Scalable, Validated Code Translation of Entire Projects using Large Language Models

HANLIANG ZHANG, University of Bristol, United Kingdom
CRISTINA DAVID, University of Bristol, United Kingdom
MENG WANG, University of Bristol, United Kingdom
BRANDON PAULSEN, Amazon, USA
DANIEL KROENING, Amazon, USA

Large language models (LLMs) show promise in code translation due to their ability to generate idiomatic code. However, a significant limitation when using LLMs for code translation is scalability: existing works have shown a drop in translation success rates for code exceeding around 100 lines. We overcome this limitation by developing a modular approach to translation, where we partition the code into small code fragments which can be translated independently and semantically validated (that is, by checking I/O equivalence). When this approach is applied naively, we discover that LLMs are unreliable when translating features of the source language that do not have a direct mapping to the target language, and that the LLM often gets stuck in repair loops when attempting to fix errors. To address these issues, we introduce two key concepts: (1) *feature mapping*, which integrates predefined translation rules with LLM-based translation to guide the LLM in navigating subtle language differences and producing semantically accurate code; and (2) *type-compatibility*, which facilitates localized checks at the function signature level to detect errors early, thereby narrowing the scope of potential repairs. We apply our approach to translating real-world Go codebases to Rust, demonstrating that we can consistently generate reliable Rust translations for projects up to 9,700 lines of code and 780 functions, with an average of 73% of functions successfully validated for I/O equivalence, considerably higher than any existing work.

CCS Concepts: • **Software and its engineering** → **Empirical software validation**; **Source code generation**; • **Computing methodologies** → **Machine learning**.

Additional Key Words and Phrases: Program Translation, Program Analysis, LLMs

ACM Reference Format:
Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. 2025. Scalable, Validated Code Translation of Entire Projects using Large Language Models. *Proc. ACM Program. Lang.* 9, PLDI, Article 212 (June 2025), 26 pages. https://doi.org/10.1145/3729315

1 Introduction

Code translation has many important practical applications. For example, a developer may wish to modernize their application [30, 34], they may wish to maintain a client SDK in multiple languages [4, 5, 18], or they may wish to obtain the benefits of another language [2, 6]. The Rust language has received special attention as a target language for code translation because it achieves a strong balance between safety and performance [6]. However, manual code translation is tedious and error prone, hence automated code translation would save developer time and energy.

Authors' Contact Information: Hanliang Zhang, University of Bristol, Bristol, United Kingdom, hanliang.zhang@bristol.ac.uk; Cristina David, University of Bristol, Bristol, United Kingdom, cristina.david@bristol.ac.uk; Meng Wang, University of Bristol, Bristol, United Kingdom, meng.wang@bristol.ac.uk; Brandon Paulsen, Amazon, Arlington, USA, bpaulse@amazon.com; Daniel Kroening, Amazon, Seattle, USA, dkr@amazon.com.

ACM 2475-1421/2025/6-ART212
https://doi.org/10.1145/3729315

Prior work in automated translation to Rust falls into two categories: rule-based (or symbolic) [2, 62] and machine-learning (ML)-based (typically using LLMs) [51, 52, 54, 55, 61]. Rule-based approaches have the advantage of being semantically correct (i.e. I/O equivalent) by construction, but often produce non-idiomatic and unmaintainable code [25, 44, 60]. Conversely, ML-based approaches are usually more idiomatic and maintainable, but come without correctness guarantees.

Recent efforts have combined LLM-based translation with a validation step such as running unit tests, differential fuzzing [25], or formal verification [60] to ensure semantic correctness. If validation fails, a new translation is generated. However, these approaches have primarily focused on translating competitive programming-style programs or small code snippets from real-world projects and struggle with translating entire projects. This limitation arises because the context windows of many state-of-the-art LLMs are not large enough to encompass entire projects. Even when the context window suffices, the probability of errors increases exponentially with the amount of code being translated [25].

In this work, we aim to move beyond code snippet translation by designing an LLM-based translation approach capable of handling *entire projects* and producing *validated translations*. A natural approach to scale to entire projects, taken by us and work parallel to ours [29, 53], is to partition the project into smaller code fragments (e.g. functions, type definitions) that can be translated individually. We can compute a dependency graph between the fragments, and translate them in post-order, which allows us to incrementally translate and validate the semantic correctness of each translated fragment before translating the next fragment. Moreover, we can provide the LLM with context about translations of previous fragments to aid in translating the current fragment.

However, this baseline approach alone still struggles to produce semantically correct translations. While the parallel works have achieved high compilation success rates [29, 53], they report that a large proportion of the original project's test suite, when translated and executed on the translated project, results in runtime errors, by which we mean errors that cause the unit tests to crash, rather than assertion failures due to incorrect results. This highlights the significant gap between generating compilable code and producing code that actually preserves the original semantics.

We observe that any mistake made by the LLM when translating a code fragment can have cascading effects on subsequent translations. First, the LLM often prioritizes generating translations that compile alongside the erroneous fragment, rather than ensuring I/O equivalence with the original code. Second, even when a semantic check identifies the mistake, it may not do so immediately—the error might only be triggered when the code fragment interacts with other parts of the project. Necessary repairs often introduce new errors, affecting other parts of the project, and the LLM can become trapped in a repair loop. This can delay or halt translation progress altogether.

Our solution is to provide specific guidance to the LLM in order to minimize the number of errors the LLM makes from the outset. Next, we outline the strategies we use to achieve this.

Predefined Mapping Rules Between Source and Target Language Features. We find that LLMs often make mistakes when (potentially subtle) differences exist between a feature in the source language and the immediately apparent equivalent in the target language. For example, the Go and Rust languages have different idioms for error handling; Go interfaces are structural, while Rust traits are nominal; and global variables in Go allow dynamic initialization, whereas Rust does not. Moreover, there may be multiple approaches to translate these features from the source language to the target language, and the user of the code translator may prefer a specific approach. Without guidance on translating these language features, the LLM frequently makes mistakes, and applies inconsistent approaches when translating them.

To precent these common LLM mistakes and encode user preferences, we propose a technique called *feature mapping*, which applies predefined translation rules that map source language features to their counterparts in the target language. As we still want to take advantage of the LLM's ability to generate idiomatic code, we introduce a combined approach that integrates rule-based and LLM-based translation. This method instructs the LLM on which rule to apply and performs static checks on the resulting translation to ensure compliance with these rules.

Type-Compatibility-Driven Translation. Inspired by language interoperability [38], we define a set of criteria for *type-compatibility*. When these criteria are met, they ensure that concrete values of a type in the source language can be converted to concrete values of the translated type in the target language. Type-compatibility is significant because it is required for I/O equivalence between the source and target code. Moreover, a type-compatibility check is simpler than an I/O equivalence check. It is localized to the signature of the newly translated function, does not require access to the rest of the code, and can be performed immediately after a function's translation. This check can quickly identify mistakes in the function's signature, such as errors in the number or type of arguments. Undetected, these mistakes will cause the unit tests to fail with runtime errors.

Accordingly, we organize the translation process into two phases: a type-driven phase that focuses on producing a type-compatible translation of the project, and a semantics-driven phase that aims to establish I/O equivalence between each original function and its translation. Both type-compatibility and I/O equivalence are evaluated using the source project's test suite.

