLM-Driven Multi-Step Translation from C to Rust Using Static Analysis

Enhancing LLM-based Code Translation in Repository Context via Triple Knowledge-Augmented

LLMigrate: Transforming "Lazy" Large Language Models into Efficient Source Code Migrators

SafeTrans: LLM-assisted Transpilation from C to Rust

Large Language Model-Powered Agent for C to Rust Code Translation

EVOC2RUST: A Skeleton-Guided Framework for Project-Level C-to-Rust Translation

Integrating Rules and Semantics for LLM-Based C-to-Rust Translation

MatchFixAgent: Language-Agnostic Autonomous Repository-Level Code Translation Validation and Repair

Adversarial Agent Collaboration for C to Rust Translation

Project-Level C-to-Rust Translation via Synergistic Integration of Knowledge Graphs and Large Language Models

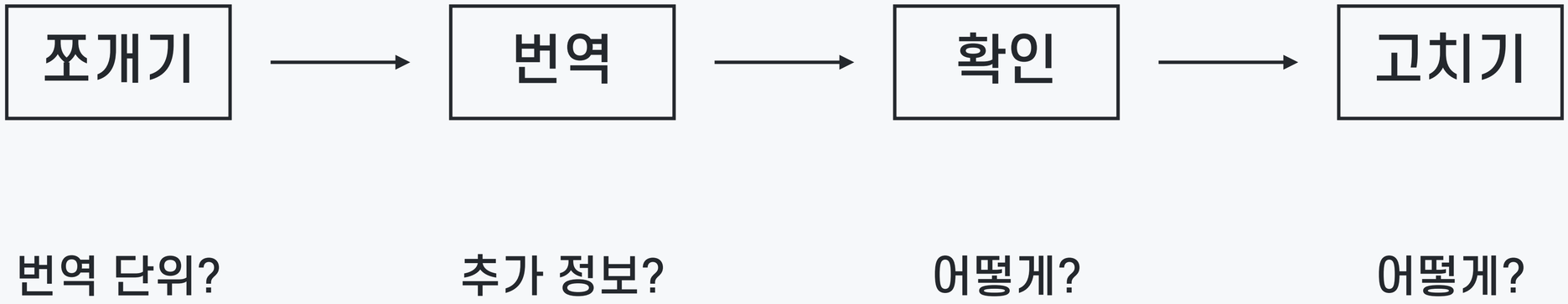
Rustify: Towards Repository-Level C to Safer Rust via Workflow-Guided Multi-Agent Transpiler

Translating Large-Scale C Repositories to Idiomatic Rust

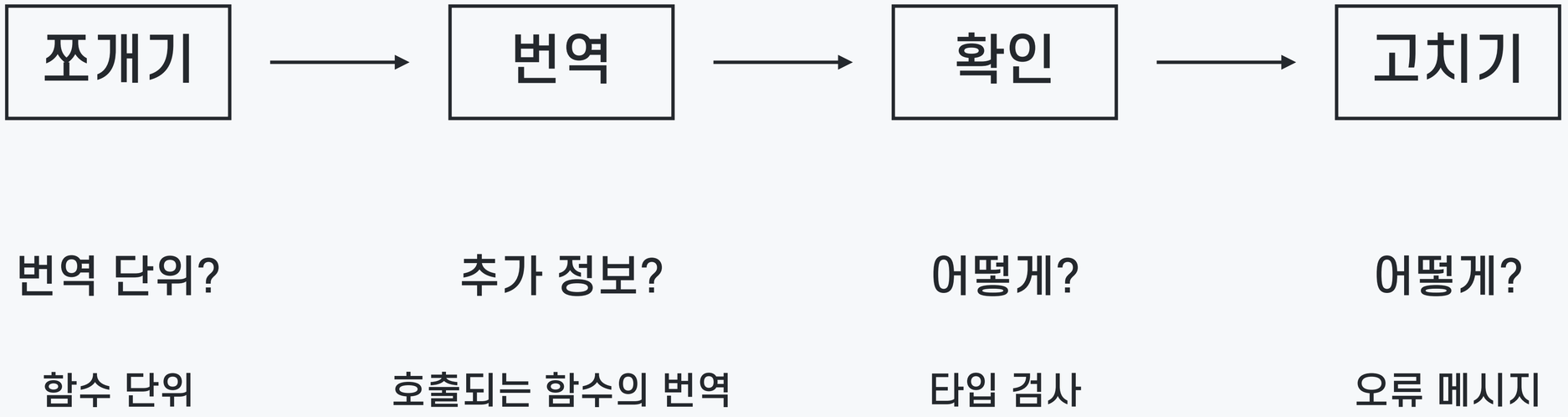
SmartC2Rust: Iterative, Feedback-Driven C-to-Rust Translation via Large Language Models for Safety and Equivalence

