Simcrat: A Signature-Modernizing C-to-Rust Translator using a Large Language Model

***Abstract*—Rust, a modern system programming language, overcomes the limitations of C by prioritizing both performance and reliability. It introduces new types that prevent memory bugs while maintaining code readability. Therefore, translating legacy system software from C to Rust is a promising approach to enhance its reliability. However, C-to-Rust translation requires careful consideration from programmers, as C's types compromise safety guarantees when used in Rust code. To maximize the advantages of translation, developers must *modernize function signatures* (parameter and return types), i.e., replace C types in signatures with Rust types. Existing automatic C-to-Rust translators fall short, as they utilize only a limited subset of Rust types. To address this problem, we leverage large language models (LLMs) to discover desirable Rust signatures. LLMs possess knowledge of preferred types and those to be avoided, thanks to their training on human-written idiomatic code. Nevertheless, LLMs often struggle to fully leverage their knowledge in modernizing signatures and frequently generate code with type errors. To tackle these challenges, we propose three techniques: (1) instructing LLMs to generate candidate signatures and selecting the most suitable one; (2) augmenting each C function with translated signatures of its callees; (3) iteratively fixing type errors using compiler feedback. Our evaluation shows that the proposed approach yields a 177% increase in modernized signatures and a 77% decrease in type errors compared to the baseline, the LLM-based translation without our approach, with modest performance overhead and expense.**

I. Introduction

Rust, a relatively new system programming language, tackles the limitations of C by emphasizing both performance and reliability [12], [31]. 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 [13]. Conversely, Rust introduces an ownership type system that enables the type checker to guarantee the absence of memory bugs and data races [26]. By leveraging Rust's modern language features, such as algebraic data types and generics, programmers can write concise and readable code that passes rigorous type checking.

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 subjecting it to the type checker [24]. 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, embarking on the translation of critical system programs to Rust. Mozilla's development of Servo, a web browser written in Rust, serves as a prime example, as modules from Servo have replaced those of Firefox [21]. Furthermore, Linux has introduced support for Rust in kernel development with its recent release [17].

However, C-to-Rust translation demands careful consideration from programmers. It involves more than simply rewriting each line of C code using Rust syntax; instead, it necessitates a deep understanding of the code and the ability to determine appropriate *function signatures*, i.e., parameter and return types. Rust introduces new types that are not in C, and leveraging these types enhances code readability and conciseness. While it is still possible to use types from C code in Rust, they compromise the safety guarantees of the Rust type checker. To maximize the benefits of translation, programmers must carefully *modernize* signatures, i.e., utilize Rust types in signatures while avoiding the use of C types.

While the challenge of finding appropriate signatures during C-to-Rust translation can be mitigated by an automatic C-to-Rust translator, current translators have little ability to modernize signatures. The most widely-used translator, C2Rust [41], focuses on maintaining syntactic fidelity, leading all the signatures to have only C types even after the translation. Recent efforts have aimed to improve C2Rust-generated code, but their achievements remain limited. Emre et al. [18], [19] have proposed techniques that replace C pointer types with reference types, i.e., Rust pointer types; Hong and Ryu [23] have presented a technique that adds guard parameters, targeting the translation of concurrent programs. However, reference and guard types represent only a small portion of Rust types, and the hitherto achieved results do not serve as a general solution to signature modernization.

To address this problem, this work leverages *large language models* (LLMs), e.g., ChatGPT [11], [35], to translate C functions to Rust functions with suitable signatures. LLMs are trained on extensive collections of human-written code, which typically adheres to the idiomatic conventions of programming languages. This gives them an understanding of programming idioms, including knowledge of which types to employ and which ones to avoid. Consequently, LLMs can generate Rust functions with idiomatic signatures. To utilize their capability at the maximum, we explicitly ask them to *generate candidate signatures*, translate the function using each candidate signature, and *select the most suitable translation*, rather than merely requesting them to translate functions.

Unfortunately, incorporating LLMs into C-to-Rust translation presents another challenge due to the limited capability

of current LLMs to generate Rust code that obeys the strict typing rules of the Rust compiler. While LLMs can translate a C function to a Rust function with a desirable signature, the resulting translated code often contains type errors, rendering it uncompilable. Programmers must invest significant manual effort in rectifying type errors to finalize the translation and obtain functional code. Reducing type errors in translated code is crucial as it minimizes the time spent on code corrections.

To mitigate the occurrence of type errors in LLM-translated Rust code, we propose two techniques, *function augmentation with callee signatures* and *compiler feedback-based iterative fix*. To translate a function, we translate its callees first and provide the LLM augmented C code that includes the signatures of the translated callees. This helps the LLM generate Rust code that calls functions with arguments of the correct number and types and uses the return values correctly. Despite this effort, translated code can still have type errors, and we iteratively fix the code to resolve them. When an error message contains a suggested fix, we apply it to the code; otherwise, we provide the error message to the LLM, enabling it to generate fixed code. Repeating this effectively reduces type errors.

Overall, our contributions are as follows:

- We leverage an LLM for signature modernization during C-to-Rust translation by asking it to generate multiple candidate signatures and pick the best (§III-A and §III-D).
- We provide augmented C code with the translated signatures of the callees to the LLM to help it generate correct code (§III-B).
- We iteratively fix LLM-translated code to reduce type errors using the compiler's feedback (§III-C).
- We concretize the proposed approach as a tool, Simcrat (**si**gnature-**m**odernizing **C**-to-**R**ust **a**utomatic **t**ranslator), and evaluate it with real-world C programs, observing a 177% increase in modernized signatures and a 77% decrease in type errors compared to the baseline, the LLM-based translation without our approach, with modest performance overhead and expense (§IV).

