Empirical Software Engineering (2025) 30:3
https://doi.org/10.1007/s10664-024-10573-2

Type-migrating C-to-Rust translation using a large language model

Jaemin Hong[1] · **Sukyoung Ryu**[1]

Accepted: 10 October 2024 / Published online: 17 October 2024

Abstract
Rust, a modern system programming language, introduces new types that prevent memory bugs and data races. This makes translating legacy system programs from C to Rust a promising approach to enhance their reliability. Since manual code translation is time-consuming, it is desirable to automate the translation. To yield satisfactory results, the translator should have the ability to perform *type migration*, i.e., removing C types and introducing Rust types in the code. In this work, we aim to automatically port an entire C program to Rust by translating each C function to a Rust function with a signature containing proper Rust types through type migration. This goal is challenging because (1) type migration cannot be achieved through syntactic mappings between type names, and (2) after type migration, function bodies should be correctly restructured based on the precise understanding of the functions' behavior. To address these difficulties, we leverage large language models (LLMs), which possess knowledge of program semantics and programming idioms. However, naïvely instructing LLMs to translate each function produces unsatisfactory Rust code, containing unmigrated or improperly migrated types and a huge number of type errors. To resolve these issues, we propose three techniques: (1) generating candidate signatures, (2) providing translated callees' signatures to LLMs, and (3) iteratively fixing type errors using compiler feedback. Our evaluation shows that the proposed approach yields 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.

Keywords Rust · Code translation · Type migration · Large language model

1 Introduction

Rust, a relatively new system programming language, tackles the limitations of C by emphasizing both performance and reliability (Matsakis and Klock 2014; Rust 2022).

Communicated by: Justyna Petke

✉ Jaemin Hong
jaemin.hong@kaist.ac.kr

Sukyoung Ryu
sryu.cs@kaist.ac.kr

[1] School of Computing, KAIST, Daejeon 34141, South Korea

C programs often encounter invalid memory accesses and concurrency bugs, such as data races, even after passing type checking. This has led to legacy system programs written in C suffering from severe bugs and security vulnerabilities (Chen et al. 2011). Conversely, Rust provides a type system and library types that are formally proven to ensure the absence of memory bugs and data races (Jung et al. 2017).

Due to the advantages of Rust, the translation of legacy system programs from C to Rust emerges as a promising approach to enhance their reliability. Developers can unveil previously unknown bugs in their legacy codebases by rewriting the existing C code in Rust and type-checking it Hutt (2021). This approach also mitigates the risk of introducing new bugs while incorporating additional features into the software following the transition to Rust.

Recognizing this potential, the system programming community has embraced this practice, starting the translation of critical system programs to Rust. Mozilla's development of Servo, a web browser written in Rust, is a prime example, as modules from Servo have replaced those of Firefox (Goregaokar 2017). Furthermore, Linux has introduced support for Rust in kernel development with its recent release (De Simone 2022).

Since manual code translation requires significant efforts from programmers, automating the translation process is highly desirable. However, automatic C-to-Rust translation presents challenges beyond merely rewriting C code using Rust syntax. It necessitates a deep understanding of the code and the ability to perform *type migration*, i.e., removing C types and introducing Rust types. Although it is still possible to use C types in Rust code, doing so compromises the safety guarantees of the Rust type checker. Rust introduces new types that do not exist in C, and safety can be ensured by utilizing these types. Therefore, to maximize the benefits of translation, the translator must appropriately migrate types.

In this work, we focus on the problem of migrating types in *function signatures* (parameter and return types) during automatic C-to-Rust 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. This work makes 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 (Zhong et al. 2010; Nguyen et al. 2014), 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.

Due to these difficulties, existing C-to-Rust translators exhibit significantly limited type migration capabilities. They focus only on a few Rust types by using static analysis to decide whether these types can be introduced in signatures. For example, C2Rust (Wingerter 2022)

introduces the never type (Rust 2023d); LAERTES (Emre et al. 2021, 2023) and CROWN (Zhang et al. 2023) introduces pointer types; and Concrat (Hong and Ryu 2023) introduces guard types. Even when used all together, they introduce less than ten Rust types as a result of type migration, leaving most C types unmigrated in the signatures.

Although these techniques have shown promising results for their target Rust types, this approach of utilizing specialized static analysis for each of Rust's numerous types is impractical. Rust offers over 600 types, comprising primitive and library types (Rust 2023b). In our investigation of ten most-starred Rust projects on GitHub, we found that Rust programmers employ at least 120 of these types in function signatures. Given the challenges in the development of the existing tools, designing static analysis for over a hundred remaining Rust types would demand tremendous research efforts.

To address this problem, this work proposes leveraging *large language models* (LLMs), such as ChatGPT (Ouyang et al. 2022; OpenAI 2022), 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 propose using LLMs for type-migrating C-to-Rust translation and identify the challenges in this approach. We also propose techniques to address these challenges. (Section 3)
- We concretize the proposed approach as a tool, Tymcrat (**Ty**pe-**m**igrating **C**-to-**R**ust **a**utomatic **t**ranslator), 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 4)

We then discuss related work (Section 5) and conclude (Section 6).

2 Background

In this section, we explain the difficulties of type migration for function signatures in C-to-Rust translation (Section 2.1) and discuss the limitations of exsiting approaches (Section 2.2).

2.1 Difficulties of Type Migration

Type migration in C-to-Rust translation is challenging for various reasons. The primary issue arises from the *m*-to-*n* correspondence between C and Rust types, making it difficult to determine the appropriate Rust type to replace a given C type. Even worse, a type may change its location in the signature during migration. To illustrate this, we present two C functions with the same pointer type in their signatures and their respective translations. In one function, the pointer type should be migrated to a reference type; in the other, it should be migrated to an `Option` type, while also adjusting its location.

We first see a case where a C pointer type is migrated to a reference type. The following C function receives an integer pointer and increases its value by one:

```
int inc(int *p) {
    *p += 1;
    return *p;
}
```

Without type migration, this function is translated as follows:

```
fn inc(p: *mut i32) -> i32 {
    *p += 1;
    *p
}
```

Here, `*mut i32` is the Rust syntax for the C type `int*`.

This translation lacks safety guarantees through type checking because the Rust compiler does not ensure the safety of a C pointer type.

As a result, Rust programmers prefer to migrate this type to a reference type, denoted as `&mut i32`, as shown below:

```
fn inc(p: &mut i32) -> i32 {
    *p += 1;
    *p
}
```

By using a reference type, the type checker can ensure the safety of dereferencing the pointer.

We now see a case where a C pointer type is migrated to an `Option` type.

In C, pointers also serve the purpose of implementing *partial functions*, i.e., functions that fail for some inputs.

For instance, consider a function that computes the quotient resulting from the division of two integers. This function is a partial function because it fails when the divisor is zero, and its implementation is as follows:

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

The function takes a dividend `n` and a divisor `d`, along with a pointer `q` for producing the quotient. Upon success, it writes the result to `q`; otherwise, it writes nothing. The return value does not convey the actual result but rather serves as an indicator of success or failure. In this example, `0` represents success, and `1` represents failure.

When translating this function, migrating `int *` to `&mut i32` is not considered correct by most Rust programmers because they prefer implementing partial functions using `Option` types, as suggested by the Rust standard library (Rust 2023c).

Therefore, the function should be translated as follows:

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

Note that `int *` is migrated to `Option<i32>` and moved to the return type. When `d` is zero, the function returns `None` to represent failure; otherwise, it returns `Some` containing the quotient.

As these two translations show, the same type `int *` can be migrated to either `&mut i32` or `Option<i32>` depending on how the function uses the parameter. To properly migrate types, the translator must consider the function's semantics instead of blindly replacing every occurrence of a C type with a specific Rust type.