LLM을 통한 번역

LLM을 통한 번역

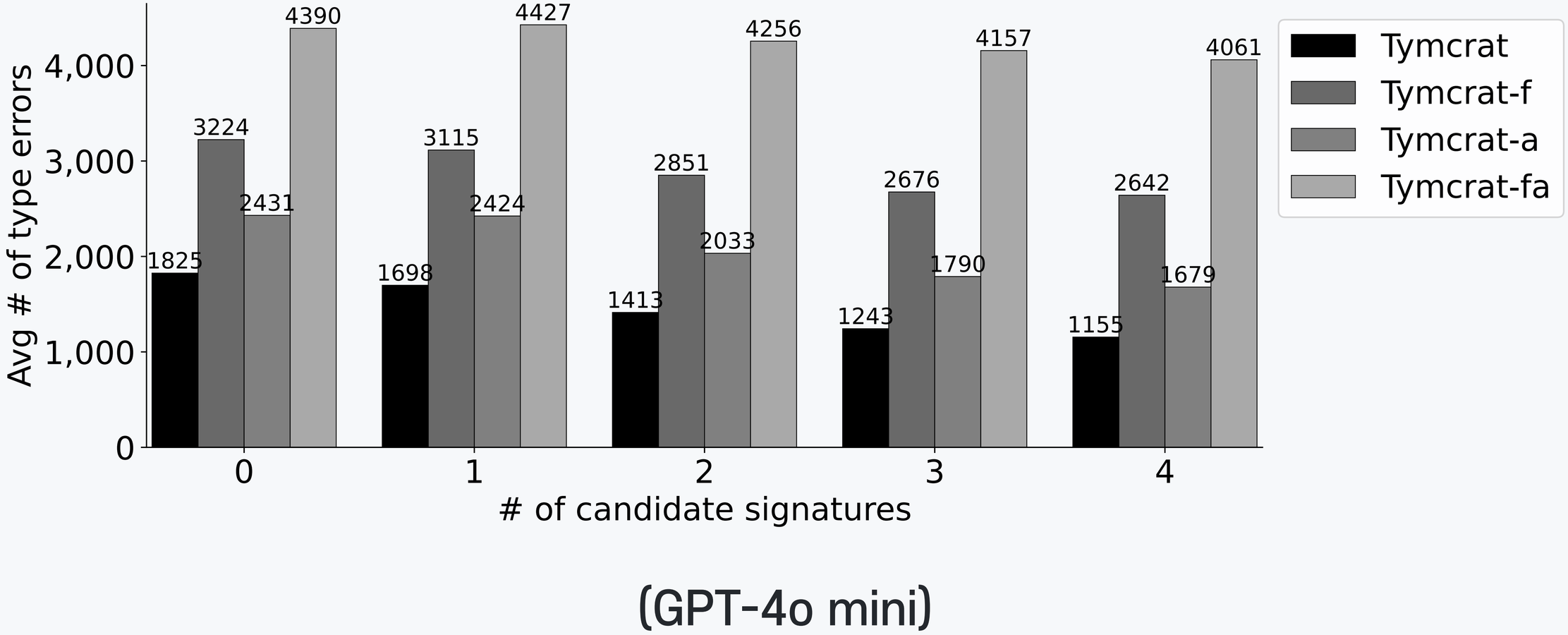


LLM을 통한 번역



Hong and Ryu, Type-Migrating C-to-Rust Translation Using a Large Language Model, EMSE 2025.