We then discuss related work (§V) and conclude (§VI).

II. Signature Modernization Example

This section provides an example of signature modernization in C-to-Rust translation, highlighting its importance. We present the translation of a *semipredicate*, i.e., a function that may fail. Specifically, we consider the following C function:

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

This function calculates the quotient resulting from the division of two integers, `n` and `d`. It is a semipredicate as it fails when the divisor is zero. In C, semipredicates are typically implemented by including an additional parameter of a pointer type. When the function succeeds, it writes the result to the pointer; otherwise, it writes nothing. The return value of the function does not convey the actual result; rather, it serves as an indicator of the success or failure. In this example, a return value of `0` signifies success, while `1` represents failure.

The caller of `div` should examine the return value to determine whether it should read the result or not. Consider the following code snippet demonstrating a call to `div`:

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

It defines an uninitialized variable `q` and passes its pointer to `div`. After `div` returns, the value of `q` can only be read if the return value is zero. Otherwise, the caller must address the failure appropriately.

Using pointer types to implement semipredicates in C negatively impacts code readability and reliability. It is generally more challenging to comprehend a function that produces the result through a parameter than a function that directly returns its result, as parameters are primarily intended for input rather than output. Moreover, a function lacks the ability to compel its caller to inspect its return value. If a caller neglects to check the return value, it may read an uninitialized variable, leading to undefined behavior [25].

When translating `div` to Rust, only replacing the C pointer type with a reference type in the signature is a naïve decision that does not address the drawbacks of the C implementation. The translated code fails to overcome the disadvantages:

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

Although translated to Rust, the code continues to rely on passing the result through a parameter and lack the ability to enforce proper handling of failure by its caller.

Rust provides a solution to this problem by introducing the `Option` type [4], adopted from the functional programming community. An `Option` value encompasses two possiblities: `Some(v)`, indicating a successful result with a single value `v`, and `None`, representing a failure. Consequently, semipredicates can be implemented as functions returning `Option` values. The following code shows the translation of `div` using `Option`:

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

In this translation, the function returns `None` when the divisor is zero; otherwise, it returns `Some` containing the quotient.

This new translation surpasses the previous one in terms of both readability and reliability. By directly returning the result, the function enhances comprehensibility compared to writing the result to a pointer. Additionally, this approach ensures that the caller handles failure, allowing it to access the inner value of `Some` only if the return value matches `Some` through pattern matching, as demonstrated below:

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

This improvement is achievable solely by discovering a signture with the `Option` type through signature modernization, making signature modernization a crucial step towards desirable C-to-Rust translation. Note that `Option` is not the only type that can be introduced through signature modernization; other examples are demonstrated in §IV-D.

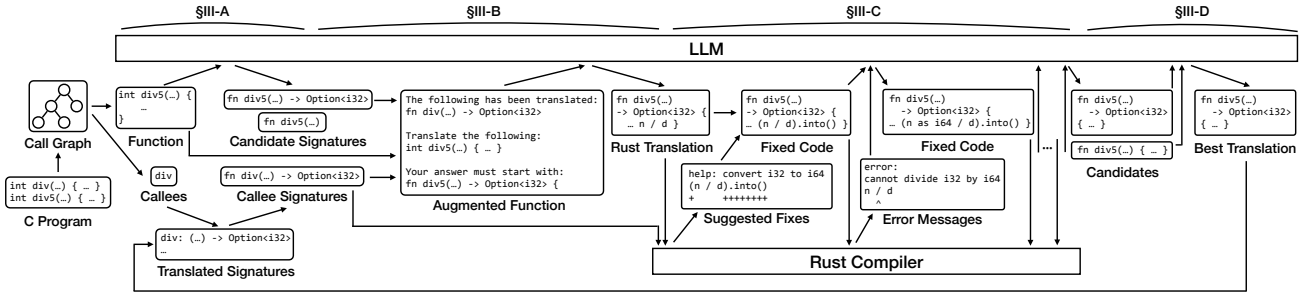

Fig. 1: Overview of signature-modernizing translation via LLM

III. Signature-Modernizing Translation via LLM

This section presents techniques to modernize function signatures and minimize type errors during C-to-Rust translation. The key idea is to leverage LLMs to automatically identify idiomatic Rust signatures for C functions. As illustrated in §II, signature modernization necessitates an understanding of the function's behavior and the corresponding Rust idioms. For instance, to construct a signature using `Option` for `div`, one needs to recognize that the function is a semipredicate and that Rust programmers typically use `Option` for semipredicates. While formal methods can analyze functions' behavior, language idioms predominantly rely on human intuition, which formal methods struggle to automate. On the other hand, LLMs excel at such reasoning as they possess knowledge of language idioms derived from training on a vast corpus of human-written idiomatic code. For this reason, we employ LLMs to automate this reasoning process.

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 [15], [27], [33], [34], [37]. 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 (§III-A). Second, we augment the function with Rust signatures of its callees and translate it to Rust for each candidate signature (§III-B). Third, we type-check the translated code and iterate to resolve type errors using compiler feedback (§III-C). Finally, we select the most suitable candidate (§III-D) 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 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.

A. 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 result in the best output. 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;}
[Completion]
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 semipredicates in real-world Rust code on which the LLM has been trained, it fails to generate code utilizing `Option` for the return type.