The second difficulty arises from restructuring a function body after migrating types in the signature. In the previous translation of `div`, `return 1` becomes `return None`, and `*q = n / d; return 0` becomes `Some(n / d)`. For this, the translator needs to determine the success or failure of the function at each return point.

Furthermore, correctly restructuring the body of a caller of `div` is also challenging.

Below is typical C code calling `div`:

```
int q;
if (div(10, 3, &q) == 0) {
    // use `q`
} else {
    // handle the failure
}
```

This code declares a variable `q` and passes its pointer to `div`. After `div` returns, the caller checks the return value and reads `q` only if the return value is zero; otherwise, it handles the failure.

The correct translation of this code is as follows:

```
match div(10, 3) {
    Some(q) => { /* use `q` */ }
    None => { /* handle the failure */ }
}
```

The Rust code does not pass a pointer to `div` and uses pattern matching on the returned `Option` value to distinguish success and failure. This translation requires the information that

`int *` has been migrated to `Option<i32>`. In addition, the translator must handle various C code patterns to produce correct Rust code.

In this example, the caller compares the return value with 0, making it easy to decide which branch of `if` should be mapped to which branch of `match`. However, the caller may compare the return value with another integer or assign the return value to a variable and use the variable in a more complex comparison. Dealing with all potential code patterns using a predefined set of syntactic rewriting rules would demand huge manual efforts and risk missing some patterns.

2.2 Limitations of Existing Approaches

2.2.1 API Mapping Mining

Researchers have proposed API mapping mining (Zhong et al. 2010; Nguyen et al. 2014) to extract type mappings from existing code for Java-to-C# translation.

This is beneficial as Java and C# types often exhibit a one-to-one correspondence. Using the mined mappings, a translator can easily migrate Java types to the corresponding C# types by replacing occurrences of each Java type. Additionally, API mapping mining can extract mappings between library methods as well, allowing the translator to correctly rewrite function bodies by substituting calls to Java methods with their C# counterparts.

However, this approach is not applicable to C-to-Rust translation. Type mappings are insufficient due to the *m*-to-*n* correspondence between C and Rust types. The translator must carefully select Rust types, considering not only C types but also functions' semantics. Furthermore, mappings between library functions are not enough because restructuring function bodies in C-to-Rust translation requires more than just substituting function names.

2.2.2 C-to-Rust Translation

Existing C-to-Rust translators migrate C types to a restricted set of Rust types. They employ syntactic rules and static analysis to determine whether the target Rust types can be introduced to a certain signature.

C2Rust (Wingerter 2022) is a syntactic C-to-Rust translator that aims to maintain syntactic fidelity. Since it does not perform semantic analysis, it generates code utilizing only two Rust types that can be introduced syntactically. First, C2Rust migrates the return type of a function to the never type (Rust 2023d) to represent that the function never returns if the original C function has the `noreturn` attribute. Second, it wraps every function pointer type in `Option` to express the possibility of null pointers. In Rust, `Option` is used not only for partial functions but also for nullable pointers. However, the migration is unsatisfactory even for these two types. No-returning C functions often lack the `noreturn` attribute, and C2Rust cannot migrate the return types of such functions. In addition, when a program passes only non-null function pointers to a certain function, the `Option` type added to its signature is meaningless.

LAERTES (Emre et al. 2021, 2023) and CROWN (Zhang et al. 2023) translate C code to Rust, initially using C2Rust and then transforming the generated Rust code. Their objective is to migrate C pointer types to Rust pointer types, which include references, slices, and `Box` types. These are pointer types whose safety is guaranteed by the type checker. A reference represents any pointer, a slice points to an array, and a `Box` points to a heap-allocated object. To achieve such type migration, LAERTES identifies pointers that can be migrated without causing type errors, utilizing the analyses of the Rust compiler, while CROWN performs an

ownership analysis to determine whether a pointer owns a certain heap-allocated object. Although they introduce `Option`, it is not intended to address partial functions. Like C2Rust, they wrap every Rust pointer type in `Option` to indicate the potential presence of null pointers, but some of these `Option` types might be unnecessary.

Concrat (Hong and Ryu 2023) also transforms C2Rust-generated code, but its goal is to migrate lock-related types used by concurrent programs. In Rust, a guard type serves as evidence of holding a lock and plays a key role in type checking to ensure the absence of data races. Concrat performs static analysis to determine which locks are held at which program points and adds guard types to signatures based on the analysis results. It also introduces tuple types when a single function returns multiple guards. However, it lacks a general ability to introduce tuple types for functions producing multiple values other than guards.

3 Type-Migrating Translation via LLM

This section presents techniques to translate C to Rust while migrating types in signatures and minimizing type errors. As illustrated in Section 2, 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 (Chen et al. 2021a; Dong et al. 2024; Li et al. 2023a, b; Yang et al. 2024a), program repair (Fan et al. 2023; Xia et al. 2023), and code summarization (Ahmed and Devanbu 2023). 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 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 (Nguyen et al. 2013; Karaivanov et al. 2014; Nguyen et al. 2015; Chen et al. 2018; Roziere et al. 2020). 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 3.1). Second, we augment the function with Rust signatures of its callees and translate it to Rust for each candidate signature (Section 3.2). Third, we type-check the translated code and iterate to resolve type errors using compiler feedback (Section 3.3). Finally, we select the most suitable translation (Section 3.4) 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.

Note that examples in this section are prompts and responses from ChatGPT (Ouyang et al. 2022; OpenAI 2022) 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.

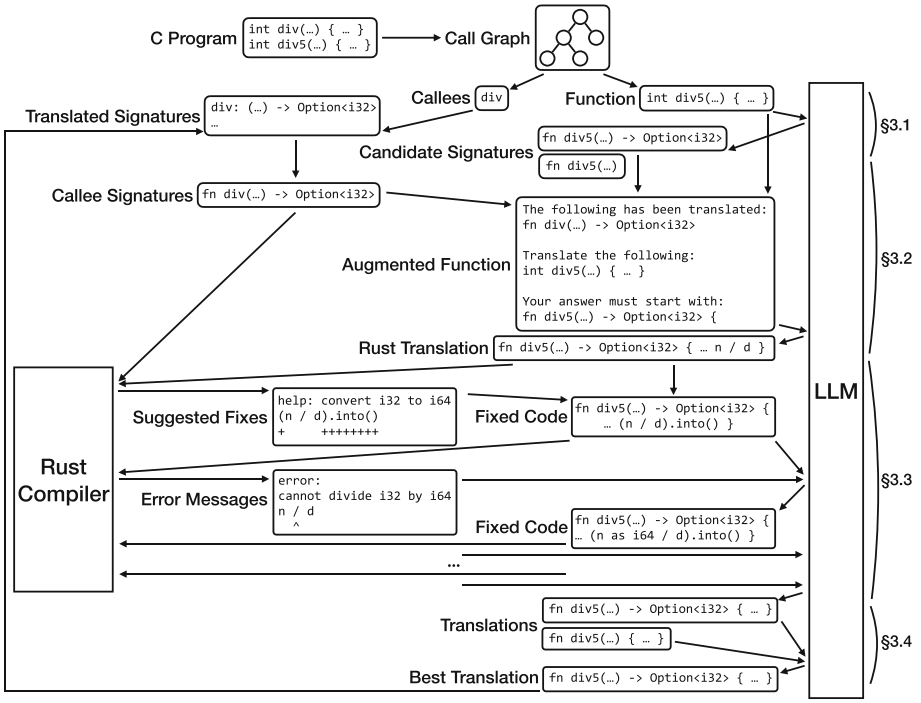


Fig. 1 Overview of type-migrating translation via LLM

3.1 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 (Mialon et al. 2023), e.g., incorporating a step-by-step thinking prompt (Kojima et al. 2022). 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.