LLM을 통한 번역



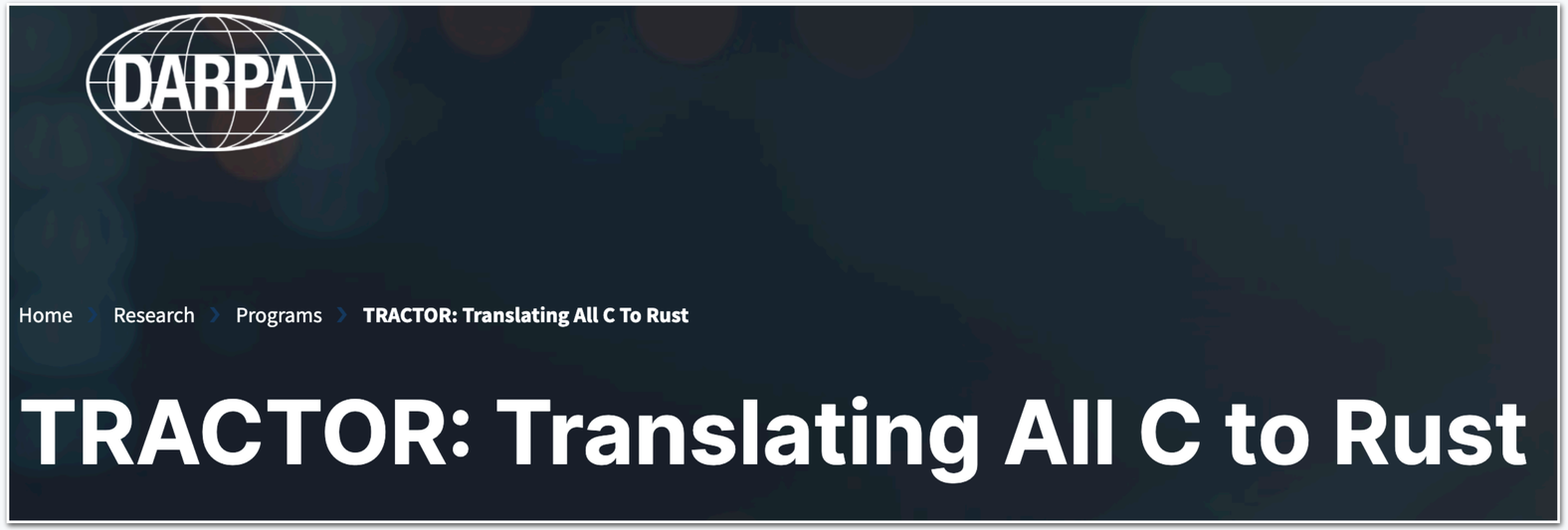
(GPT-4o mini)

Hong and Ryu, Type-Migrating C-to-Rust Translation Using a Large Language Model, EMSE 2025.

LLM을 통한 번역

번역 단위?	추가 정보?	어떻게?	어떻게?
함수 단위	호출되는 함수의 번역	타입 검사	오류 메시지
상호 재귀 함수	정적·동적 분석	테스팅	?
엄청 큰 함수	RAG	생성·비교	

DARPA TRACTOR 과제



기간: 2025년 6월 ~ 2029년 6월

참여 연구팀

MIT(벤치마크 제작)