We implement these strategies in a translation tool called **Oxidizer**. While the fundamental ideas proposed in our work are agnostic to the programming language, **Oxidizer** specifically targets translating Go projects into Rust projects. We choose this language pair for several reasons. First, there is a strong push to use languages that enforce memory safety at compile time like Rust [6]. Second, Go and Rust operate a similar level of abstraction, and they are often used for similar tasks. Third, Rust offers performance and safety benefits over Go. Specifically, (safe) Rust eliminates data races, and it does not have the overhead of a runtime. Thus there is a strong argument to be made for re-writing Go code to Rust.

Results. We evaluate **Oxidizer** on eight open-source Go projects covering diverse use cases, including banking transactions, statistical analysis, and string algorithms. The biggest project has 9.7K lines of code with 780 functions. To our knowledge, this is the largest project translated to Rust using LLMs.

Our results demonstrate that our approach reliably generates translations that pass the Rust compiler. Crucially, we successfully validate the I/O equivalence for an average of 73% of the translated functions (ranging from 63% to 86%). This is significantly higher than parallel efforts on full project translation. Shiraishi and Shinagawa [53] focused on generating compilable Rust code and reported that most unit tests crashed on their translations. Ibrahimzada et al. [29] validated I/O equivalence for an average of 25.8% of the translated functions (corresponding to 45.9% of the functions actually covered by unit tests) and noted that semantic checks resulted in runtime errors for 24.7% of functions. By contrast, we experienced *no runtime errors*—every failing unit test failed due to assertion errors when using our approach. We also show that both feature mapping and type-compatibility are critical for reliably producing translations of entire projects.

These findings are highly encouraging, especially considering that parallel research [29], which incorporated a user study, found that even partial translations with considerably lower I/O equivalence rates (25.8% overall and 45.9% for covered functions) can substantially reduce the time developers spend translating a codebase.

Contributions.

- We present an LLM-based translation approach for entire projects, achieving a high 73% rate of functions validated for I/O equivalence.
- We introduce feature mapping, a technique that combines predefined translation rules with LLM-based translation to guide the LLM in handling subtle language differences, ensuring semantically correct translations.

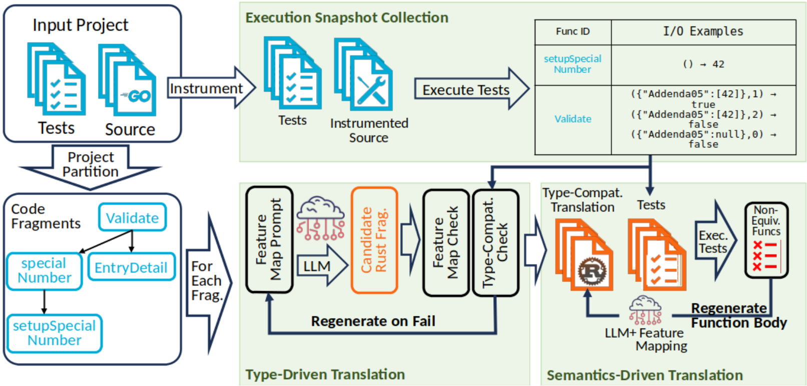

Fig. 1. Workflow of **Oxidizer**.

- We introduce type-compatibility as a prerequisite for I/O equivalence, enabling localized checks at the function signature level to identify errors early and prevent runtime failures.
- We implement our proposed approach in a tool **Oxidizer**, and demonstrate its ability to reliably produce useful Rust translations on eight open-source Go projects.

2 Overview of Oxidizer

Throughout this paper we use **blue** to denote code in the source language (Go in our case), and **orange** to denote code in the target language (Rust in our case). We illustrate our approach using the example in Figure 2, which is a contrived version of code extracted from one of our real-world benchmarks for validating banking transactions [9]. The example consists of three files: "types.go", which declares the type **EntryDetail**; "globals.go", which defines the global variable **specialNumber** initialized by a call to **setupSpecialNumber**; and "validator.go", which contains the definition of the function **Validate**. We assume all three files are located in the same package.

```
// in file globals.go
var specialNumber int = setupSpecialNumber()

func setupSpecialNumber() int { ... }

// in file types.go
type EntryDetail struct {
    Addenda05 []int
}
```

```
// in file validator.go
func Validate(entry *EntryDetail, length int) bool {
    if entry.Addenda05 != nil {
        // length check
        if len(entry.Addenda05) != length {
            return false
        }
        // search for special number
        for _, r := range entry.Addenda05 {
            if r == specialNumber { return true; }
        }
        return false;
    }
    return false;
}
```

Fig. 2. Source Go code consisting of three files: globals.go, types.go, and validator.go

Translation happens in four phases: **execution snapshot collection**, **project partitioning**, **type-driven translation**, and **semantics-driven translation**. Each phase is illustrated in Figure 1.

2.1 Execution Snapshot Collection

We begin by collecting input-output examples for the functions of the source project. These examples are used in both the type-driven and semantics-driven translation phases to check equivalence of the original source function and its translation. We collect the examples by instrumenting each function in the source project to log its inputs and outputs (including side-effects) to files, and then running the project's unit tests. Further implementation details on the execution snapshot collector are provided in Section 7.1.1.

2.2 Project Partitioning

Next, our approach partitions the project into code fragments to be translated individually. We create a fragment for each function/method, type definition (i.e. struct or interface), and global variable declaration. Mutually-recursive functions are placed in the same code fragment to be translated together. For our running example, we create fragments for: the type definition **EntryDetail**, the global variable **specialNumber**, the function **setupSpecialNumber**, and the function **Validate**. These fragments are then organized into a *dependency graph*, where the graph edges capture a code fragment's usage of other type definitions, global variables, and functions/methods. The relations we have are: **specialNumber** depends on **setupSpecialNumber**; **Validate** depends on **EntryDetail** and **specialNumber**.

2.3 Type-Driven Translation

Next, we move into the type-driven translation phase. We iterate over the code fragments in post-order according to the dependency graph, and prompt the LLM for a translation of each code fragment. The goal of our type-driven translation phase is to get a compiling translation, and to catch certain errors related to types. We defer checking I/O equivalence until the next phase.

Assume we have a translation for the function **setupSpecialNumber**, and are now translating the global variable declaration **specialNumber**. A translation obtained from Claude 3 is given in Figure 3, where the global variable definition is translated to a global static definition.

```
// in file globals.rs
static special_number: i32 = setup_special_number()

fn setup_special_number() -> i32 { ... }

// in file types.rs
struct EntryDetail {
    addenda05: Vec<i32>,
}
```

```
// in file validator.rs
fn validate(length: u32, entry: &EntryDetail)->bool {
    if entry.addenda05.iter().all(|x| *x!=0) {
        // length check
        if entry.addenda05.len() != length as usize {
            return false
        }
        // search for special number
        for r in entry.addenda05.iter() {
            if *r == special_number { return true }
        }
        return false
    }
    return false
}
```

Fig. 3. Incorrect Rust translation for the Go code in Figure 2

Challenge: LLM Incorrectly Maps Source Language Features to Target Language Features. While this may seem like an intuitive translation, it will trigger a compilation error because, while a global variable can be initialized by arbitrary functions in Go, static global variables in Rust can only be initialized by functions that can be evaluated at compile-time.

The difficulty in correctly mapping Go global variable definitions to Rust highlights a broader challenge: LLMs often struggle with accurately translating language features, particularly when there are subtle syntactic or semantic differences between constructs in the source and target languages [61]. In such cases, LLMs may incorrectly mimic source language syntax in the target language or misalign source APIs with target ones, resulting in code that fails to compile or is not semantically equivalent.