3.2 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.

3.3 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)
}
```

Algorithm 1 Fix-by-suggestion algorithm.

```
1 Function fix-by-suggestion begin
      Input  : code
      Output : code, errors_no-fix
2     errors_fix, errors_no-fix ← type-check(code);
3     while errors_fix is not empty do
4         code ← apply-fix(code, errors_fix);
5         errors_fix, errors_no-fix ← type-check(code);
6     end
7 end
```

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)
}
```

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 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

Algorithm 2 Fix-by-LLM algorithm.

```
1  Function fix-by-llm begin
       Input  : code
       Output : code, errors
2      code, errors ← fix-by-suggestion(code);
3      while errors is not empty do
4          code′ ← llm-fix(code, errors);
5          code′, errors′ ← fix-by-suggestion(code′);
6          if length(errors) ≤ length(errors′) then
7              break;
8          end
9          code, errors ← code′, errors′;
10     end
11 end
```

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 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 9). The iteration continues as long as the fix is successful, terminating when no type errors remain (line 3).

3.4 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`.

4 Evaluation

In this section, we first provide an overview of our implementation (Section 4.1) and the process of collecting benchmark programs (Section 4.2). 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 4.3)
- RQ2. Quality of type migration: Do the Rust types introduced by the proposed approach adhere to Rust idioms? (Section 4.4)

– RQ3. Type error reduction: Does the proposed approach effectively reduce type errors by augmenting functions and iteratively fixing errors? (Section 4.5)
– RQ4. Comparison with existing approaches: How does the translation of the proposed approach differ from that of the existing approaches? (Section 4.6)
– RQ5. Overhead: Does the proposed approach entail reasonable overhead? (Section 4.7)

Finally, we dicuss threats to validity (Section 4.8).

4.1 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 (OpenAI 2024). 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 (OpenAI 2023). 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.

4.2 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

Table 1 The size of each benchmark program

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

the packages listed in GNU Package Blurbs (GNU 2023). We then filtered out programs that exceeded 100,000 lines, as measured by `cloc` (Danial 2023), 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 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.

4.3 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.

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` (Rust 2023a) 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 2 illustrates the average number of Rust types introduced in benchmark programs translated by Tymcrat_0 to Tymcrat_4.

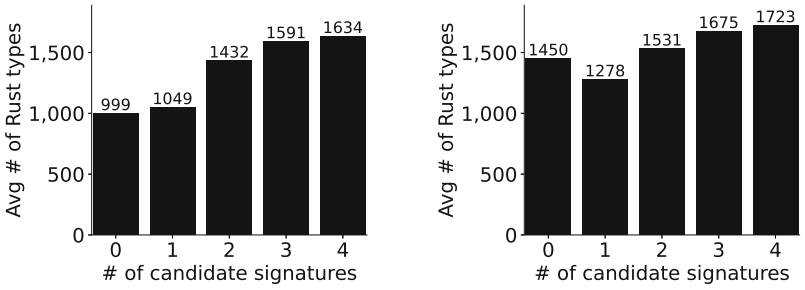
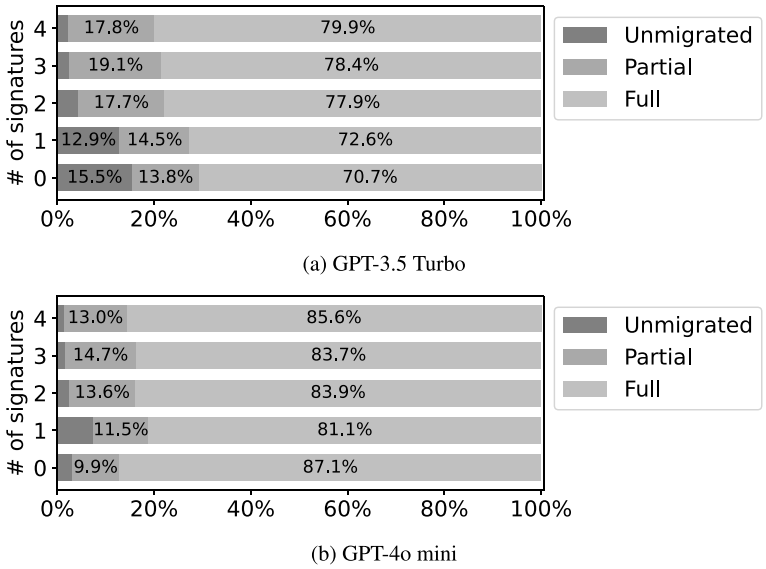


Fig. 2 RQ1: Promotion of Type Migration. Average number of Rust types introduced in each GNU package after translation using GPT-3.5 Turbo or GPT-4o mini, generating 0 to 4 candidate signatures per function

When using GPT-3.5 Turbo (Fig. 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 4.7, 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 (Fig. 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 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 3 depicts the proportions of unmigrated, partially-migrated, and fully-migrated signatures in all benchmark programs translated by Tymcrat$_0$ to Tymcrat$_4$.



(a) GPT-3.5 Turbo

(b) GPT-4o mini

Fig. 3 RQ1: Promotion of Type Migration. Proportions of unmigrated, partially migrated, and fully migrated function signatures in the GNU packages after translation using GPT-3.5 Turbo or GPT-4o mini, generating 0 to 4 candidate signatures per function

When using GPT-3.5 Turbo (Fig. 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 (Fig. 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 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.

4.4 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 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 (Fig. 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 (Fig. 4b), Tymcrat_0 already introduces 57 distinct types, outperforming Tymcrat_1 to Tymcrat_3. This indicates that generating fewer than four candidates

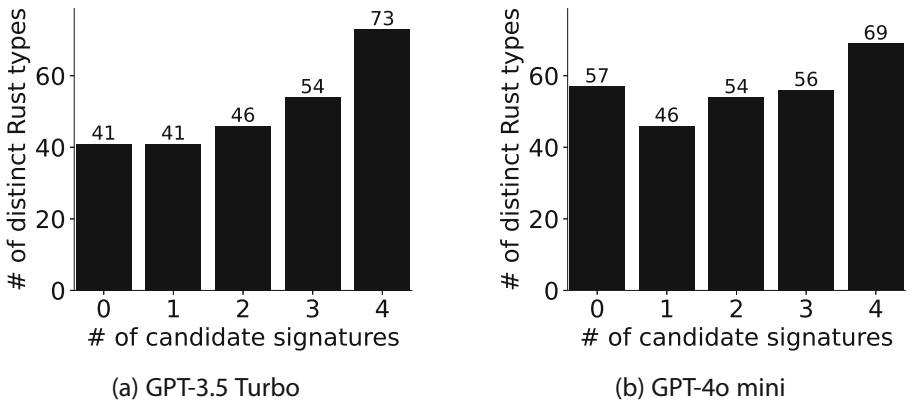


Fig. 4 RQ2: Quality of Type Migration. Number of distinct Rust types introduced in the GNU packages after translation using GPT-3.5 Turbo or GPT-4o mini, generating 0 to 4 candidate signatures per function

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 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