예일 대학교

Aarno Labs

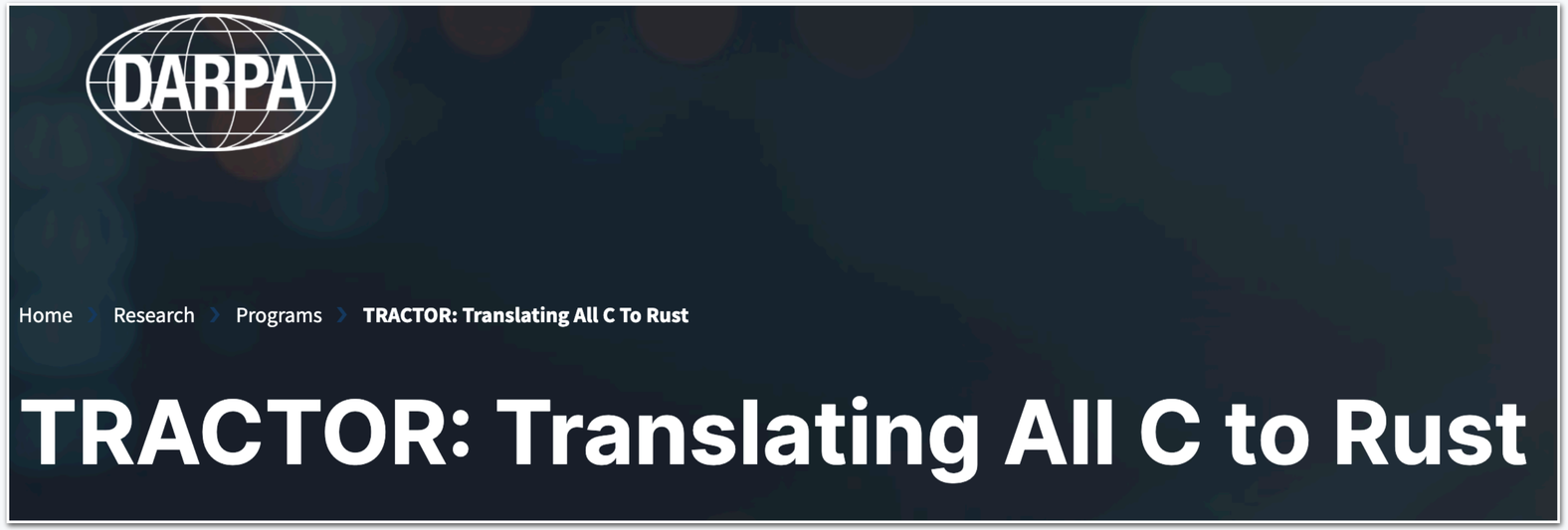
Galois

위스콘신 대학교

워싱턴 대학교

Intel Labs

DARPA TRACTOR 과제



기간: 2025년 6월 ~ 2029년 6월

참여 연구팀

- MIT(벤치마크 제작)
  - 예일 대학교 (정적 분석)
  - Aarno Labs (정적 분석)
  - Galois (C2Rust + LLM)
- 위스콘신 대학교 (LLM)
- 워싱턴 대학교 (LLM)
- Intel Labs (LLM)

DARPA TRACTOR 과제

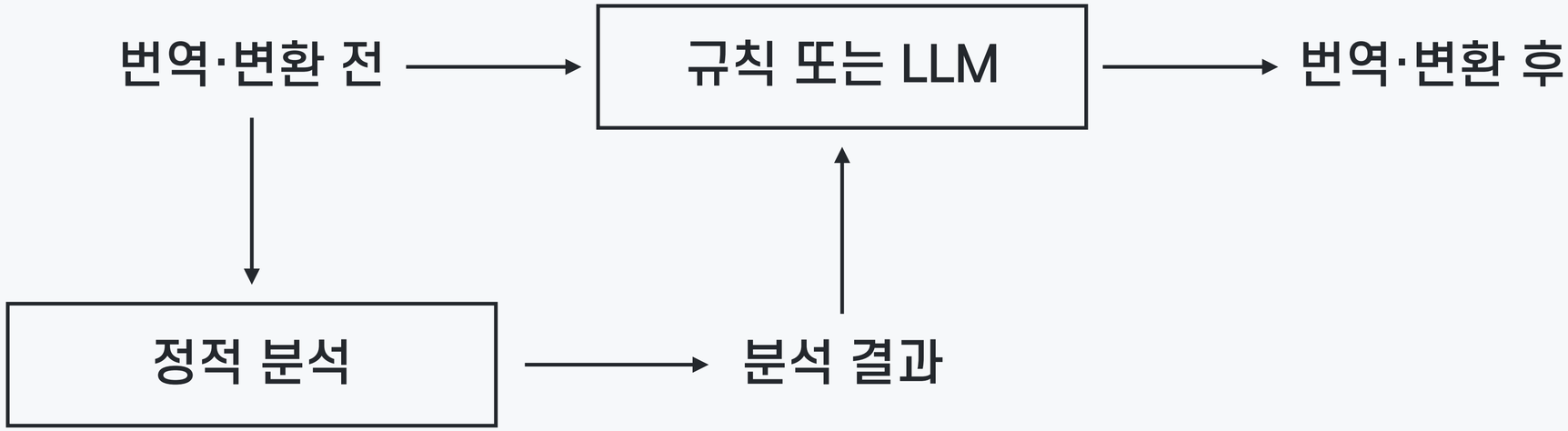
mariusarvinte opened on Dec 11, 2025

It seems that the third argument of `blake256_update` is expressed in bits:

But when this function is called in many places in the driver `app`, it is invoked using *bytes*:

As a side comment, we found Claude Sonnet 4.5 very "capable" of implementing this bug, but GPT-5 systematically fixes it - which causes tests to fail.