Solution: Feature Mapping Rules. To address this issue, we propose to use predefined feature mapping rules to constrain the LLM-based translation. These rules are tailored to a specific pair of source and target languages. For the particular case of a global variable definition, several workarounds are available, utilizing constructs like **Lazy** and **lazy_statics**, which are used for lazy initialization of static variables, ensuring that a value is only computed when it is first accessed. Essentially, they defer function evaluation until the variable is accessed for the first time. We pick the former option, **Lazy**, and instruct the LLM on the desired target feature. Consequently, Claude 3 generates the correct translation for **special_number** shown in Figure 4. To ensure rules are followed, we pair each translation rule with a set of static, symbolic checks, and re-query the LLM for a new translation if these checks fail.

```
// in file globals.rs
static special_number: Lazy<i32> =
        Lazy::new(|| setup_special_number());
fn setup_special_number() -> i32 { ... }
// in file types.rs
pub struct EntryDetail {
    addenda05: Option<Vec<i32>>,
}
```

```
// in file validator.rs
use crate::types::EntryDetail;
use crate::globals::special_number;
fn validate(entry: &EntryDetail, length: u32)->bool {
    if entry.addenda05.is_some() {
        // same as Figure 3
        ...
```

Fig. 4. Correct Rust translation for the Go code in Figure 2

Challenge: Incompatible Types. Now assume we move on to translating the **EntryDetail** type definition, and obtain the translation given in Figure 3. While the translation compiles successfully, errors remain. In the original Go code, the **Addenda05** field of **EntryDetail** has type **[]int**, which is nullable, whereas in the Rust translation, the corresponding field **addenda05** has type **Vec<i32>**, which is not nullable. This translation is semantically inequivalent to the original, and will cause semantic validation to fail in the next phase of translation. Moreover, if we continue translating the rest of the code fragments, they will use this incorrectly translated type definition, which will increase the number of repairs we will need to make in the future. This is illustrated by the translation for the function **validate** in Figure 3, which checks that all values in **entry.addenda05** are non-zero, whereas in the original Go function, **Validate** checks **entry.Addenda05 != nil**.

Solution: Type-Compatibility Checks. To catch these semantic in-equivalences early on, we define a notion of *type-compatibility*. Type-compatibility ensures that concrete values of a type in the source language can be converted to concrete values of the corresponding type in the target language, a property necessary for validating I/O equivalence between functions in the source language and target language. We validate type-compatibility using the concrete values collected in the first phase.

The translation given in Figure 3 fails to satisfy this definition because **nil** can inhabit the **Addenda05** field of **EntryDetail** in the Go source, but the same is not true for the Rust translation. Although this error would eventually be detected by the semantic validation check in the next phase, this is delayed until **Validate** is translated (as semantic validation checks that functions in the original and translated code are I/O equivalent). By that stage, additional code has been generated, making the error more difficult to diagnose: it could mistakenly appear as an issue with the **validate** translation rather than with **EntryDetail**. Also, fixing the error would require backtracking. With type-compatibility checks, we can catch this error immediately after generating **EntryDetail**, avoiding these complications.

Following the failure of the type compatibility check, we re-query the LLM, and obtain the correct translation **EntryDetail** in Figure 4, which wraps **Option** around **Vec**. We also perform type compatibility checks for function/method signatures, which assert the compatibility of their constituent types. For our example, the signature of **Validate** would also fail the type compatibility check with its Rust translation **validate** in Figure 3 because the two arguments are swapped. After re-querying, Claude 3 returns the correct translation in Figure 4.

Visibility modifiers and import statements. LLMs often struggle to generate accurate visibility modifiers and import statements when the organizational principles of the source and target languages differ. To mitigate this, we apply post-processing to all translations. While Rust visibility modifiers are derived from their Go counterparts, import statements require a dependency analysis. Figure 4 illustrates the correct visibility modifiers and import statements for our running example.

2.4 Semantics-Driven Translation

After the type-driven translation phase, we assume we have a *type-compatible project translation*, which compiles successfully, and all type definitions and function/method signatures in the source project are type-compatible with their translations, as shown in Figure 4. We then move to the semantics-driven translation phase, during which we test the translated functions for I/O equivalence with the corresponding functions in the source project. We use the I/O examples from the first phase to construct a unit test for each translated function. Each unit test runs the collected inputs on the function, and asserts the returned output matches the collected output. To isolate a particular function's translation during I/O equivalence validation, we mock the translated function's callees, which ensures that I/O equivalence failures are due to bugs in the function under test, and allows us to focus our repair efforts on that function. In this example, the translation successfully passes the semantic checks. Otherwise, we would freeze its signature and regenerate its body using the LLM, guided by feature mapping.

3 Project Partitioning

The first step in our translation procedure is partitioning the source project into fragments that can be translated individually. We describe our partitioning strategy for Go, though the same approach will apply for most source languages. For simplicity, we assume the project follows the simplified FeatherweightGo [27] specification, as given in Figure 5. A Global declaration defines a package-level global variable x with type T, which is initialized by either a value v or through a function call $f(\overline{v})$. A Type declaration defines a named type T that is either a struct or an interface, with the latter being a set of function signatures. A Function/Method declaration defines a function/method named f with its signature and body. In this simplified language specification, we ignore function bodies as they are not important for the purpose of code partitioning (as we never partition a function's body). A Signature allows multiple input and output types. Notably, the idiomatic Go error handling mechanism makes use of multiple return types (where some correspond to the possible errors). Then, a Go project consists of a set of top-level declarations $\overline{D}$ for global variables, types, functions and methods, which can be spread across multiple files. We partition the Go project according to these declarations. We made this decision as each declaration carries distinct type and semantic information, making it suitable for independent translation and correctness checks, as discussed further in the paper.

4 Translating an Individual Code Fragment

In this section, we describe the translation of an individual code fragment. In the following sections, we explain translating the entire project.

The most critical aspect when translating a code fragment is our *feature mapping rules*. As illustrated previously in section 2, LLMs don't always generate the semantically correct feature mapping. Our observation is that, while getting these translations correct is challenging for an LLM, a rule-based approach can predefine mappings between features of the source and target

Function Name			f, g
Field Name			F, G
Variable Name			x, y
Type Name			T, U, I
Value			v, w
Global		::=	$\textbf{var}\ x\ T\ =\ v \mid f(\overline{v})$
Type		::=	$\textbf{type}\ T\ \ell$
TypeLiteral	ℓ	::=	$\textbf{struct}\ \{\overline{F\ T}\} \mid \textbf{interface}\ \{\overline{f\ S}\}$
Fn		::=	$\textbf{func}\ f\ S\ \{\dots\}$
Method		::=	$\textbf{func}\ (x\ T)\ f\ S\{\dots\}$
Signature	S	::=	$(\overline{x\ T})\ \overline{U}$
Declaration	D	::=	Global | Type | Fn | Method
Project	P	::=	$\overline{D}$

Fig. 5. Simplified Go specification

Function Name			f, g
Field Name			F, G
Variable Name			x, y
Type Name			T, U
Trait Name			I
Value			v, w
Static		::=	$\textbf{static}\ x : T\ =\ v$
Type		::=	$\textbf{struct}\ T\ \{\overline{F : T}\}$
Trait		::=	$\textbf{trait}\ I\ \{\overline{\textbf{fn}\ f\ S_m}\}$
Fn		::=	$\textbf{fn}\ f\ S_f\ \{\dots\}$
Impl		::=	$\textbf{impl}\ T\ \{\overline{\textbf{fn}\ f\ S_m\ \{\dots\}}\}$
FnSignature	S_f	::=	$(\overline{x : T}) \to \overline{U}$
MethodSignature	S_m	::=	$(\textbf{self}, \overline{x : T}) \to \overline{U}$
SelfType	**self**	::=	**&self** | **&mut self** | ...