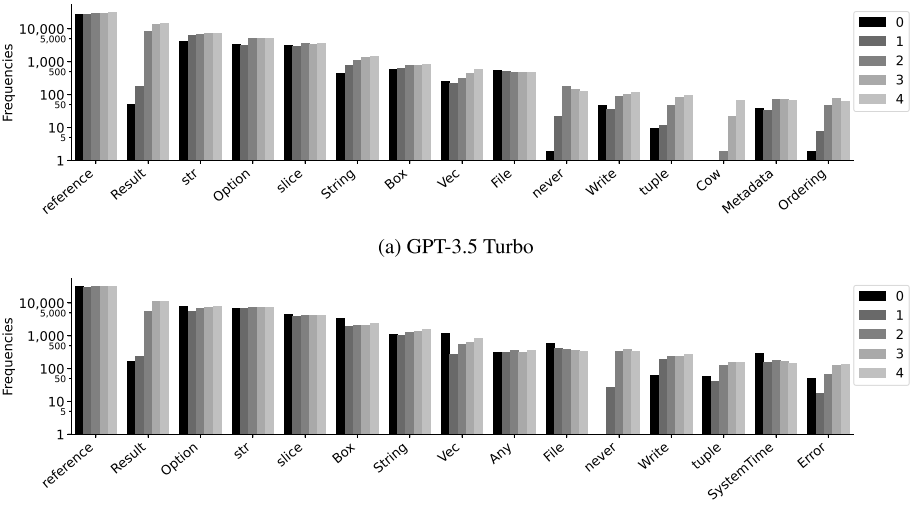


Fig. 5 RQ2: Quality of Type Migration. Frequencies of each introduced Rust type in the GNU packages after translation using GPT-3.5 Turbo or GPT-4o mini, generating 0 to 4 candidate signatures per function. The y-axis is log scale

list from GPT-3.5 Turbo, whereas `Any`, `SystemTime`, and `Error` appear only in the list from GPT-4o mini.

Figure 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 (Fig. 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 (Fig. 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 3.4, 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.

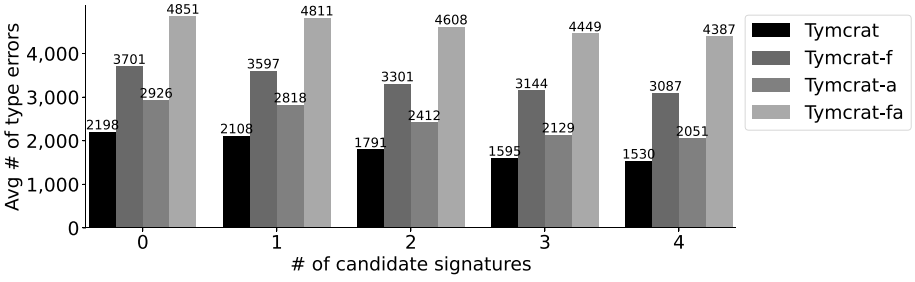
These examples demonstrate that using LLMs for candidate signature generation and best translation selection does not always yield accurate results. LLMs often produce signatures with non-idiomatic types and may favor translations with such signatures during selection. This suggests that adopting heuristic 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

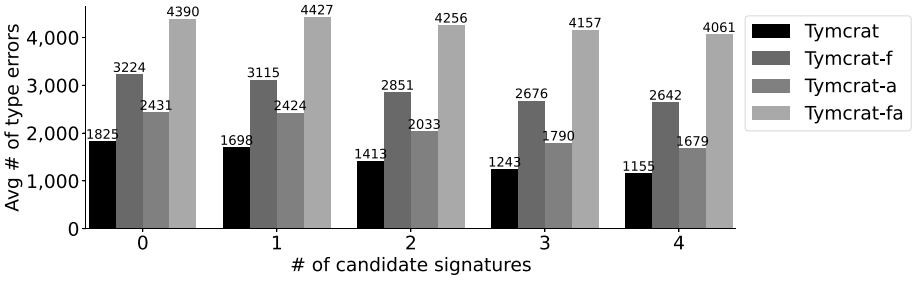
- Generating two or more candidate signatures enhances the diversity of introduced Rust types when using GPT-3.5 Turbo, but four candidates are needed for a similar effect with GPT-4o mini.
- Most of the top fifteen types in the translated code also frequently appear in real-world Rust code.
- Generating 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`.

4.5 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



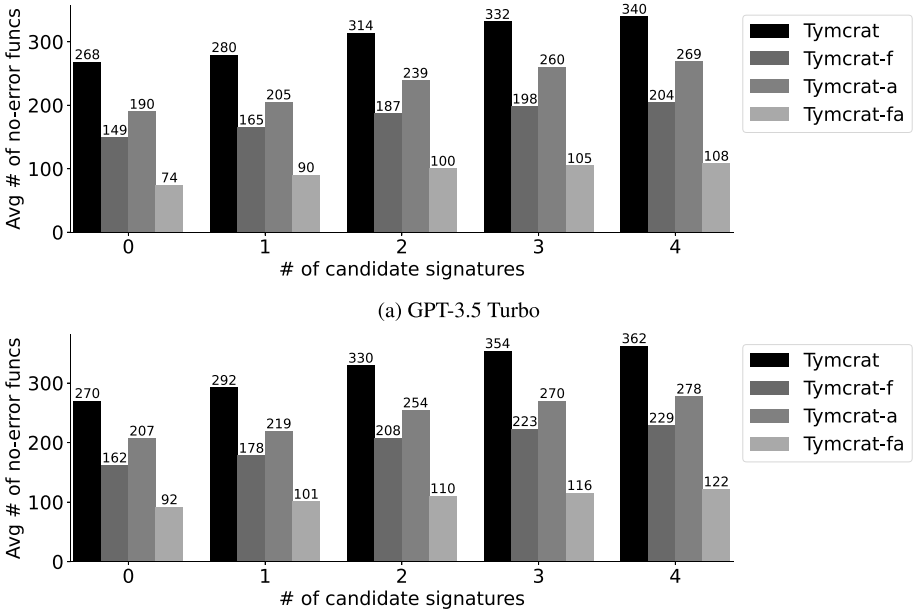
(b) GPT-4o mini

Fig. 6 RQ3: Type Error Reduction. Average number of type errors in each GNU package after translation using GPT-3.5 Turbo or GPT-4o mini, generating 0 to 4 candidate signatures per function and selectively enabling function augmentation and error fixes

Figure 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 (Fig. 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 (Fig. 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.



(a) GPT-3.5 Turbo

(b) GPT-4o mini

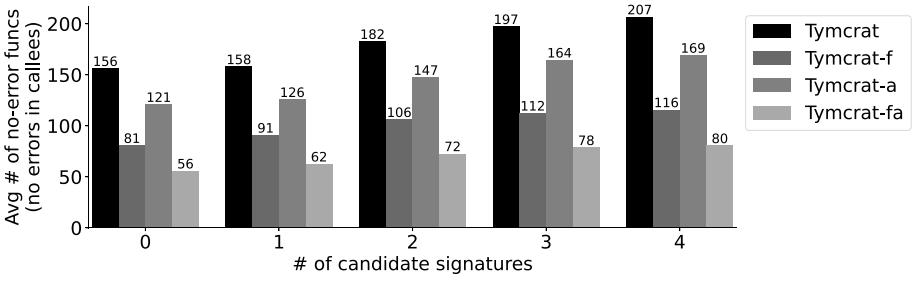
Fig. 7 RQ3: Type Error Reduction. Average number of functions without type errors in each GNU package after translation using GPT-3.5 Turbo or GPT-4o mini, generating 0 to 4 candidate signatures per function and selectively enabling function augmentation and error fixes

Nevertheless, even with Tymcrat$_4$ using GPT-4o mini, each program exhibits an average of 1,155.3 type errors, which is still a significant number. Further research is needed to reduce the number of type errors to a manageable level.