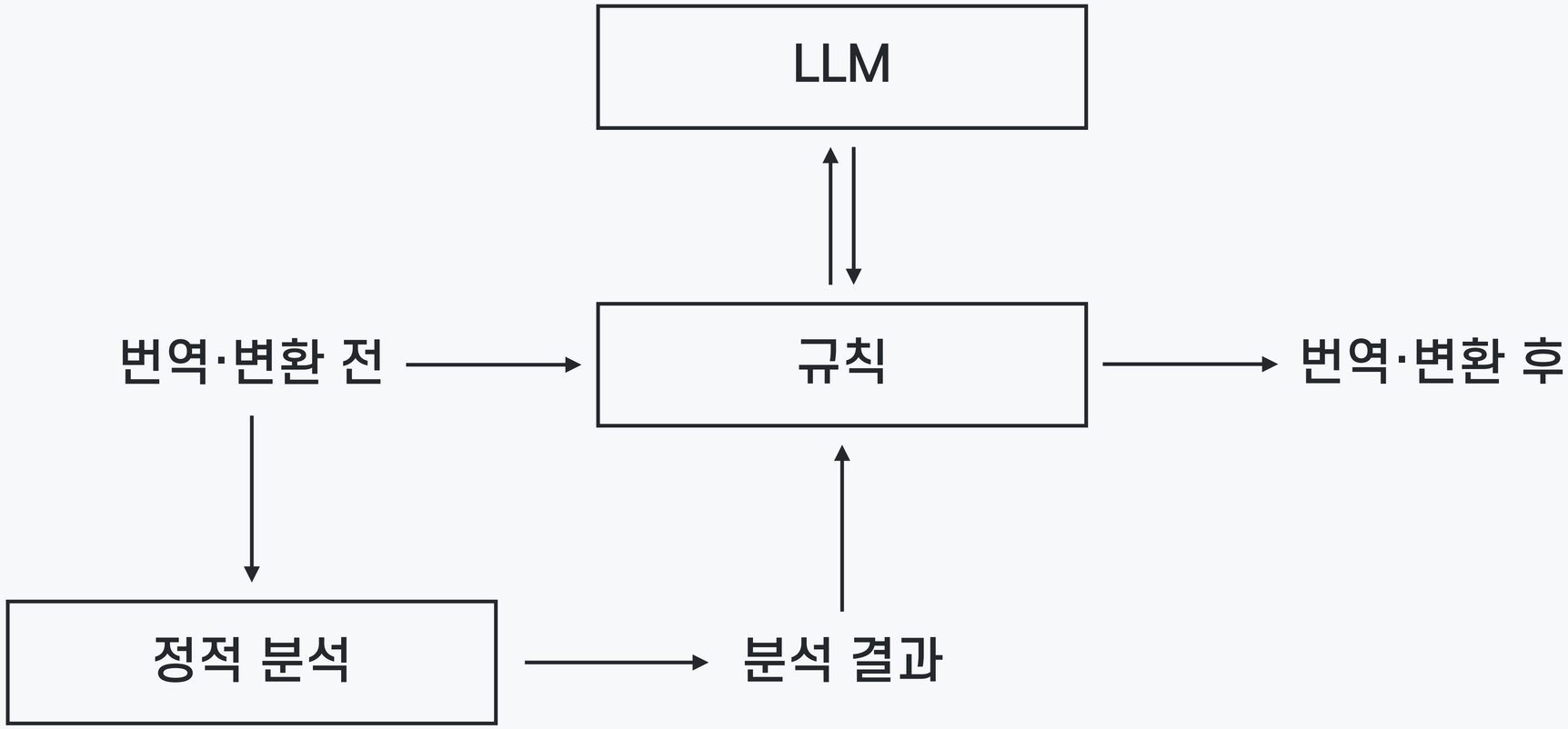
분석과 LLM을 함께 사용하기



규칙 우선: 사람이 작성한 규칙에 따라 번역하되, 특정 부분에 LLM을 사용

LLM 우선: LLM으로 번역하되, 특정 부분에 사람이 작성한 규칙을 사용

규칙 우선 번역 중 LLM 활용하기



규칙 우선 번역 중 LLM 활용하기

Consider the following Rust code:
x = y.offset(1);
where the types are given as follows:
x: *mut i32
y: *mut i32
I have changed the types as follows:
x: Option<&mut i32>
y: Option<Box<[i32]>>
Rewrite the given code in Safe Rust according to these changes while maintaining the semantics and without consuming the ownership. Provide only the code. Do not explain anything.