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 [32], e.g., incorporating a step-by-step thinking prompt [28]. By instructing the LLM to produce candidate signatures and use them to translate the function, we can obtain more Rust-idiomatic code. 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.
[Completion]
fn div(n: i32, d: i32, q: &mut i32) -> i32;
fn div(n: i32, d: i32) -> Option<i32>;
```

It proposes a signature that incorporates the `Option` type in addition to the one with a pointer-type parameter. We set the number of candidate signatures to two for brevity in this example, while it is configurable.

B. 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 shown below:

```
[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> {
[Completion]
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` using the LLM without providing the translated signature of `div`, the resulting Rust code is incorrect, as demonstrated 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> {
```

```
[Completion]
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 translated code 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 modernized signature of `div`, which takes two arguments and returns `Option`. As a result, the translated code encounters type errors and fails to compile. This outcome is expected because 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. 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> {
```

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

Considering the provided signature of `div`, the LLM generates the correct code that passes two arguments to `div` and directly returns 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.

C. Compiler Feedback-Based Iterative Fix

After translating each function, we type-check the function along with its callees. During type checking, each callee's code comprises the function signature and a body consisting solely of the `todo!` macro invocation, 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 the function augmentation 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 fixing them. Specifically, we select code snippets that exhibit missing type casts as examples due to their simplicity.

Consider the following `div` function that produces a `long`, rather than an `int`:

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

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

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

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

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

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

Unfortunately, not all error messages provide a suggested 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 the following error message:

```
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: 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
```

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
```

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 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.
[Completion]
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).

D. Best Candidate Selection

In the final step, we choose the most desirable translation from the available candidates. Note that each candidate is the function's translation using each candidate signature.