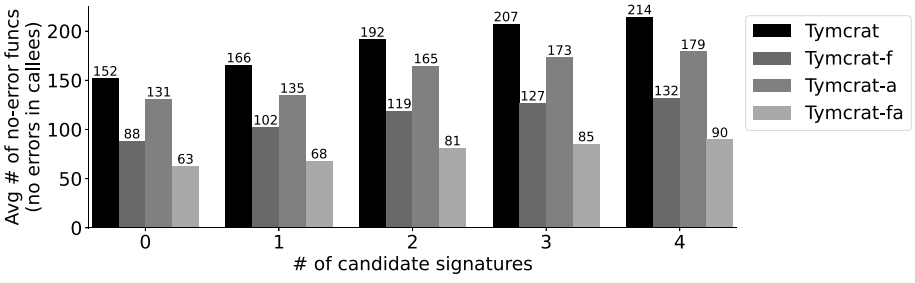
We further investigate the effectiveness of the proposed approach in reducing type errors by examining the number of functions without type errors. Figure 7 shows the average number of such functions in benchmark programs translated by each setting. The results demonstrate that our techniques significantly increase the number of functions without type errors by reducing type errors. When using GPT-3.5 Turbo (Fig. 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 (Fig. 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 8 shows the average number of such functions in benchmark programs translated by each setting. The overall trend is similar to that in Fig. 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.



(a) GPT-3.5 Turbo



(b) GPT-4o mini

Fig. 8 RQ3: Type Error Reduction. Average number of functions that do not have type errors in themselves and their callees in each GNU package after translation using GPT-3.5 Turbo or GPT-4o mini, generating 0 to 4 candidate signatures per function and selectively enabling function augmentation and error fixes

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

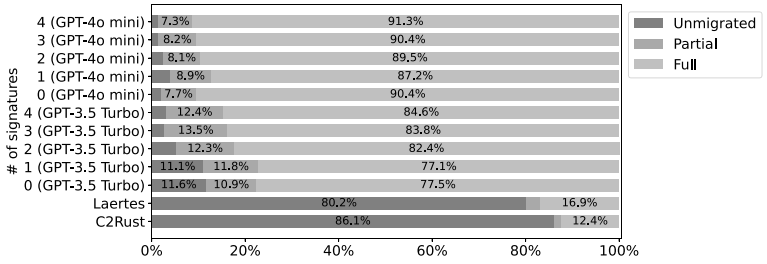
4.6 RQ4: Comparison with Existing Approaches

We compare the translation of the proposed approach with that of existing C-to-Rust translators: C2Rust, Laertes, Crown, and Concrat. 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

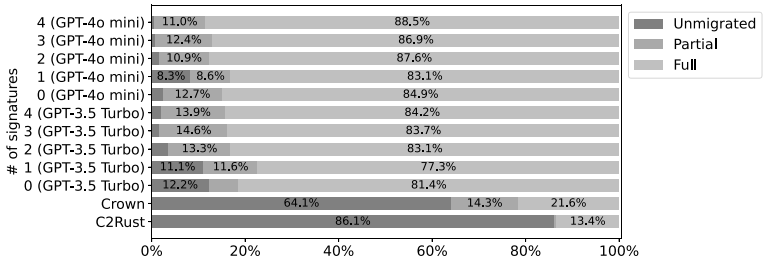
and can be used in a complementary manner. We believe that future work is required to unify these directions and leverage the best of both worlds.

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 (Emre et al. 2023), resulting in a total of 15 programs.

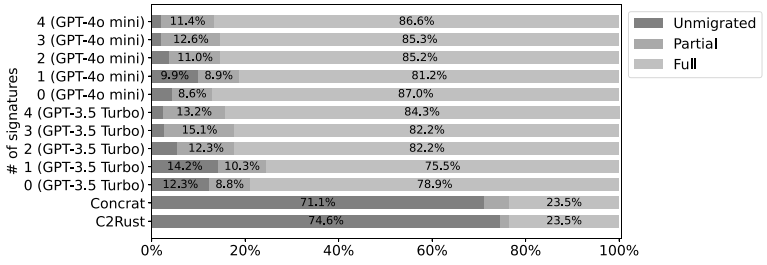
Figure 9a depicts the proportions of different kinds of signatures in the 15 programs translated by C2Rust, LAERTES, or Tymcrat$_0$ to Tymcrat$_4$ using GPT-3.5 Turbo or GPT-4o mini. Our approach provides significantly better type migration capabilities compared to C2Rust and LAERTES, regardless of the number of candidates. C2Rust introduces only never and



(a) Comparison with LAERTES



(b) Comparison with CROWN



(c) Comparison with Concrat

Fig. 9 RQ4: Comparison with Existing Approaches. Proportions of unmigrated, partially migrated, and fully migrated function signatures in the benchmark programs after translation with C2Rust, LAERTES, CROWN, Concrat, or Tymcrat using GPT-3.5 Turbo or GPT-4o mini, generating 0 to 4 candidate signatures per function

`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 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 (Zhang et al. 2023).

Figure 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.

We now compare the proposed approach with Concrat. Concrat also works by transforming compilable Rust code generated by C2Rust, and we cannot use the 32 benchmark programs that C2Rust fails to translate to compilable code. Moreover, while Concrat only migrates lock-related types, none of the remaining 9 programs use locks. For this reason, we employ 46 programs used in the Concrat paper (Hong and Ryu 2023).

Figure 9c displays the proportions of signatures in 46 programs translated by C2Rust, Concrat, or Tymcrat_0 to Tymcrat_4 using GPT-3.5 Turbo or GPT-4o mini. Our approach migrates significantly more types than Concrat. Specifically, C2Rust fully migrates 23.5% and partially migrates 1.9% of the signatures, and Concrat increases the portion of partially migrated signatures to 5.4% by introducing guard and tuple types. On the other hand, Tymcrat_4 using GPT-4o mini fully migrates 86.6% of the signatures. However, despite lacking general type migration capabilities, Concrat introduces guard types more frequently than our approach, highlighting the advantages of its design tailored to lock-related types.

Summary of RQ4: Comparison with Existing Approaches

- The proposed approach migrates more signatures than existing approaches, including C2Rust, LAERTES, CROWN, and Concrat.
- LAERTES and CROWN do not introduce type errors during translation, while the proposed approach may introduce them.
- Concrat is tailored to lock-related types and introduces guard types more frequently than the proposed approach.
- Future research should explore the possibility of combining LLM-based translation with static analysis-based translation to effectively migrate types while avoiding type errors.

4.7 RQ5: Overhead

To evaluate the performance overhead of the proposed approach, we investigate the following settings: Tymcrat$_n$, Tymcrat$_n^{-f}$, Tymcrat$_n^{-a}$, and Tymcrat$_n^{-fa}$, with n ranging from 0 to 4. Figure 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 (Fig. 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 translation attempt. Additionally, comparing Tymcrat$_4^{-fa}$ and Tymcrat$_4^{-a}$, 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$_n^{-fa}$ and Tymcrat$_n^{-f}$ exhibit similar translation times for any n because providing callee signatures does not prolong the translation process. Moreover, Tymcrat$_n$ is faster than Tymcrat$_n^{-a}$, 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 (Fig. 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