rust 코드 복사

```rust
x = y.as_deref_mut().and_then(|s| s.get_mut(1));
```

규칙 우선 번역 중 LLM 활용하기

Consider the following Rust code:

```
memcpy(
    x as *mut core::ffi::c_void,
    y as *const core::ffi::c_void,
    (::core::mem::size_of::<i32>() as size_t).wrapping_mul(10 as
size_t),
);
```

where the types are given as follows:

x: *mut i32

y: *mut i32

I have changed the types as follows:

x: Option<&mut [i32]>

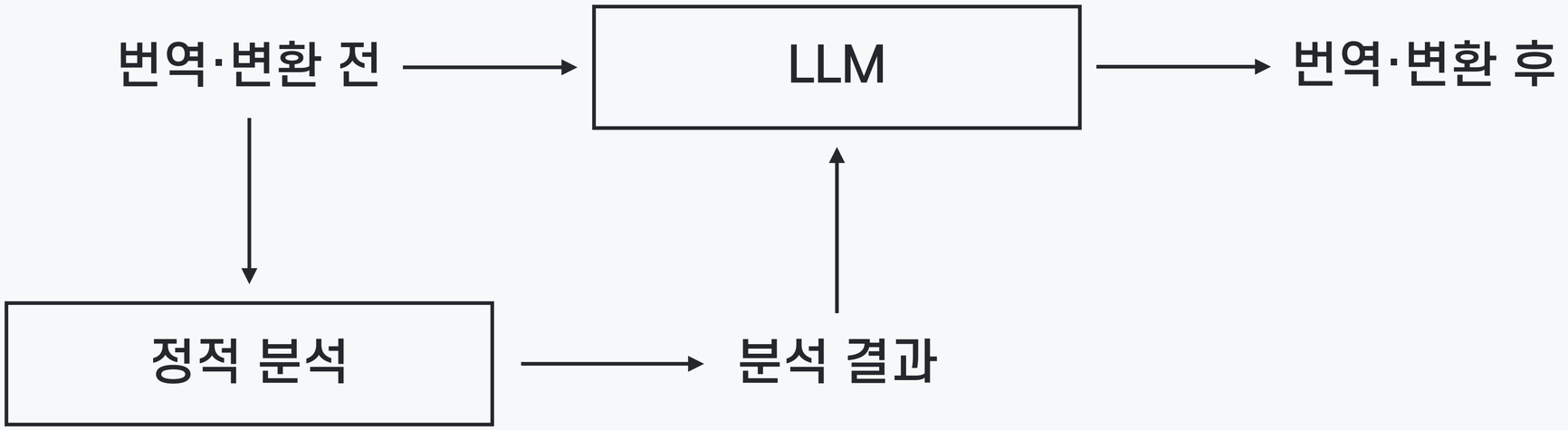
y: Option<Box<[i32]>>

Rewrite the given code in Safe Rust according to these changes while maintaining the semantics and without consuming the ownership. Provide only the code. Do not explain anything.