Fig. 6. Simplified Rust specification

languages. While we could write purely symbolic translation rules, this approach has two major disadvantages. First, as prior work [25, 44, 60] has shown, symbolic translation rules often produce unidiomatic code. Second, a purely symbolic approach demands an exhaustive set of translation rules covering every feature. This would incur a significant amount of developer effort.

This motivates us to take a hybrid approach that combines an LLM and static analysis to implement feature mapping rules. We express a feature mapping rule as a judgment, and implement it in three parts: (1) a syntactic pattern that detects when the rule applies, (2) a natural language description of the translation rule, which is provided to an LLM, and (3) a set of static checks that validate whether the rule was applied correctly (i.e. the validation check). We explain our feature mapping rules for Go to Rust in subsection 4.1.

Algorithm 1 describes how we apply our feature mapping rules (and how we obtain candidate translations as well). The algorithm takes a source code fragment to translate, **D**, and a re-query budget *requery_budget*. It returns a translation of **D** that satisfies the conclusions of any applicable feature mapping rules.

We first statically analyze **D** to determine which, if any, feature mapping rules apply, and retrieve the natural language descriptions of the rules, and their validation checks. Next, we generate a summary of the translations for all dependencies of **D**. This summary is a condensed representation of the translated code, including function and method signatures (without bodies), type definitions, and global variable declarations. We then incorporate both the translated code and the dependency summary into the LLM prompt, as shown in Figure 7. We then iteratively query the LLM with this prompt to obtain a translation, and check that the feature mapping rules were correctly applied. Once the validation checks are satisfied, we compute the necessary visibility modifiers. If the *requery_budget* runs out, we abort translation, but this did not happen in our experiments.

4.1 Mapping Language Features

In the rest of the section, we discuss the Go-to-Rust feature mapping rules for basic features (e.g. variable definitions, struct definitions), interface declarations and error handling.

4.1.1 Mapping basic features. Figure 8 illustrates the inference rules for mapping basic Go features. Each rule's premise states that, given a Go fragment **D** containing a feature of interest, we query

Algorithm 1 Translation of an individual fragment with feature mapping

```
Require: Source Code Fragment D, Requery Budget requery_budget
Ensure: target_code, Target Language Code that Satisfies Feature Mapping Rules
    rule_descriptions, conclusions ← GetFeatureMappingRules(D)
    dependencies ← GetDependenciesSummary(D)
    prompt ← GeneratePrompt(D, rule_descriptions, dependencies)
    correct_mapping ← false
    while ¬correct_mapping do
        code ← QueryLLM(prompt)
        correct_mapping ← CheckFeatureMapping(code, conclusions)
        requery_budget ← requery_budget − 1
        if requery_budget ≤ 0 then
            AbortTranslation()
        end if
    end while
    target_code ← SetVisibilityModifiers(target_code)
    return target_code
```

```
Below, you are given a fragment of Go code. Your job is to translate it to Rust.
${SOURCE_CODE_FRAGMENT}

The dependencies of the above Go code have already been translated to Rust. A condensed version
of their translations is given below. Make sure to use them in your translation.
${TRANLSATED_DEPENDENCIES}

When translating the fragment of Go code, apply the following translation rules.
${FEATURE_MAPPING_RULES}
```

Fig. 7. LLM prompt template

the LLM to produce **code**, denoted by $\rightsquigarrow_{\text{LLM}}$. The augmentation of the prompt with the specified feature mapping is implied within $\rightsquigarrow_{\text{LLM}}$. The symbol $\Downarrow$ in the conclusion denotes the validation check confirming that **code** includes the expected Rust feature. If this condition is met, then the fragment **D** is successfully translated to **code**.

For example, the rule (Map-Var-Init) says that given a Go global variable definition **var** $x = f(\overline{v})$, where the variable is initialized by a function call, we construct a prompt to query the LLM for a translation **code**. The expected translation for the global variable in **code** is a Rust global static definition with its initializer wrapped in **Lazy::new**, as checked by $\Downarrow$. If the check succeeds, then **code** is treated as the translation of **D**. Code wrapped by **Lazy::new** is evaluated upon first access, which we consider to be an emulation of Go global **var** initialization. While there are alternative emulation methods, we choose **Lazy::new** for its simplicity. However, different developers may customize the rule differently, opting for a different alternative.

Beyond the **Lazy::new** wrapper, we don't impose any additional constraints. The LLM is free to generate names (e.g., corresponding to y in the conclusion of the rule), types (e.g., corresponding to U), and initializers (the argument of **Lazy::new** is not required to be a function call). For idiomaticity purposes, we do not constrain the variable, type and field names: Go prefers Camel Case, whereas Rust prefers Snake Case.

The remaining rules in Figure 8 outline feature mappings for struct type definitions, functions, and methods: a Go struct type is mapped to a Rust struct type, Go functions translate to Rust free-standing functions, and Go methods map to Rust inherent implementations. Similar to the (Map-Var-Init) rule, names and types in the conclusion are left unconstrained.

$$\frac{\mathbf{D} : \mathbf{var}\ x\ T = f(\overline{v}) \rightsquigarrow_{\text{LLM}} \mathbf{code}}{\mathbf{code} \Downarrow \mathbf{static}\ y : \mathbf{Lazy\text{<}}U\mathbf{\text{>}} = \mathbf{Lazy{::}new(||}\ \ldots\mathbf{)} \vdash \mathbf{D} \mapsto \mathbf{code}}\ \text{(Map-Var-Init)} \qquad \frac{\mathbf{D} : \mathbf{type}\ T\ \mathbf{struct}\ \{\ldots\} \rightsquigarrow_{\text{LLM}} \mathbf{code}}{\mathbf{code} \Downarrow \mathbf{struct}\ U\{\ldots\} \vdash \mathbf{D} \mapsto \mathbf{code}}\ \text{(Map-Struct)}$$

$$\frac{\mathbf{D} : \mathbf{func}\ (x\ T)\ f\ S \rightsquigarrow_{\text{LLM}} \mathbf{code}}{\mathbf{code} \Downarrow \mathbf{impl}\ Tr\ \{\mathbf{fn}\ g\ S_m\{\ldots\}\} \vdash \mathbf{D} \mapsto \mathbf{code}}\ \text{(Map-Method)} \qquad \frac{\mathbf{D} : \mathbf{func}\ f\ S \rightsquigarrow_{\text{LLM}} \mathbf{code}}{\mathbf{code} \Downarrow \mathbf{fn}\ g\ S_f \vdash \mathbf{D} \mapsto \mathbf{code}}\ \text{(Map-Fn)}$$