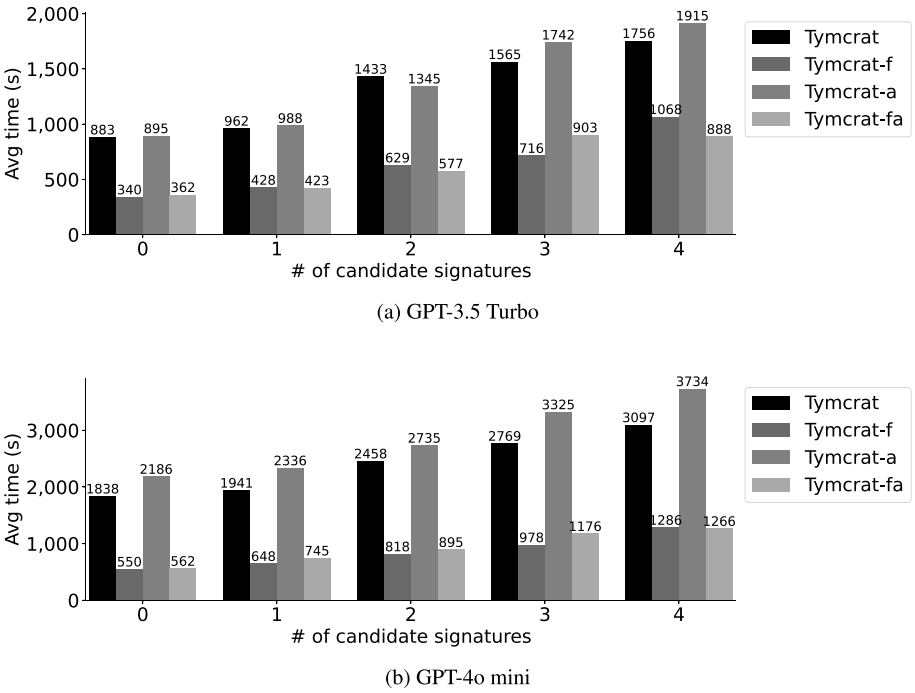
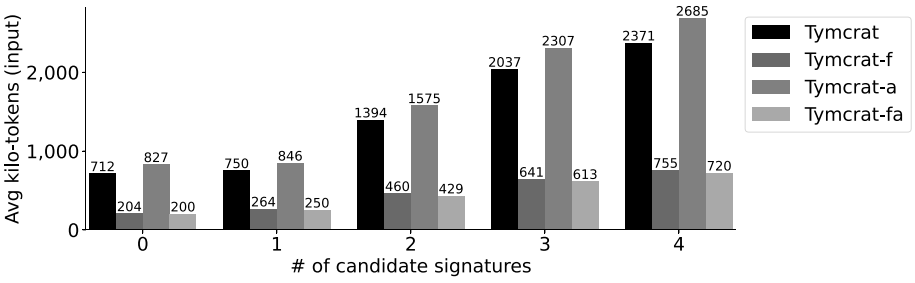
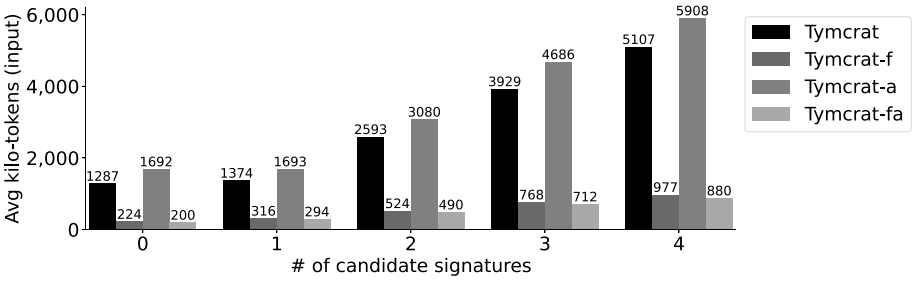


Fig. 10 RQ5: Overhead. Average time taken to translate each GNU package using GPT-3.5 Turbo or GPT-4o mini, generating 0 to 4 candidate signatures per function and selectively enabling function augmentation and error fixes



(a) GPT-3.5 Turbo



(b) GPT-4o mini

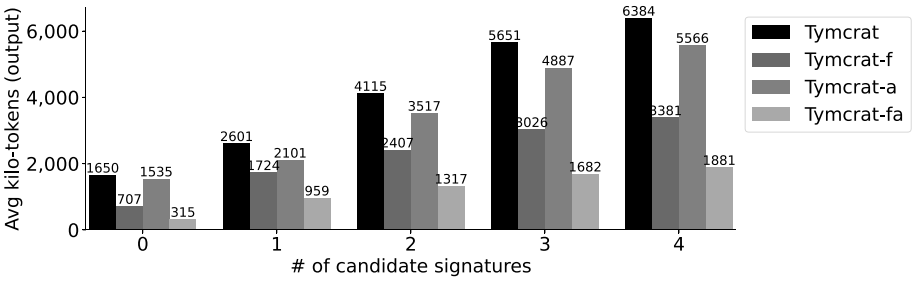
Fig. 11 RQ5: Overhead. Average number of input tokens required to translate each GNU package using GPT-3.5 Turbo or GPT-4o mini, generating 0 to 4 candidate signatures per function and selectively enabling function augmentation and error fixes

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%.

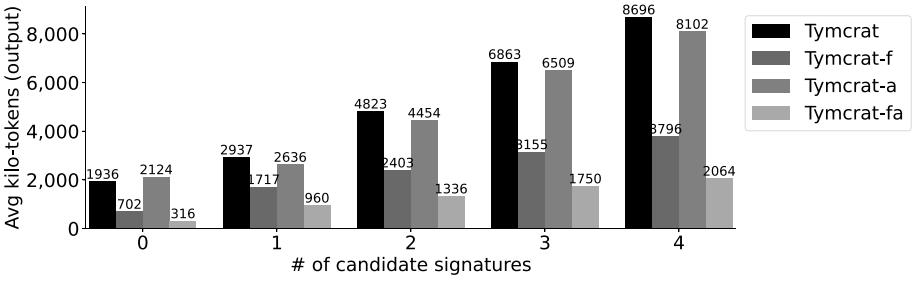
We also report important statistics to further understand the runtime of the proposed approach. When using Tymcrat$_4$ with GPT-3.5 Turbo, the minimum and maximum translation times are 150 seconds for `time-1.9` and 5,474 seconds for `gawk-5.2.2`, with a standard deviation of 1,446 seconds. When using 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 11 shows the average 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 (Fig. 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 (Fig. 11b), candidate signature generation and error fixes increase input tokens by 296.7% and 571.1%, respectively.

Figure 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 (Fig. 12a), candidate signature generation and error



(a) GPT-3.5 Turbo



(b) GPT-4o mini

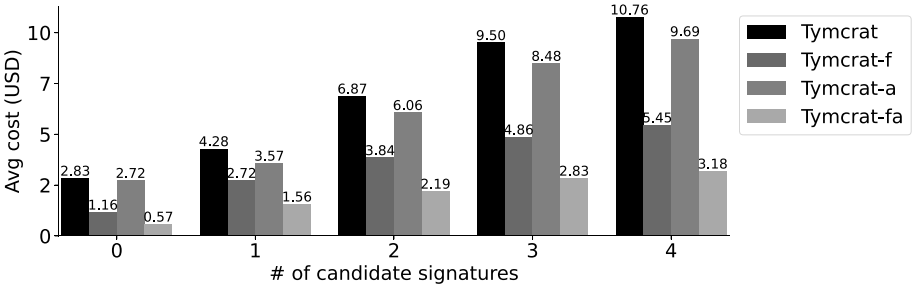
Fig. 12 RQ5: Overhead. Average number of output tokens required to translate each GNU package using GPT-3.5 Turbo or GPT-4o mini, generating 0 to 4 candidate signatures per function and selectively enabling function augmentation and error fixes

fixes increase the number of output tokens by 287.0% and 195.9%, respectively. When using GPT-4o mini (Fig. 12b), the increases are 349.1% and 292.5%, respectively.

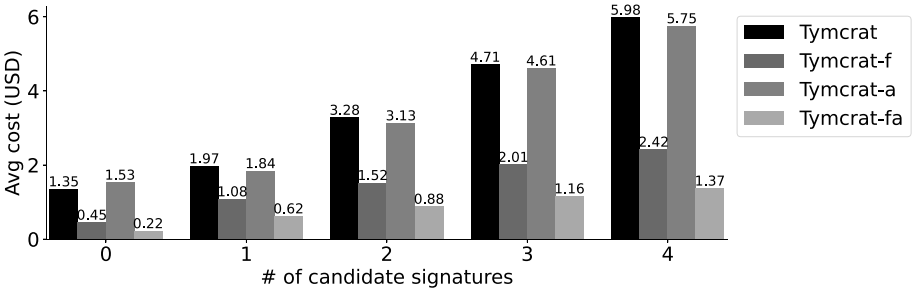
Finally, we investigate the monetary cost of the proposed approach. The OpenAI API charges different prices for input and output tokens. The cost for gpt-3.5-turbo-0125 is $0.50 per 1M input tokens and $1.50 per 1M output tokens, while the cost for gpt-4o-mini-2024-07-18 is $0.15 per 1M input tokens and $0.60 per 1M output tokens. Figure 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.