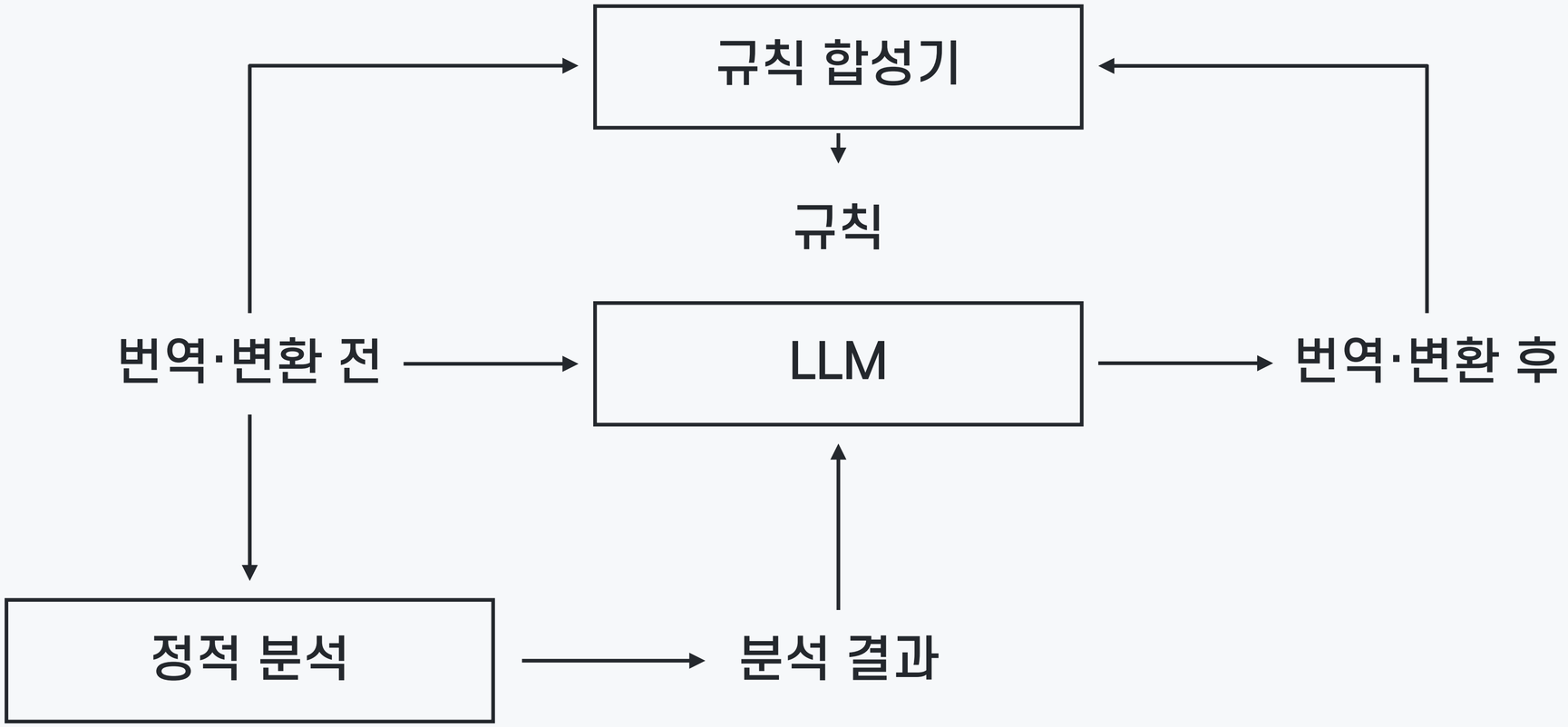
rust 코드 복사

```rust
if let (Some(x), Some(y)) = (x, y.as_deref()) {
    x[..10].copy_from_slice(&y[..10]);
}
```