Fig. 8. Basic feature mapping

4.1.2 Mapping error handling. Go and Rust adopt drastically different error handling styles. In Go, error handling often involves returning an **error** as an additional return value, which is then checked by the caller using simple if statements, as illustrated in Figure 9. The builtin **error** type is an **interface** type containing exactly one method: **Error() string**. Although custom error types can be constructed separately, they often implement this interface and are passed around as **error** rather than concrete types. Then, type assertions are used to recover the concrete underlying type. In this example, the caller **addendaFieldInclusion** checks whether the error returned by **checkAddenda02** is an instance of the custom error **BatchError** (i.e. **err.(*BatchError)**). By contrast, idiomatic Rust represents potential errors using the **Result** type, enabling monadic error propagation using the **?** operator. Rust does not have a built-in error type; instead, idiomatic Rust often involves passing concrete error types, with additional code needed to convert between them as necessary.

```go
func f() (uint32, error) { ... }
func g() error {
    if x, err := f(); err != nil {
        if _, ok := err.(*BatchError); ok {
            return err
        }
        return BatchError{err}
    }
    ...
}
```

Fig. 9. Source Go snippet

```rust
fn f() -> Result<u32, FError> { ... }
fn g() -> Result<(), BatchError> {
    let x = f().map_err(|err| {
        if let Ok(err) = err.downcast::<BatchError>()
        {
            return err
        }
        return BatchError(err)
    })?;
    ...
}
```

Fig. 10. Incorrect translation to Rust

Figure 10 presents a translation generated by LLMs for the code in Figure 9. Although this code looks idiomatic, it does not compile due to the erroneous emulation of the source syntax: the Go type assertion **err.(*BatchError)** is translated to **err.downcast::<BatchError>()**, which is a compilation error since **FError** does not have a **downcast** method.

There are various ways to map error handling between Go and Rust, giving developers flexibility to choose their preferred approach. In this work, we chose to predefine a unified concrete Rust error type. We selected **anyhow::Error** from a widely used third-party Rust library for idiomatic error handling [57]. The advantage of **anyhow::Error** is that it acts as a lightweight wrapper for any type implementing the built-in **std::error::Error** trait, facilitating easy conversion between such types and enabling instance checking for custom error types that also implement **std::error::Error**. Additionally, the trait **std::error::Error** relies on **std::fmt::Display** for pretty-printing of error messages, which provides a natural correspondence for the **Error() string** method.

We define the feature mapping rules for error handling in Figure 11. The rule (Map-Custom-Error) instructs the LLM to translate the **error** interface implementation into three Rust trait implementations so that the translated custom error implements **std::error::Error**. The rule (Map-Error-Handling-Fn) ensures that error handling Go functions are translated to idiomatic Rust, where the error type is constrained to be the predefined **anyhow::Error**.

$$\frac{\mathbf{D} : \mathbf{func}\ (t\ T)\ \mathbf{Error}()\ \mathbf{string}\ \{\dots\} \rightsquigarrow_{\text{LLM}} \mathbf{code}}{\mathbf{code} \Downarrow \begin{array}{c} \mathbf{impl\ Debug\ for}\ U\ \{\dots\} \\ \mathbf{impl\ Display\ for}\ U\ \{\dots\} \\ \mathbf{impl\ std::error::Error\ for}\ U\ \{\dots\} \end{array} \vdash \mathbf{D} \mapsto \mathbf{code}} \qquad \text{(Map-Custom-Error)}$$

$$\frac{\mathbf{D} : \mathbf{func}\ f\ (\overline{x\ T})\ (\overline{U}, \mathbf{error}) \rightsquigarrow_{\text{LLM}} \mathbf{code}}{\mathbf{code} \Downarrow \mathbf{fn}\ g\ (\overline{x : Tr}) \longrightarrow \mathbf{Result<}\overline{U}r, \mathbf{anyhow::Error>} \vdash \mathbf{D} \mapsto \mathbf{code}} \qquad \text{(Map-Error-Handling-Fn)}$$

Fig. 11. Mapping error handling

4.1.3 *Mapping Go interfaces to Rust traits.*

Go **interface** and Rust **trait** are similar abstractions in the sense that both achieve polymorphism by defining a collection of methods that get implemented for different types. Figure 12 presents two Go interfaces, **Batcher** and **canValidate**, extracted from the real-world benchmark **ach**. Given that a Go interface is *structural* and **Batcher** and **canValidate** share the method signature for **Validate**, they exhibit a sub-interfacing relation. As a result, a value of type **Batcher** is *assignable* to a variable of type **canCandidate**. Furthermore, the concrete implementation of the **Validate** method only needs to be provided once.

```go
type Batcher interface {
    SetHeader(*BatchHeader)
    Validate() error
}


type canValidate interface {
    Validate() error
}


func (t *T) SetHeader(*BatchHeader) { ... }
func (t *T) Validate() error        { ... }
```

Fig. 12. Source Go snippet

```rust
trait Batcher {
    fn set_header(&mut self, _: Option<BatchHeader>);
    fn validate(&self) -> Result<()>;
}
trait canValidate {
    fn validate(&self) -> Result<()>;
}
impl Batcher for T {
    fn set_header(&mut self, _: Option<BatchHeader>)
    { ... }
    fn validate(&self) -> Result<()> { ... }
}
impl canValidate for T {
    fn validate(&self) -> Result<()> { ... }
}
```

Fig. 13. Incorrect translation to Rust

Figure 13 presents a translation obtained from the LLM for Figure 12 using Rust traits. Conversely to Go interfaces, a Rust **trait** is *nominal*, and thus the sub-interfacing relation does not hold. This is a translation error as, for any Go code where a value of type **Batcher** is assigned to/upcast to a variable of type **canValidate**, there will not exist any compilable Rust translation. Moreover, this translation also poses a challenge for our design based on modular translation and validation of individual functions: all the method implementations for an interface must be translated at the same time in order for the code to pass the compiler check.

To address these challenges, we propose a target feature that decomposes a trait definition into multiple sub-traits (note that our notion of sub-trait doesn't induce a subtyping relation), each containing a single method signature. Methods that appear in multiple interfaces only need one such sub-trait. Then the main trait (corresponding to the original Go interface) is bounded by all its sub-traits. Figure 14 illustrates this solution for the traits in Figure 12. If we take the example of **canValidate**, it gets translated to the following Rust components:

(1) *main trait* **canValidate**, which is bounded by its corresponding sub-trait **canValidate_Validate**;
(2) *sub-trait* **canValidate_Validate**, corresponding to method **Validate**;
(3) implementation of sub-trait **canValidate_Validate** for T;
(4) implementation of the main trait for T; this is automatically generated by **Oxidizer** as part of **PostProcessing** in Algorithm 1 once all sub-traits (in this case just **canValidate_Validate**) are implemented.

```
// main trait
trait canValidate: canValidate_Validate {}
// sub-trait
trait canValidate_Validate {
    fn validate(&self) -> Result<()>;
}
// auto impl
impl<T> canValidate for T
where
    T: canValidate_Validate {}

// main trait
trait Batcher: canValidate_Validate +
     Batcher_SetHeader + canValidate {}
// sub-trait
trait Batcher_SetHeader {
    fn set_header(&mut self, _: Option<BatchHeader>);
}
```

```
// auto impl
impl<T> Batcher for T
where
    T: canValidate_Validate + Batcher_SetHeader {}

impl Batcher_SetHeader for T {
    fn set_header(&mut self, _: Option<BatchHeader
        >)
    { ... }
}

impl canValidate_Validate for T {
    fn validate(&self) -> Result<()> { ... }
}
```

Fig. 14. Correct translation of Go **interface** to Rust **trait**

We follow a similar approach for **Batcher**, where we reuse the already generated sub-trait **canValidate_Validate** corresponding to method **Validate**. The main trait **Batcher** is bounded by the sub-traits **canValidate_Validate** and **Batcher_SetHeader**. Additionally, we add an extra bound **canValidate** for **Batcher** to properly reflect the sub-interfacing relation present in the source Go code. Due to space limitations, we omit the formal rule for interface mapping.