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.



(a) GPT-3.5 Turbo



(b) GPT-4o mini

Fig. 13 RQ5: Overhead. Average cost required to translate the GNU packages using GPT-3.5 Turbo or GPT-4o mini, generating 0 to 4 candidate signatures per function and selectively enabling function augmentation and error fixes

4.8 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 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.

5 Related Work

In this section, we focus on neural machine translation of programming languages and the evaluation of code translated by neural networks. Existing techniques for translation without machine learning are covered in Section 2.2.

5.1 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# (Nguyen et al. 2013; Karaivanov et al. 2014; Nguyen et al. 2015; Chen et al. 2018). To address this limitation, TransCoder (Roziere et al. 2020) introduces unsupervised learning to programming language translation, enabling training with monolingual code bases. Further studies (Lachaux et al. 2021; Roziere et al. 2022; Szafraniec et al. 2023) enhance TransCoder's translation capabilities by utilizing obfuscated code, unit tests, and compilers' intermediate code representation during training. SDA-Trans (Liu et al. 2023) 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 (Wang et al. 2021; Chen et al. 2021b; Feng et al. 2020; Guo et al. 2021), which can be effectively leveraged by our approach.

Only a few studies have focused on utilizing pre-trained LLMs for code translation. Notable among them is UniTrans (Yang et al. 2024b), 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 com-

plex 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. (2024) 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.

5.2 Evaluating Translated Code

While we evaluate translated code 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 (Nguyen et al. 2013; Karaivanov et al. 2014; Nguyen et al. 2015; Chen et al. 2018). CodeBLEU (Ren et al. 2020), a recently introduced metric, enhances BLEU by considering the similarity of syntax trees and dataflow graphs. Roziere et al. (2020) 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.

6 Conclusion

We tackle the problem of signature type migration in C-to-Rust translation with LLMs. We fully leverage LLMs' capabilities in migrating types by explicitly asking them to generate candidate signatures. In addition, we mitigate type errors by providing translated callee signatures to LLMs and iteratively fixing errors using compiler feedback. Our evaluation shows the effectiveness of these techniques in increasing migrated types and decreasing type errors.

Acknowledgements This research was supported by National Research Foundation of Korea (NRF) (Grants 2022R1A2C200366011 and 2021R1A5A1021944), Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (2022-0-00460), and Samsung Electronics Co., Ltd (G01210570).

Author Contributions Jaemin Hong: Conceptualization, Data curation, Formal Analysis, Investigation, Methodology, Software, Writing - original draft; Sukyoung Ryu: Funding acquisition, Supervision Writing - review & editing.

Funding Open Access funding enabled and organized by KAIST.

Data Availability The implementation, benchmark programs, and the evaluation scripts are at https://github.com/kaist-plrg/simcrat.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

Ahmed T, Devanbu P (2023) Few-shot training llms for project-specific code-summarization. In: Proceedings of the 37th IEEE/ACM international conference on automated software engineering. Association for Computing Machinery, New York, NY, USA, ASE '22. https://doi.org/10.1145/3551349.3559555

Chen H, Mao Y, Wang X, Zhou D, Zeldovich N, Kaashoek MF (2011) Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In: Proceedings of the second asia-pacific workshop on systems. Association for Computing Machinery, New York, NY, USA, APSys '11. https://doi.org/10.1145/2103799.2103805

Chen M, Tworek J, Jun H, Yuan Q, de Oliveira Pinto HP, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G, Ray A, Puri R, Krueger G, Petrov M, Khlaaf H, Sastry G, Mishkin P, Chan B, Gray S, Ryder N, Pavlov M, Power A, Kaiser L, Bavarian M, Winter C, Tillet P, Such FP, Cummings D, Plappert M, Chantzis F, Barnes E, Herbert-Voss A, Guss WH, Nichol A, Paino A, Tezak N, Tang J, Babuschkin I, Balaji S, Jain S, Saunders W, Hesse C, Carr AN, Leike J, Achiam J, Misra V, Morikawa E, Radford A, Knight M, Brundage M, Murati M, Mayer K, Welinder P, McGrew B, Amodei D, McCandlish S, Sutskever I, Zaremba W (2021a) Evaluating large language models trained on code. arXiv:2107.03374

Chen M, Tworek J, Jun H, Yuan Q, de Oliveira Pinto HP, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G, Ray A, Puri R, Krueger G, Petrov M, Khlaaf H, Sastry G, Mishkin P, Chan B, Gray S, Ryder N, Pavlov M, Power A, Kaiser L, Bavarian M, Winter C, Tillet P, Such FP, Cummings D, Plappert M, Chantzis F, Barnes E, Herbert-Voss A, Guss WH, Nichol A, Paino A, Tezak N, Tang J, Babuschkin I, Balaji S, Jain S, Saunders W, Hesse C, Carr AN, Leike J, Achiam J, Misra V, Morikawa E, Radford A, Knight M, Brundage M, Murati M, Mayer K, Welinder P, McGrew B, Amodei D, McCandlish S, Sutskever I, Zaremba W (2021b) Evaluating large language models trained on code. arXiv:2107.03374

Chen X, Liu C, Song D (2018) Tree-to-tree neural networks for program translation. In: Bengio S, Wallach H, Larochelle H, Grauman K, Cesa-Bianchi N, Garnett R (eds) Advances in neural information processing systems. Curran Associates, Inc., vol 31. https://proceedings.neurips.cc/paper_files/paper/2018/file/d759175de8ea5b1d9a2660e45554894f-Paper.pdf

Danial A (2023) cloc. https://github.com/AlDanial/cloc

De Simone S (2022) Linux 6.1 officially adds support for Rust in the kernel. https://www.infoq.com/news/2022/12/linux-6-1-rust/

Dong Y, Jiang X, Jin Z, Li G (2024) Self-collaboration code generation via chatgpt. arXiv:2304.07590

Emre M, Schroeder R, Dewey K, Hardekopf B (2021) Translating C to safer Rust. Proc ACM Program Lang 5(OOPSLA). https://doi.org/10.1145/3485498

Emre M, Boyland P, Parekh A, Schroeder R, Dewey K, Hardekopf B (2023) Aliasing limits on translating C to safe Rust. Proc ACM Program Lang 7(OOPSLA1). https://doi.org/10.1145/3586046

Fan Z, Gao X, Mirchev M, Roychoudhury A, Tan SH (2023) Automated repair of programs from large language models. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp 1469–1481. https://doi.org/10.1109/ICSE48619.2023.00128

Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M (2020) CodeBERT: A pre-trained model for programming and natural languages. arXiv:2002.08155

GNU (2023) GNU package blurbs. https://www.gnu.org/manual/blurbs.html

Goregaokar M (2017) Fearless concurrency in Firefox Quantum. https://blog.rust-lang.org/2017/11/14/Fearless-Concurrency-In-Firefox-Quantum.html

Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S, Tufano M, Deng SK, Clement C, Drain D, Sundaresan N, Yin J, Jiang D, Zhou M (2021) GraphCodeBERT: Pre-training code representations with data flow. arXiv:2009.08366

Hong J, Ryu S (2023) Concrat: An automatic C-to-Rust lock API translator for concurrent programs. In: Proceedings of the 45th International Conference on Software Engineering, IEEE Press, ICSE '23, p 716-728. https://doi.org/10.1109/ICSE48619.2023.00069,