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 candidates, 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 candidates exist, collecting the code of all candidates may exceed the token limit of the prompt. To address this issue, we only compare two candidates 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?
[Completion]
Implementation 2
```

The LLM is expected to choose a translation that follows Rust idioms. For instance, if a function is a semipredicate, a translation using the `Option` type would be favored. Conversely, if a function always succeeds, a translation with the `Option` type would be less preferable since it is not idiomatic for a function to always return `Some` and never return `None`.

IV. Evaluation

In this section, we first provide an overview of our implementation (§IV-A) and the process of collecting the test set (§IV-B). We then evaluate the effectiveness of the proposed approach with the following four research questions:

- RQ1. Promotion of signature modernization: Does the proposed approach effectively promote signature modernization by generating candidate signatures? (§IV-C)
- RQ2. Quality of modernized signatures: Do the signatures modernized by the proposed approach adhere to Rust idioms? (§IV-D)

- RQ3. Type error reduction: Does the proposed approach effectively reduce type errors by augmenting functions and iteratively fixing errors? (§IV-E)
- RQ4. Overhead and expense: Does the proposed approach entail reasonable overhead and expense? (§IV-F)

Finally, we dicuss threats to validity (§IV-G).

A. Implementation

We realized the proposed approach as a tool, Simcrat. Simcrat 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 the language model component of Simcrat, we employ ChatGPT, specifically utilizing the gpt-3.5-turbo-0301 model with a temperature setting of 0. By setting the temperature to 0, we ensure that the behavior of the language model is predominantly deterministic, although some nondeterministic behavior may still occur [1]. Simcrat interacts with ChatGPT through the API provided by OpenAI.

Unfortunately, the token limit of the ChatGPT API poses a restriction on Simcrat's ability to translate lengthy functions. Specifically, the employed model has a token limit of 4,096, which encompasses both prompt and completion. To ensure successful translation, Simcrat refrains from translating functions that exceed 1,500 tokens. This threshold is less than half of the token limit because a translated Rust function generally requires more tokens than its original C equivalent, and there must be space for the callees' signatures as well.

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 for simultaneous translation. Furthermore, the translation of a function using different candidate signatures is also independent of each other, enabling parallel translation.

While our primary focus lies in the translation of functions, Simcrat is also capable of translating type definitions and global variables, as these are commonly found in real-world C programs. The translation strategy for types and variables is similar to that of functions. We ensure that each entity is translated only after all the entities it refers to have been translated, thereby supplying the necessary information about the referred entities to the LLM. For instance, when a variable refers to a user-defined type, the variable declaration is augmented with the translated definition of the type.

To perform experiments, we selectively enable or disable each proposed technique. We denote the disabled features using superscripts. When we refer to Simcrat, it is the version with all features enabled. The superscript -c denotes the absence of candidate signature generation. Similarly, -a means the lack of function augmentation with callee signatures, and -f indicates the absence of error fixes. For instance, Simcrat^{-c} does not generate candidate signatures but still provides callee signatures to the LLM and attempts to fix errors.

TABLE I: The size of each program in the test set

Program	LOC	Types	Variables	Functions	Omitted
`ttygif`	508	7	1	32	0
`tini`	524	1	11	15	1
`microsocks`	546	8	7	31	0
`C-Thread-Pool`	553	11	2	23	0
`greatest`	576	21	7	79	1
`genann`	608	3	6	25	0
`tiny-AES-c`	618	2	3	30	1
`mon`	687	5	2	39	0
`db_tutorial`	695	11	38	53	0
`minilisp`	722	3	14	62	0
`endlessh`	729	5	6	29	0
`entr`	739	2	16	18	0
`cmatrix`	748	2	8	10	1
`tinyvm`	755	8	3	46	0
`mptun`	774	4	1	22	0
`vmtouch`	808	1	35	32	0
`gc`	850	6	4	56	0
`sds`	875	6	3	51	1
`no-more-secrets`	893	1	8	33	0
`linenoise`	944	8	13	53	0
`kilo`	986	8	5	36	0
`kcp`	1012	16	24	45	1
`ChinaDNS`	1021	31	28	28	0
`picohttpparser`	1033	3	6	30	2
`dyad`	1094	8	8	54	0
`graftcp`	1118	12	15	56	2
`GloVe`	1193	9	41	34	2
`json-parser`	1196	8	17	10	1
`earlyoom`	1216	5	2	36	1
`shc`	1236	0	38	18	2
`cpulimit`	1236	6	7	48	0

B. Test Set Collection

We collected 31 real-world C programs, all of the public GitHub repositories that met the following criteria: (1) having over 1,000 stars, (2) containing C code ranging from 500 to 1,250 lines, as measured by `cloc` [16], (3) having ten or more functions, (4) not relying on external libraries, and (5) compatible with Linux for building and execution.

Table I presents the collected programs and their respective code sizes. The second column displays the number of lines of C code; the third to fifth columns indicate the numbers of type definitions, global variable declarations, and function definitions, respectively; the last column shows the numbers of functions omitted from translation due to exceeding 1,500 tokens. While being small, the programs have multiple functions, ranging from 10 to 79, whose idiomatic translation with suitable signatures constitutes the main focus of this work.

C. RQ1: Promotion of Signature Modernization

To evaluate the effectiveness of the proposed approach in promoting signature modernization, we compare Simcrat to Simcrat^{-c}, which serves as the baseline by not generating candidate signatures. For Simcrat, we set the number of candidate signatures for each function to three. We compare the number of signatures modernized by each setting. A signature is considered modernized when the Rust signature differs from the original C signature, with the following exceptions:

- Integer types, `char`, and `bool` are not distinguished. For instance, translating `char` to `u8` is not considered a signature modernization. This is because both C and

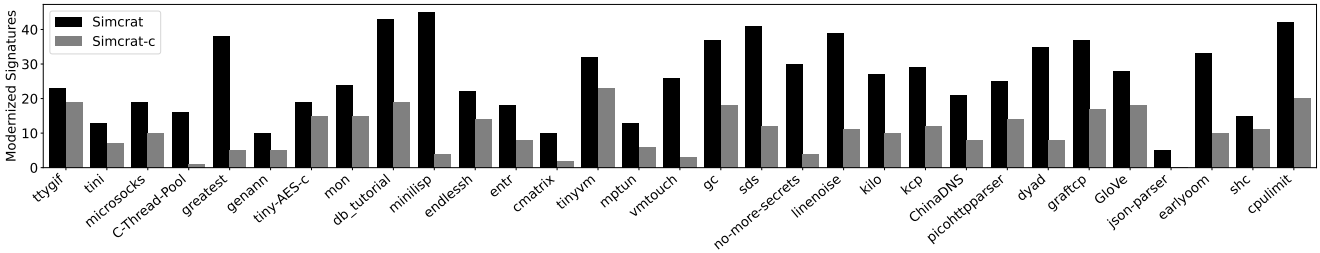

Fig. 2: The count of modernized signatures in each program using each setting

Rust include these types, and such translations do not benefit from new types introduced in Rust.
- Floating point types are not distinguished.
- C pointer types and reference types are not distinguished. Emre et al. [18], [19] have already proposed techniques to replace C pointers with references, so this translation does not reveal a novel aspect of our work.
- C pointer types and reference types wrapped in `Option` are not distinguished. Since C pointers can be `NULL` while references cannot, it is natural to translate nullable C pointers to references wrapped in `Option`. This idea has also been employed by Emre et al.

Figure 2 illustrates the count of modernized signatures in each program by employing Simcrat and Simcrat^{-c}.

The results show that the candidate signature generation process significantly enhances the number of modernized signatures. Across all programs, Simcrat outperforms Simcrat^{-c}. With the exception of `json-parser`, where Simcrat^{-c} does not modernize any signatures, Simcrat modernizes 2.77× more signatures on average compared to Simcrat^{-c}. The term average refers to the geometric mean in our evaluation.

D. RQ2: Quality of Modernized Signatures

To evaluate the quality of the signatures modernized by the proposed approach, we categorize them and conduct case studies on representative examples from each category, focusing on their adherence to Rust idioms. Through manual investigation, we classify the signatures into ten categories as follows:

- Option: Introduces `Option` or `Result` [5], where `Result` can carry information about the failure. A `Result` value is either `Ok`, containing a value, or `Err`, containing information about the failure.
- Tuple: Introduces a tuple type.
- Vec: Introduces `Vec` [7] or a slice type [9] instead of a pointer type to represent a contiguous sequence. `Vec` is a heap-allocated sequence, and a slice is a pointer to a sequence accompanied by the length.
- String: Introduces `String` [6] or `str` [10] instead of `char *`. `String` is a heap-allocated string, and `str` is a string slice.
- File: Introduces a Rust type related to the file system operation, replacing its C counterpart.
- Never: Introduces `!` (called never) [8], representing that a function never returns when used as the return type.
- Generics: Introduces a type parameter to make a function generic.

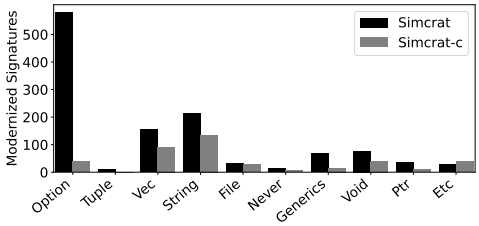

Fig. 3: The count of modernized signatures per category

- Void: Replaces `void *` with another type.
- Ptr: Replaces a pointer type with its pointee type, or vice versa.
- Etc: Represents cases that belong to none of the above.

Note that a single modernized signature can belong to multiple categories, such as introducing both `Option` and `Vec`.

Figure 3 shows the count of modernized signatures per category. Simcrat outperforms Simcrat^{-c} in all categories, except for Etc. Notably, Simcrat effectively incorporates `Option` or `Result` for semipredicates while Simcrat^{-c} modernizes a much fewer number of semipredicates.

Our case studies show that the signatures modernized by Simcrat adhere to Rust idioms. We present examples from these case studies to illustrate instances where Simcrat^{-c} fails to identify idiomatic signatures, as well as the advantageous signatures discovered by Simcrat.

1) Option: We demonstrate the utilization of `Result` in signature modernization, similar to `Option` discussed in §II, through the `ttyread` function in `ttygif`:

```
int ttyread(char **buf) { if (...) { return 0; }
    *buf = malloc(len); ... return 1; }
```

When a particular condition is satisfied, the function allocates a buffer, writes contents to it, and outputs it via a parameter. Simcrat^{-c} replaces one C pointer type with a reference type but does not introduce `Option` nor `Result`:

```
fn ttyread(buf: &mut *mut u8) -> i32;
```

However, Simcrat discovers a signature that employs `Result`:

```
fn ttyread() -> Result<Box<[u8]>, ()>;
```

It returns `Ok` containing the buffer when it succeeds; otherwise, it returns `Err`.

Another example that involves `Result` is the `isolate_child` function in `tini`:

```
int isolate_child() {if (...) {return 1;} return 0;}
```

The function performs a specific operation and signals its success or failure through the return value. Simcrat^{-c} maintains the signature:

```
fn isolate_child() -> i32;
```

However, Rust programmers prefer returning a `Result`, whose intention is clearer than an integer. Simcrat successfully discovers the desired signature:

```
fn isolate_child() -> Result<(), ()>;
```

It returns `Ok` to indicate success and `Err` to indicate failure.

2) Tuple: An example from this category is the `get_cursor_position` function in `kilo`:

```
int get_cursor_position(int *rows, int *cols) {
    if (...) { return -1; } ...
    *rows = ...; *cols = ...; return 0; }
```

It checks a certain condition and produces the cursor position, represented by two integers, through two parameters. Simcrat^{-c} only introduces reference types:

```
fn get_cursor_position(
    rows: &mut i32, cols: &mut i32) -> i32;
```

However, Rust programmers favor using tuples when returning multiple values. Simcrat can find a signature using a tuple type:

```
fn get_cursor_position() -> Option<(i32, i32)>;
```

3) Vec: An example involving `Vec` is the `resolve` function in `microsocks`:

```
int resolve(struct addrinfo **addr) { ... }
```

It provides a list of `addrinfo` if the host and the port are valid. Simcrat^{-c} does not use `Vec` to represent the collection:

```
fn resolve(addr: &mut *mut addrinfo);
```

In contrast, Simcrat employs `Vec` in the signature:

```
fn resolve() -> Result<Vec<SocketAddr>, ()>;
```

Note that it also replaces the C type `addrinfo` with the Rust type `SocketAddr`.

Another example in this category involves a slice, as seen in the `check_auth_method` function in `microsocks`:

```
void check_auth_method(char *buf, size_t n) { ... }
```

The function takes a buffer and its length because a C pointer type does not carry the length. Simcrat^{-c} transforms a pointer to a slice, but it fails to exploit that a slice already contains the length, maintaining the length parameter:

```
fn check_auth_method(buf: &[u8], n: usize);
```

However, Simcrat can remove the unnecessary length parameter from the signature:

```
fn check_auth_method(buf: &[u8]);
```

4) String: We omit this category as it resembles Vec.

5) File: An example from this category is the `tun_alloc` function in `mptun`:

```
int tun_alloc() {int fd = open(...); ... return fd;}
```

The function opens a file and returns the file descriptor. Simcrat^{-c} generates the following signature, which still returns a file descriptor:

```
fn tun_alloc() -> Result<i32, ()>;
```

However, a file descriptor does not guarantee that the file is open, and Rust programmers prefer using the `File` type to represent an open file. Simcrat is capable of introducing `File`:

```
fn tun_alloc() -> Result<File, ()>;
```

6) Never: An example involving the `!` type is the `die` function in `endlessh`:

```
void die() { ... exit(...); }
```

It never returns because it calls `exit`. However, Simcrat^{-c} does not employ `!` in the signature:

```
fn die();
```

Conversely, Simcrat introduce `!`:

```
fn die() -> !;
```

7) Generics: An example from this category is the `tvm_htab_add_ref` function in `tinyvm`:

```
void tvm_htab_add_ref(void *valptr, int len) { ... }
```

The function takes a pointer to a value of any type and the size of the value. In C, the use of a `void` pointer for this purpose is essential due to the lack of generics. However, Rust provides generics, allowing the size of the type parameter to be determined at compile time through monomorphization [3]. Unfortunately, Simcrat^{-c} does not leverage generics:

```
fn tvm_htab_add_ref(
    valptr: *const c_void, len: usize);
```

On the other hand, Simcrat can introduce a type parameter and eliminate the size parameter.