5 Type-Driven Translation

So far, we have focused on translating individual code fragments. In this section, we shift our attention to the overall project translation, specifically discussing the type-driven phase. The objective of this phase is to generate a *type-compatible project translation*.

The type-driven translation phase follows Algorithm 2. We start from a Go project P that gets partitioned and the code fragments get organized into a dependency graph by the **DependencyAnalysis** function. We then traverse the resulting graph in post order according to **PostOrder**. For each code fragment **D**, we apply **FeatureMapping** (as described in Algorithm 1), **CompilationCheckAndRepair** (as described in Section 5.1) and we check type-compatibility with respect to the execution snapshots extracted by the **ExecutionSnapshotsCollector** from the project's unit tests (the type-compatibility check is discussed in Section 5.2 and the execution snapshots collector is described in Section 7.1.1). If the resulting translation, **target_code**, is both compilable and type-compatible, we store it in the *translations* map and move to the next code fragment. Otherwise, if we reached the maximum number of tries and the translation is type-compatible, we mock the current function and move to the next fragment (mocking is described in Section 5.3). Type-compatibility provides a principled approach for mocking a function and proceeding with the translation of the rest of the project when the LLM encounters difficulties. However, mocking is only feasible if the generated signature is compatible with that of the source function. If the current translation is not type-compatible, we have no other choice than to abort the translation. In our experimental evaluation, this situation did not arise (except during the ablation study).

Algorithm 2 Type-driven translation phase

```
Require: Source Go Project $P = \overline{\mathbf{D}}$, Budget max_tries, Budget requery_budget, P's test suite test_suite
Ensure: translations, Which Maps Code Fragments, $\overline{\mathbf{D}}$, to Corresponding Type-Compatible Rust Translations
  translation_order ← PostOrder(DependencyAnalysis(P))
  translations ← {}
  for D ∈ translation_order do
      budget ← max_tries
      execution_snapshots ← ExecutionSnapshotsCollector(D, test_suite)
      while true do
          target_code ← FeatureMapping(D, requery_budget)
          target_code, compiled ← CompilationCheckAndRepair(translations + [D : target_code])
          type_compatible ← TypeCompatibilityCheck(D, target_code, execution_snapshots)
          if compiled ∧ type_compatible then
              translations[D] ← target_code
              break
          end if
          budget ← budget − 1
          if budget ≤ 0 then
              if type_compatible then
                  translations[D] ← Mock(target_code)
                  break
              end if
              AbortTranslation()
          end if
      end while
  end for
```

5.1 Compilation check and repair

The compilation check and repair routine assembles all previously translated fragments into a Rust project that can be sent to the Rust compiler. This task is complex, as real-world Go projects often have their source code spread across intricate project layouts, and Go and Rust follow different organizational principles. In Go, a project is composed of packages that may span multiple files, with private items accessible within the same package even if they are in different files. By contrast, Rust follows a module system where each file in a cargo project is treated as a separate module, and items in one module are not directly accessible in another. Generating appropriate import statements is challenging and has been reported as a leading source of compilation errors in LLM-based translations [61].

Internal imports. We symbolically generate the appropriate import statements by using our dependency analysis, and amend the translated code accordingly.

External imports. Translation of Go code that uses libraries (either standard or 3rd party) is challenging because we either need to translate the entire library, or we must find appropriate Rust libraries that have the same functionality as their Go counterpart. Our solution is to rely on the LLM to find appropriate library mappings – our experience is that, owing to their large training sets, LLMs are frequently able to identify suitable libraries. Then, we symbolically generate the corresponding external import statements, and add them to the translation.

Compilation repair. If the compilation check fails, we requery the LLM to obtain a repair for the current fragment being translated. This repair process is localized, restricting changes to only the fragment in question and preventing modifications to other parts of the project. The requerying process leverages error messages from the Rust compiler and follows the approach outlined in [22].

5.2 Type-compatibility

The usage of a type within a project depends on local preconditions at its use sites. For instance, in Go, the **int** primitive type represents a platform-dependent signed integer. To fully preserve semantics, **int** should map to **isize** in Rust. However, **int** is often used for array indexing in Go, where **usize** is typically used in Rust. Depending on the local preconditions at the use sites (i.e. whether **int** is used for array indexing at all occurrences), the appropriate translation could be either **usize** or **isize**.

Instead of enforcing strict type equivalence between the original and translated types, we introduce “type-compatibility”, inspired by inter-language interoperability [38], which addresses scenarios where values are sent over the boundary between two languages. The core idea is that any feasible value that can inhabit a type in the original Go project should also be able to inhabit the corresponding type in the translated Rust project. By “feasible”, we mean values that adhere to the use-site preconditions in the source Go code. These preconditions could, in theory, be statically inferred, but doing so is challenging. Therefore, in this work, we ensure value feasibility by selecting only values from the execution snapshots extracted from the project’s test suite.

Type-compatibility checks. We leverage the data marshaling mechanisms available in both languages, given that Go and Rust provide well-maintained libraries for marshaling complex data types. Each Go type T has an associated serialization function, $\mathcal{S}_T$, which converts a Go value of type T into JSON. Similarly, each Rust type T_r has a corresponding serialization function, $\mathcal{S}_{T_r}$.

For both languages, we require fallible deserialization functions—$\mathcal{D}_T$ for Go and $\mathcal{D}_{T_r}$ for Rust—which take JSON and return a value of the underlying type (T and T_r, respectively). If the JSON object does not conform to the underlying type, the deserialization functions return an error. These serialization and deserialization functions are expected to satisfy the following round-tripping property: $(\mathcal{D}_T \circ \mathcal{S}_T)(v) \equiv v$ and $(\mathcal{D}_{T_r} \circ \mathcal{S}_{T_r})(v) \equiv v$. In the other direction, given a JSON object j, if $\mathcal{D}_T\ (j)$ and $\mathcal{D}_{T_r}\ (j)$ do not abort with an error, we have $(\mathcal{S}_T \circ \mathcal{D}_T)(j) \equiv j$ and $(\mathcal{S}_{T_r} \circ \mathcal{D}_{T_r})(j) \equiv j$.

Type-compatibility checks for type definitions. For each type definition in the translated Rust code, we check type-compatibility with its corresponding Go type definition according to Definition 1.

Definition 1. (Type-compatibility) *Given a type T in the source code, a set V of feasible values of type T and a target type T_r, we say that T_r is compatible with T with respect to V (written as $T_r \Lleftarrow_V T$) if any $v \in V$ can cross the boundary to the target language as v_r of type T_r such that $v_r = (\mathcal{D}_{T_r} \circ \mathcal{S}_T)\ (v)$ and $v = (\mathcal{D}_T \circ \mathcal{S}_{T_r})\ (v_r)$.*

Type compatibility checks for function and method signatures. For each function declaration in the translated Rust code, we check type-compatibility between its signature and the one of its Go counterpart according to Definition 2.