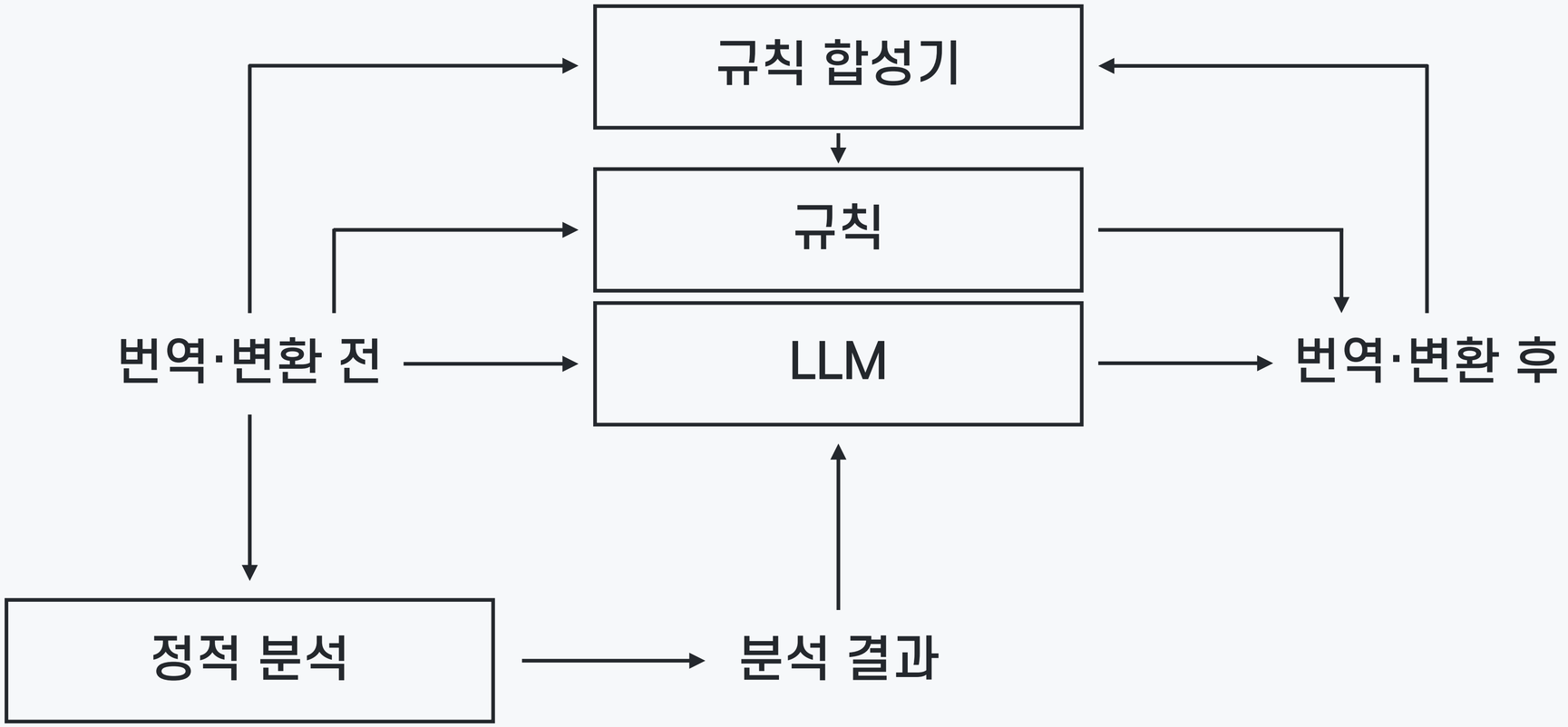
LLM의 번역으로부터 규칙 합성하기

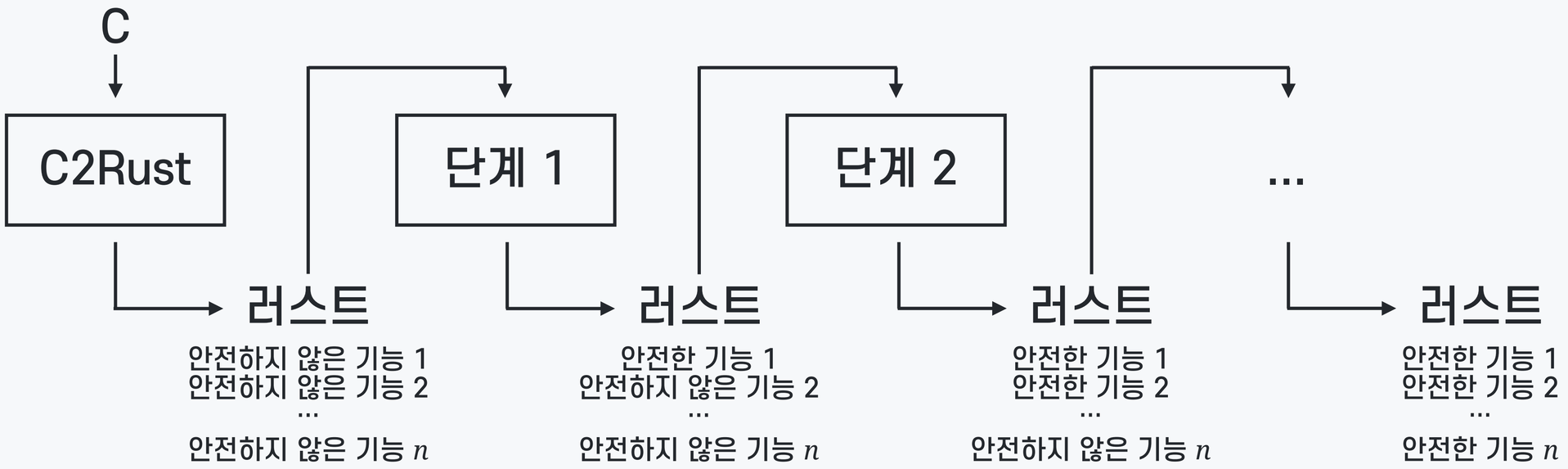


LLM의 번역으로부터 규칙 합성하기



LLM의 번역으로부터 규칙 합성하기





분석이 아주 정확하지 않아도 괜찮다

```
x = y;
```

`*mut i32` `&mut i32` `Option<&mut i32>` `&mut [i32]` `Option<&mut [i32]>`

`Box<i32>` `Option<Box<i32>>` `Box<[i32]>` `Option<Box<[i32]>>`

규칙 우선: 사람이 작성한 규칙에 따라 번역하되, 특정 부분에 LLM을 사용