```
fn tvm_htab_add_ref<T>(valptr: &T);
```

8) Void: Some `void` pointers should be replaced with pointers of a specific type, rather than a type parameter, as seen in the `cmp_net_mask` function in `ChinaDNS`:

```
int cmp_net_mask(const void *a, const void *b) {
  NetMaskT *neta = (NetMaskT *)a;
  NetMaskT *netb = (NetMaskT *)b; ... }
```

The function unnecessarily takes `void` pointers; it immediately casts them to `NetMaskT *`. However, Simcrat^{-c} maintains the `void` pointers:

```
fn cmp_net_mask(
    a: *const c_void, b: *const c_void) -> c_int;
```

Simcrat can replace the `void` pointers with `&NetMaskT`:

```
fn cmp_net_mask(
    a: &NetMaskT, b: &NetMaskT) -> Ordering;
```

Note that Simcrat also replaces an integer with `Ordering`, which represents the result of a comparison between two values, clarifying the intention of the function.

9) Ptr: An example from this category is the `make_env` function in `minilisp`:

```
void make_env(Obj **vars, Obj **up) { ...
    r->vars = *vars; r->up = *up; ... }
```

While the function receives two pointers to `Obj *` values, it merely reads the pointer. It would be more preferable to directly accept `Obj *` values. However, Simcrat^{-c} lacks the capability to make such modifications:

```
fn make_env(vars: &mut *mut Obj, up: &mut *mut Obj);
```

On the other hand, Simcrat can identify the desired signature:

```
fn make_env(vars: &mut Obj, up: &mut Obj);
```

An example of the opposite direction is the `system_exec` function in `ttygif`:

```
void system_exec(Options o) { ... }
```

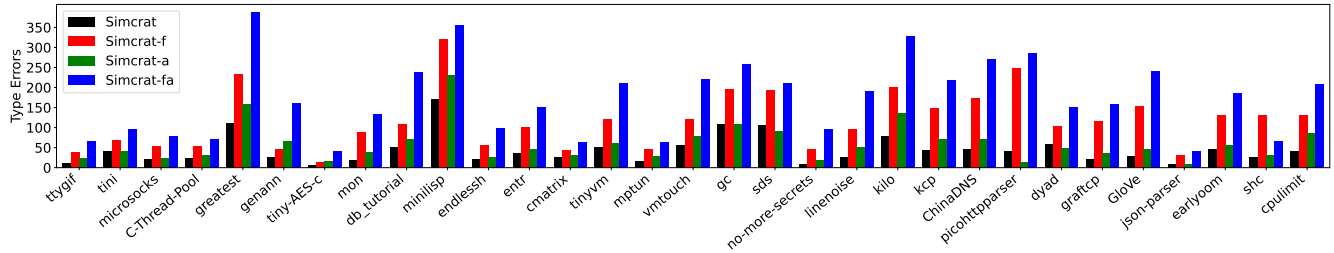

Fig. 4: The count of type errors in translated code using each setting

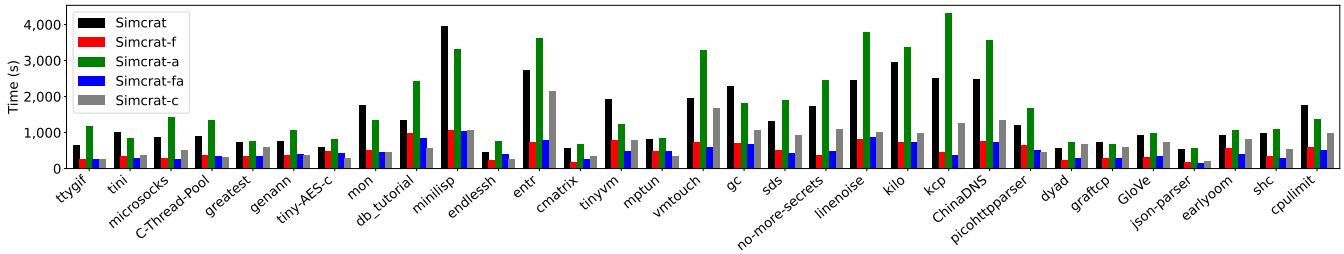

Fig. 5: The translation time of each setting

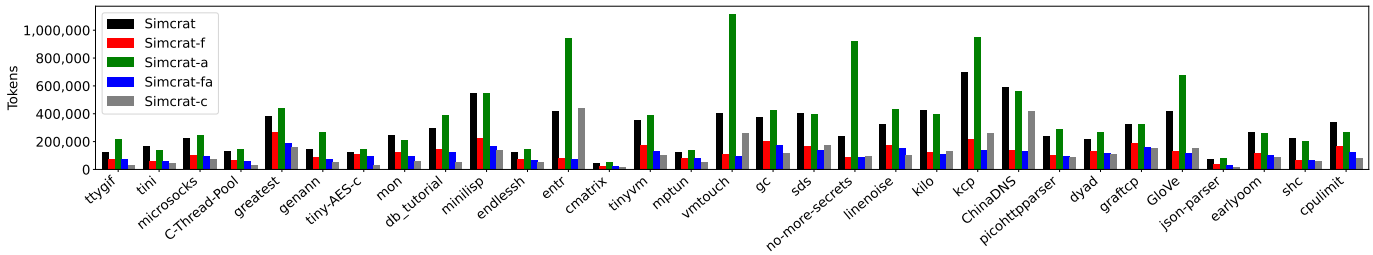

Fig. 6: The count of the tokens used by each setting

The function takes an `Options` object to retrieve configuration information for command execution. Given that the function solely reads the object, and considering its substantial size, passing a pointer instead of the object would be more efficient. However, Simcrat^{-c} does not change the signature:

```
fn system_exec(o: Options);
```

Simcrat can replace the object with a pointer:

```
fn system_exec(o: &Options);
```

E. RQ3: Type Error Reduction

To evaluate the efficacy of the proposed approach in reducing type errors, we compare four settings: Simcrat, Simcrat^{-f}, Simcrat^{-a}, and Simcrat^{-fa}. We measure the number of type errors in the code translated by each setting, calculating the sum of type errors across all functions. Figure 4 shows the type error count for each program, corresponding to each setting.

The results demonstrate that the proposed approach effectively reduce type errors. First, we compare Simcrat to Simcrat^{-fa}, which serves as the baseline by not employing any error-reducing techniques. Simcrat outperforms the baseline across all programs, resulting in a 77% reduction in type errors on average. Compiler feedback-based iterative fix and function augmentation with signature callees collectively contribute to this improvement.

Next, we assess the effectiveness of each technique individually. We evaluate the impact of iterative fix by comparing Simcrat to Simcrat^{-f} and Simcrat^{-a} to Simcrat^{-fa}. Simcrat shows a 65% average reduction in type errors compared to Simcrat^{-f}, and Simcrat^{-a} exhibits a 68% average reduction compared to Simcrat^{-fa}. These results indicate that the proposed technique effectively resolves type errors by iteratively fixing them using compiler feedback, achieving a substantial reduction of about two-thirds, irrespective of the use of function augmentation.

Finally, we evaluate the effectiveness of function augmentation by comparing Simcrat to Simcrat^{-a} and Simcrat^{-f} to Simcrat^{-fa}. On average, Simcrat achieves a 28% reduction in type errors compared to Simcrat^{-a}, and Simcrat^{-f} achieves a 35% reduction compared to Simcrat^{-fa}. These results indicate that function augmentation effectively reduce type errors by providing additional information to the LLM. Note that Simcrat^{-f} performs worse than Simcrat^{-fa} for one program (`shc`). This implies that function augmentation does not guarantee a reduction in type errors, unlike iterative fix, due to the probabilistic nature of the LLM. Additionally, Simcrat performs worse than Simcrat^{-a} for four programs (`sds`, `picohttpparser`, `dyad`, and `json-parser`), suggesting that having fewer type errors before the fix does not necessarily result in fewer type errors after the fix.

F. RQ4: Overhead and Expense

To assess the performance overhead and the expense of the proposed approach, we examine five settings: Simcrat, Simcrat^{-f}, Simcrat^{-a}, Simcrat^{-fa}, and Simcrat^{-c}. In each setting, we measure the translation time and the number of tokens used for communication with the LLM, summing prompts and

completions. The ChatGPT API follows token-based payment, with a cost of $0.002 per 1,000 tokens [2]. Figures 5 and 6 respectively present the translation time and token usage for each program, corresponding to each setting.

The results show that the proposed approach incurs a reasonable additional time compared to the baseline. Both candidate signature generation and iterative fix increase the translation time. On average, Simcrat takes 1.92× longer than Simcrat^{-c}. This overhead arises from that Simcrat translates each function three times, using each candidate signature, while Simcrat^{-c} translates each function only once. Although the candidates are translated in parallel, the maximum translation time among the three is likely to exceed the time required for a single translation performed by Simcrat^{-c}. In addition, on average, Simcrat takes 2.64× longer than Simcrat^{-f} and 2.75× longer than Simcrat^{-fa}. This is primarily attributed to the iterative fix, which inherently takes a signficant time due to a series of sequential LLM invocations.

On the other hand, when used in conjunction with iterative fix, function augmentation diminishes the translation time. On average, Simcrat takes 17% less time than Simcrat^{-a}. This is because function augmentation mitigates type errors in the initial translation, often reducing the iterations to fix them.

The results also indicate that the proposed approach introduces a reasonable additional expense compared to the baseline. The trend is consistent with that of the translation time. On overage, Simcrat uses 2.23× more tokens than Simcrat^{-f}, 2.57× more tokens than Simcrat^{-fa}, and 2.87× more tokens than Simcrat^{-c}. Note that the increase in token usage compared to Simcrat^{-c} is significantly higher than the increase in time, as token usage does not benefit from parallelism. On the other hand, Simcrat uses 19% fewer tokens than Simcrat^{-a}.

G. Threats to Validity

The threats to external validity pertain to the selection of C programs, each having over 1,000 stars and C code ranging from 500 to 1,250 lines. Translating less popular or larger programs may reveal different facets. Conducting additional experiments involving more C projects will provide greater confidence in the generalizability of our approach.

The threats to construct validity include evaluation metrics. We measured the number of type errors to evaluate the translation, but fewer type errors do not necessarily indicate a superior translation. In some cases, code with fewer type errors may require more code changes to succeed compilation. Furthermore, even when the translated code has no type errors, it may not preserve the original semantics. To enhance the reliability of our approach, future work could complement the current metric, e.g., by running unit tests.

V. Related Work

A. Translating C to Rust

Several studies have explored automatic C-to-Rust translation, but their ability to modernize signatures is limited. As mentioned in §I, C2Rust [41] preserves all the types from the original C code during translation, and the approaches of Emre et al. [18], [19] and Hong and Ryu [23] enhance C2Rust-generated code by substituting specific C types with their Rust counterparts, partially modernizing signatures. Ling et al. [30] introduced CRustS, which replaces C primitive types with their Rust equivalents, e.g., replacing `c_int` with `i32`, in C2Rust-generated code. These studies rely on syntactic transformation rules and static analyses, instead of utilizing LLMs, lacking a general solution to signature modernization.