Definition 2 (Type-compatible function signatures). *Given a function **func** $f(\overline{x\ T})\ \overline{U}\ \{...\}$ in the source code, sets $\overline{V_T}$ of feasible values for types $\overline{T}$, sets $\overline{V_U}$ of feasible values for types $\overline{U}$ and a target function **fn** $f(\overline{x:\ T_r})$ -> $\overline{U_r}\ \{...\}$, we say that the target function is type-compatible with its source counterpart if the following hold:*

- *$\overline{T_r}$ is respectively type-compatible to $\overline{T}$ with respect to $\overline{V_T}$*
- *$\overline{U_r}$ is respectively type-compatible to $\overline{U}$ with respect to $\overline{V_U}$.*

The type-compatibility check for method signatures is similar to the one for functions, with the addition that the types of the receivers must also be compatible.

For brevity, throughout this paper, we use "type-compatibility" to indicate that a Rust entity B is type-compatible with a Go entity A based on the execution snapshots.

Type-compatible project translation. At the end of the type-driven phase, we expect to obtain a project translation that is type-compatible with the source project based on the feasible values obtained from the execution snapshots, as defined next.

Definition 3 (Type-compatible project translation). *Given a source project P and its translation P_r, P_r is type-compatible with P if:*

i *For each two corresponding type definitions T and T_r from P and P_r, respectively, T_r is type-compatible with T (with respect to the values collected from the unit tests).*
ii *For each two corresponding functions/methods f and f_r from P and P_r, respectively, f_r's signature is type-compatible with f's signature (with respect to the values collected from the unit tests).*
iii *Project P_r passes the target Rust compiler.*

5.3 Function mocking

Certain idioms allowed in the source code are strictly disallowed in the target language, making it challenging for the LLM to produce a viable translation. This poses a significant issue when translating large codebases, as encountering such cases can halt the entire translation process.

```
func updateRanks(ranks *Rank, algorithm Algorithm) {
    for _, word := range ranks.Words {
        weight := algorithm.WeightingHits(word.ID,
              ranks)
        word.Weight = weight
        ...
    }
    ...
}
```

Fig. 15. Source Go snippet

```
pub fn update_ranks(ranks: &mut Rank, algorithm: &dyn
      Algorithm) {
    for word in ranks.words.values_mut() {
        // Failed to compile
        let weight = algorithm.weighing_hits(word.id,
              ranks);
        word.weight = weight;
        ...
    }
}
```

Fig. 16. Incorrect translation to Rust

```
pub fn update_ranks(ranks: &mut Rank, algorithm: &dyn Algorithm) {
    extern "C" {
        // import original Go function directly
        fn updateRanks(..)
    }
    return updateRanks(..)
}
```

Fig. 17. Function mock

For illustration, the Go snippet in Figure 15 is particularly challenging to translate to Rust as it violates invariants demanded by the Rust's borrow checker. In particular, the loop condition **for _, word := range ranks.Words** conceptually mutably borrows **ranks.Words** since the loop body performs mutations through **word**. However, in the loop body, **ranks** is used for the second time: **algorithm.WeightingHits(word.ID, ranks)**. This is similar to the *iterator invalidation* problem in C++, and it is not allowed in Rust. LLMs tend to follow the syntax of the original Go snippet and generate the Rust code in Figure 16, which doesn't pass the Rust compiler.

We observe that the translation process should be able to bypass such scenarios where the LLM struggles to find a compilable translation, and to continue translating the rest of the project. To address this, we define *function mocks*. Such a mock consists of a Rust function signature that passes our type-compatibility check, and a function body that calls the original Go function as-is. For the current example, the mock is given in Figure 17. This necessitates the Go-Rust boundary notions mentioned before, as we need to transform Rust inputs into Go inputs, and the Go output to the Rust output. Assuming **func** $f\ (x\ T)\ U\ \{\dots\}$ is translated into **fn** $g\ (y : T_r) \longrightarrow U_r\ \{\dots\}$, we define the function mock for g to be: $\mathbf{Mock}(g) ::= \mathcal{D}_{U_r} \circ \mathcal{S}_U \circ f \circ \mathcal{D}_T \circ \mathcal{S}_{T_r}$. Notably, function mocking is only possible if we manage to obtain a function signature that passes type-compatibility checks.

A mock function needs to capture the side-effects of the original Go function. For this purpose, we check the pre and post execution states of the Go function and reflect these changes in the Rust translation accordingly. For example, in Figure 15, the Go function **updateRanks** is mutating its input. To capture this behavior, we instrument the function so that input mutations are returned as part of the output. When translating to Rust, we update the **ranks** variable by incorporating these returned mutations. A similar approach is applied to global variables.

6 Semantics-Driven Translation

The objective of this stage is to refine the type-compatible project translation produced by the type-driven translation so that it becomes I/O equivalent to the original. The semantics-driven translation phase follows Algorithm 3. For each translation in the *translations* map produced by the type-driven translation, **I/OEquivalenceCheck** checks whether it is I/O equivalent with the original Go fragment on the execution snapshots extracted by the **ExecutionSnapshotsCollector** from P's unit tests. Notably, this check is local, only focusing on the semantic correctness of the current fragment. To achieve this, we construct mocks for all its callees (if applicable), which emulate the behavior of the original code as described in Section 5.3. If the check succeeds, then we exit the **while** loop and update the *translations* map. Otherwise, we attempt to re-generate and refine the body of the current function by calling **FeatureMapping** and **CompilationCheckAndRepair**, both with *freezing_signature* set to *true* so that they are disallowed from changing types. We keep refining and checking I/O equivalence until we obtain a semantically correct translation or we run out of budget. On exit from the **while** loop, the translation is guaranteed to be type-compatible (as the LLM was disallowed from modifying the signature), but it may still not be I/O equivalent to the original fragment.

6.1 I/O equivalence check

We define two types of semantic checks for a Go project P: one for global variable initializations and another for functions. Our I/O equivalence check operates under the assumption that the target Rust code adheres to the structure specified by the feature mapping in Section 4.1. As we need to pass values between Go and Rust, we make use of the serialization and deserialization functions introduced in Section 5.2.

Definition 4 (I/O equivalence of global variable initialization). *If we have* **D** : ***var*** $x\ T = f(\overline{v})$*, that gets translated to translations*[**D**]*, we say that* **D** *and translations*[**D**] *are I/O equivalent if translations*[**D**] *is of the form* ***static*** y : *Lazy<*U*>* = *Lazy::new(|| . . .)* *and* $\mathcal{S}_T(x) = \mathcal{S}_U($*Lazy::force(*$y$*)*$)$.

Lazy::force forces the evaluation of the static var y. The left and right hand sides of the equality check in the definition are serialized to JSON objects, which allows for plain string comparison.

I/O equivalence of functions. Let's consider the Go function **D** : **func** $f\ (\overline{x\ T})\ (\overline{U}, $ **error**$) \in P$, which has inputs $\overline{i}$, outputs $\overline{o}$, and an additional error output err. By the (Map-Error-Handling-Fn) feature mapping rule in Section 4.1.2, we expect *translations*[**D**] to be of the form **fn** $g\ (\overline{y : Tr}) \longrightarrow$ **Result<**$\overline{U}r$, **anyhow::Error>**. We check I/O equivalence between functions/methods with respect to a set of values $V = \{(\overline{i}, \overline{o'}, err)^*\}$ collected by running the unit tests in the original Go repository.

We overload the output by considering $\overline{o'}$ to be an extension of the actual output $\overline{o}$ that accounts for possible side-effects.