Hutt T (2021) Would Rust secure cURL? https://blog.timhutt.co.uk/curl-vulnerabilities-rust/
Jung R, Jourdan JH, Krebbers R, Dreyer D (2017) RustBelt: Securing the foundations of the Rust programming language. Proc ACM Program Lang 2(POPL).https://doi.org/10.1145/3158154
Karaivanov S, Raychev V, Vechev M (2014) 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. Association for Computing Machinery, New York, NY, USA, Onward! 2014, pp 173-184. https://doi.org/10.1145/2661136.2661148
Kojima T, Gu SS, Reid M, Matsuo Y, Iwasawa Y (2022) Large language models are zero-shot reasoners. In: Koyejo S, Mohamed S, Agarwal A, Belgrave D, Cho K, Oh A (eds) Advances in neural information processing systems. Curran Associates, Inc., vol 35, pp 22199–22213
Lachaux MA, Roziere B, Szafraniec M, Lample G (2021) DOBF: A deobfuscation pre-training objective for programming languages. In: Ranzato M, Beygelzimer A, Dauphin Y, Liang P, Vaughan JW (eds) Advances in neural information processing systems. Curran Associates, Inc., vol 34, pp 14967–14979. https://proceedings.neurips.cc/paper_files/paper/2021/file/7d6548bdc0082aacc950ed35e91fcccb-Paper.pdf
Li J, Li G, Li Y, Jin Z (2023a) Structured chain-of-thought prompting for code generation. arXiv:2305.06599
Li J, Zhao Y, Li Y, Li G, Jin Z (2023b) Acecoder: Utilizing existing code to enhance code generation. arXiv:2303.17780
Liu F, Li J, Zhang L (2023) Syntax and domain aware model for unsupervised program translation. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp 755–767. https://doi.org/10.1109/ICSE48619.2023.00072
Matsakis ND, Klock FS (2014) The Rust language. In: Proceedings of the 2014 ACM SIGAda annual conference on high integrity language technology. Association for Computing Machinery, New York, NY, USA, HILT '14, p 103-104. https://doi.org/10.1145/2663171.2663188
Mialon G, Dessì R, Lomeli M, Nalmpantis C, Pasunuru R, Raileanu R, Roziere B, Schick T, Dwivedi-Yu J, Celikyilmaz A, Grave E, LeCun Y, Scialom T (2023) Augmented language models: a survey. arXiv:2302.07842
Nguyen AT, Nguyen TT, Nguyen TN (2013) Lexical statistical machine translation for language migration. In: Proceedings of the 2013 9th Joint meeting on foundations of software engineering. Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2013, pp 651-654. https://doi.org/10.1145/2491411.2494584
Nguyen AT, Nguyen HA, Nguyen TT, Nguyen TN (2014) Statistical learning approach for mining API usage mappings for code migration. In: Proceedings of the 29th ACM/IEEE international conference on automated software engineering. Association for Computing Machinery, New York, NY, USA, ASE '14, p 457-468. https://doi.org/10.1145/2642937.2643010
Nguyen AT, Nguyen TT, Nguyen TN (2015) 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. IEEE Press, ASE '15, pp 585-596. https://doi.org/10.1109/ASE.2015.74
OpenAI (2022) Introducing ChatGPT. https://openai.com/blog/chatgpt
OpenAI (2023) OpenAI documentation: Models. https://platform.openai.com/docs/models
OpenAI (2024) GPT-4o mini: advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/
Ouyang L, Wu J, Jiang X, Almeida D, Wainwright C, Mishkin P, Zhang C, Agarwal S, Slama K, Ray A, Schulman J, Hilton J, Kelton F, Miller L, Simens M, Askell A, Welinder P, Christiano PF, Leike J, Lowe R (2022) Training language models to follow instructions with human feedback. In: Koyejo S, Mohamed S, Agarwal A, Belgrave D, Cho K, Oh A (eds) Advances in neural information processing systems. Curran Associates, Inc., vol 35, pp 27730–27744
Pan R, Ibrahimzada AR, Krishna R, Sankar D, Wassi LP, Merler M, Sobolev B, Pavuluri R, Sinha S, Jabbarvand R (2024) 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. Association for Computing Machinery, New York, NY, USA, ICSE '24. https://doi.org/10.1145/3597503.3639226
Ren S, Guo D, Lu S, Zhou L, Liu S, Tang D, Sundaresan N, Zhou M, Blanco A, Ma S (2020) CodeBLEU: a method for automatic evaluation of code synthesis. 2009.10297
Roziere B, Lachaux MA, Chanussot L, Lample G (2020) Unsupervised translation of programming languages. In: Larochelle H, Ranzato M, Hadsell R, Balcan M, Lin H (eds) Advances in neural information processing systems. Curran Associates, Inc., vol 33, pp 20601–20611. https://proceedings.neurips.cc/paper_files/paper/2020/file/ed23fbf18c2cd35f8c7f8de44f85c08d-Paper.pdf
Roziere B, Zhang J, Charton F, Harman M, Synnaeve G, Lample G (2022) 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. https://openreview.net/forum?id=cmt-6KtR4c4

Rust (2022) The Rust programming language. http://rust-lang.org/
Rust (2023a) Crate libc. https://docs.rs/libc
Rust (2023b) The Rust standard library: List of all items. https://doc.rust-lang.org/std/all.html
Rust (2023c) The Rust standard library: Module std::option. https://doc.rust-lang.org/std/option/
Rust (2023d) The Rust standard library: Primitive type never. https://doc.rust-lang.org/std/primitive.never.html
Szafraniec M, Roziere B, Leather H, Charton F, Labatut P, Synnaeve G (2023) Code translation with compiler representations. arXiv:2207.03578
Wang Y, Wang W, Joty S, Hoi SCH (2021) CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv:2109.00859
Wingerter F (2022) C2Rust is back. https://immunant.com/blog/2022/06/back/
Xia CS, Wei Y, Zhang L (2023) Automated program repair in the era of large pre-trained language models. In: 2023 IEEE/ACM 45th international conference on software engineering (ICSE), pp 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129
Yang Z, Keung JW, Sun Z, Zhao Y, Li G, Jin Z, Liu S, Li Y (2024) Improving domain-specific neural code generation with few-shot meta-learning. Inf Softw Technol 166:107365. https://doi.org/10.1016/j.infsof.2023.107365. https://www.sciencedirect.com/science/article/pii/S0950584923002203
Yang Z, Liu F, Yu Z, Keung JW, Li J, Liu S, Hong Y, Ma X, Jin Z, Li G (2024b) Exploring and unleashing the power of large language models in automated code translation. Proc ACM Softw Eng 1(FSE). https://doi.org/10.1145/3660778
Zhang H, David C, Yu Y, Wang M (2023) Ownership guided c to rust translation. In: Enea C, Lal A (eds) Computer Aided Verification. Springer Nature Switzerland, Cham, pp 459–482
Zhong H, Thummalapenta S, Xie T, Zhang L, Wang Q (2010) Mining API mapping for language migration. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering - Volume 1, Association for Computing Machinery, New York, NY, USA, ICSE '10, p 195-204. https://doi.org/10.1145/1806799.1806831

Jaemin Hong is a Ph.D. candidate in Computer Science at KAIST. His research interests include static analysis and program transformation. He received his B.S. in Computer Science and Mathematical Sciences at KAIST.

Sukyoung Ryu is a professor in the School of Computing, KAIST. Her research interests are in programming languages and program analysis. She is a recipient of various awards including the Google Faculty Research Award and several Best Paper Awards in top conferences. She received her Ph.D. in Computer Science at KAIST and worked at Harvard University and Sun Microsystems Laboratories.