B. 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# [15], [27], [33], [34]. To address this limitation, TransCoder [37] introduces unsupervised learning to programming language translation, enabling training with monolingual code bases. Further studies [29], [38], [39] enhance TransCoder's translation capabilities by utilizing obfuscated code, unit tests, and compilers' intermediate code representation during training. TransCoder and its successors primarily focus on translating small programs containing one or two functions, specifically solutions to coding problems found on online platforms. Consequently, the need for signature modernization and function augmentation with callee signatures has not been motivated. There exist several LLMs capable of code translation [14], [20], [22], [40], which can be effectively leveraged by our approach.

C. 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 [15], [27], [33], [34]. CodeBLEU [36], a recently introduced metric, enhances BLEU by considering the similarity of syntax trees and dataflow graphs. Roziere et al. [37] proposed computational accuracy, which verifies if the translated code preserves the original semantics through the execution of unit tests. This work does not empoly existing metrics due to the absence of human-translated Rust code for the collected C programs and the presence of type errors preventing compilation and unit test execution.

VI. Conclusion

We tackle the problem of signature modernization in C-to-Rust translation by utilizing LLMs. We fully leverage LLMs' capabilities in modernizing signatures 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 modernized signatures and decreasing type errors.

References

[1] OpenAI documentation: Models. https://platform.openai.com/docs/models.

[2] OpenAI: Pricing. https://openai.com/pricing.

[3] Rust compiler development guide: Monomorphization. https://rustc-dev-guide.rust-lang.org/backend/monomorph.html.

[4] The Rust standard library: Module std::option. https://doc.rust-lang.org/std/option/.

[5] The Rust standard library: Module std::result. https://doc.rust-lang.org/std/result/.

[6] The Rust standard library: Module std::string. https://doc.rust-lang.org/std/string/.

[7] The Rust standard library: Module std::vec. https://doc.rust-lang.org/std/vec/.

[8] The Rust standard library: Primitive type never. https://doc.rust-lang.org/std/primitive.never.html.

[9] The Rust standard library: Primitive type slice. https://doc.rust-lang.org/std/primitive.slice.html.

[10] The Rust standard library: Primitive type str. https://doc.rust-lang.org/std/primitive.str.html.

[11] Introducing ChatGPT. https://openai.com/blog/chatgpt, 2022.

[12] The Rust programming language. http://rust-lang.org/, 2022.

[13] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, New York, NY, USA, 2011. Association for Computing Machinery.

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. https://doi.org/10.48550/arXiv.2107.03374, 2021.

[15] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[16] Al Danial. cloc. https://github.com/AlDanial/cloc.

[17] Sergio De Simone. Linux 6.1 officially adds support for Rust in the kernel. https://www.infoq.com/news/2022/12/linux-6-1-rust/, 2022.

[18] Mehmet Emre, Peter Boyland, Aesha Parekh, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Aliasing limits on translating C to safe Rust. *Proc. ACM Program. Lang.*, 7(OOPSLA1), apr 2023.

[19] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Translating C to safer Rust. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. https://doi.org/10.48550/arXiv.2002.08155, 2020.

[21] Manish Goregaokar. Fearless concurrency in Firefox Quantum. https://blog.rust-lang.org/2017/11/14/Fearless-Concurrency-In-Firefox-Quantum.html, 2017.

[22] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training code representations with data flow. https://doi.org/10.48550/arXiv.2009.08366, 2021.

[23] Jaemin Hong and Sukyoung Ryu. Concrat: An automatic C-to-Rust lock API translator for concurrent programs. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Tim Hutt. Would Rust secure cURL? https://blog.timhutt.co.uk/curl-vulnerabilities-rust/, 2021.

[25] ISO/IEC 9899:2018. Information technology — programming languages — C, 2018.

[26] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[27] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, page 173–184, New York, NY, USA, 2014. Association for Computing Machinery.

[28] Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 22199–22213. Curran Associates, Inc., 2022.

[29] Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. DOBF: A deobfuscation pre-training objective for programming languages. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 14967–14979. Curran Associates, Inc., 2021.

[30] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R. Cordy, and Ahmed E. Hassan. In Rust we trust – a transpiler from unsafe C to safer Rust. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 354–355, 2022.

[31] Nicholas D. Matsakis and Felix S. Klock. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery.

[32] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Roziere, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. Augmented language models: a survey. https://doi.org/10.48550/arXiv.2302.07842, 2023.

[33] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 651–654, New York, NY, USA, 2013. Association for Computing Machinery.

[34] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Divide-and-conquer approach for multi-phase statistical migration for source code. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, page 585–596. IEEE Press, 2015.

[35] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744. Curran Associates, Inc., 2022.

[36] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: a method for automatic evaluation of code synthesis. https://doi.org/10.48550/arXiv.2009.10297, 2020.

[37] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 20601–20611. Curran Associates, Inc., 2020.

[38] Baptiste Roziere, Jie Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[39] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. Code translation with compiler representations. https://doi.org/10.48550/arXiv.2207.03578, 2023.

[40] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. https://doi.org/10.48550/arXiv.2109.00859, 2021.

[41] Frances Wingerter. C2Rust is back. https://immunant.com/blog/2022/06/back/, 2022.