Definition 5 (I/O equivalence of functions). *If we have* $\mathbf{D}$: $\mathbf{func}$ f $(\overline{x\ T})$ $(\overline{U}, \mathbf{error})$*, we say that* $\mathbf{D}$ *and translations*$[\mathbf{D}]$ *are I/O equivalent with respect to* V *if translations*$[\mathbf{D}]$ *is of the form* $\mathbf{fn}\ g\ (\overline{y : Tr}) \longrightarrow \mathbf{Result{<}}\overline{Ur}, \mathbf{anyhow{::}Error{>}}$ *and*

$$\bigwedge \begin{array}{l} err \equiv \mathbf{nil} \longrightarrow \mathbf{Ok}(\overline{res}) = g(\mathcal{D}_{\overline{Tr}}(\mathcal{S}_{\overline{T}}(\bar{i}))) \wedge \mathcal{S}_{\overline{Ur}}(\overline{res}) \equiv \mathcal{S}_{\overline{U}}(\overline{o'}) \\ err \not\equiv \mathbf{nil} \longrightarrow \mathbf{Err}(\_) = g(\mathcal{D}_{\overline{Tr}}(\mathcal{S}_{\overline{T}}(\bar{i}))) \end{array}$$

We abuse the notation using $S_{\overline{T}}(\bar{i})$ to mean pairwise application of the corresponding serialization function to each individual input in the tuple $\bar{i}$. The definition says that, for any input $\bar{i}$, if the Go function f succeeds on $\bar{i}$, then so should the Rust translation g on the corresponding input $\mathcal{D}_{\overline{Tr}}(\mathcal{S}_{\overline{T}}(\bar{i}))$, and the output states should be the same, $\mathcal{S}_{\overline{Ur}}(\overline{res}) \equiv \mathcal{S}_{\overline{U}}(\overline{o'})$. Alternatively, if f fails, then so should g.

Algorithm 3 Semantics-driven translation phase

```
Require: translations Maps Code Fragments, $\overline{\mathbf{D}}$, to Type-Compatible Rust Translations, Budget requery_budget, Budget
    max_tries, P's test suite test_suite
Ensure: translations Maps code fragments, $\overline{\mathbf{D}}$, to Their Final Rust Translations
  for D, target_code ∈ translations do
      budget ← max_tries
      execution_snapshots ← ExecutionSnapshotsCollector(D, test_suite)
      while budget > 0 do
          equivalent ← I/OEquivalenceCheck(D, target_code, execution_snapshots)
          if equivalent then
              break
          end if
          compiled ← false
          while ¬compiled ∧ budget > 0 do
              target_code ← FeatureMapping(D, requery_budget, freezing_signature = true)
              target_code, compiled ← CompilationCheckAndRepair(target_code, freezing_signature = true)
              budget ← budget − 1
          end while
      end while
      translations[D] ← target_code
  end for
```

7 Experimental Evaluation

We conduct an evaluation of **Oxidizer** to assess the following research question:

- How effective is **Oxidizer** at translating entire projects? Specifically, how much of the translated code can compile, and how many of the functions can be validated I/O equivalent?
- How much do our proposed type-compatibility checks and feature mapping rules benefit translation?
- How much does our proposed semantics-driven phase benefit translation?
- How does **Oxidizer** compare to parallel works in whole-repository translation?

7.1 Experimental Setup

7.1.1 Implementation. **Oxidizer** (available at https://zenodo.org/records/15242640) takes as input (1) a Go project, (2) *requery_budget* for Algorithm 1, (3) *max_tries* for Algorithm 2, and (4) *max_tries* for Algorithm 3. **Oxidizer** can be configured to use both proprietary and open-source LLMs. **Oxidizer** outputs a Rust translation of the input project. The translation may have some function/method bodies replaced with mocks as described in Section 5. The core code modules of **Oxidizer** are (1) the execution snapshot collector, (2) the project partitioner, (3) the type-driven translator, and (4) the semantics-driven translator. All modules are implemented in python, and use py-tree-sitter for parsing and instrumenting code. The project partitioner, type-driven translator, and the semantics-driven translator implement the algorithms described in the previous sections, so we focus on the execution snapshot collector, and testing I/O equivalence.

The execution snapshot collector collects inputs and expected outputs for functions in the source project. It first instruments each function in the source project with statements that log the inputs and outputs of all functions in JSON format. It then executes the unit tests of the source project to collect these inputs and outputs. These input-output examples are used to test type-compatibility during type-driven translation, and they are used to create unit tests for the I/O equivalence check in the semantics-driven translation.

To accurately capture side effects, the execution snapshot collector tracks the following types of mutations: (1) *Input mutations*—the collector records both pre- and post-execution states of function inputs; (2) *Global state mutations*—the collector also records pre- and post-execution states of global variables during function execution. These snapshots are then used to verify whether the translated Rust function correctly preserves input and global state mutations. In our benchmarks, we observed that global states remain unchanged during execution and are only modified during initialization, which occurs when the file is loaded into memory. Therefore, we impose additional validation steps to ensure that global states are correctly initialized in the Rust translation.

Before performing semantics-driven translation, **Oxidizer** creates a Rust unit test for each function in the Rust translation. For a given function in Rust, its unit test first loads all input-output examples collected for the corresponding function in the source project using JSON deserialization. It then executes all loaded inputs on the Rust function, and compares the computed outputs to the expected outputs that were loaded. A unit test passes if and only if all computed outputs match the expected outputs. If we are not able to collect any input-output examples for a given function, we create an empty unit test that automatically fails. The percentage of all unit tests that pass is equivalent to the percentage of functions that are validated I/O equivalent.

7.1.2 LLMs. We use Anthropic's Claude 3 Sonnet [3] provided by Amazon Bedrock. Prior work [25] has shown that Claude 3 Sonnet performs similarly to other state-of-the-art proprietary LLMs, such as GPT-4o [42] and Gemini Pro [7], thus we believe our results apply to them as well. In order to enable others to reproduce our results deterministically without the need to pay to run the LLM, **Oxidizer** logs the inputs and outputs of the LLM, and supports replaying these logs.

7.1.3 Benchmarks. We use real-world projects collected from GitHub as our benchmarks. Our main criteria for selecting projects are that (1) the project has more than 100 stars on GitHub or is actively maintained (specifically, the project has commits in the last 6 months) and (2) the project only makes use of Go standard libraries. As explained in Section 5.1, our approach supports generating code that utilizes third-party libraries and we already do so for standard Go libraries that don't always have Rust standard library correspondents and instead require Rust 3rd party libraries (e.g. the regexp Go library [8]). However, for this evaluation, we selected Go benchmarks that do not rely on such libraries as standard Go libraries are more likely to have equivalent libraries (including third-party options) available in Rust. The benchmarks we select are listed below.

Table 1. Benchmark details

Project	LoC	# Functions/ Methods	# Structs/ Interfaces	Unit Test Statement Coverage	Unit Test Function Coverage	Stars	Forks
geo	9766	780	75	87.0 %	87.6 %	1.7k	182
ach	6642	369	75	92.9%	95.2%	442	145
go-edlib	639	25	1	100%	100%	480	24
stats	1241	79	8	93.2%	99.2%	2.9K	168
textrank	1132	69	20	94.6%	98.3%	205	22
histogram	314	23	5	43.2%	61.9%	175	31
gonameparts	413	15	2	96.1%	100%	42	5
checkdigit	428	29	9	100%	100%	110	7

Table 2. Translation results for **Oxidizer**

Benchmark	**Full**		**No Semantics-Driven**		**No Type Check**		**No Feature Map**	
	% Compiled Func. / LoC.	**% Equiv. Func. / LoC.**	**% Compiled Func. / LoC.**	**% Equiv. Func. / LoC.**	**% Compiled Func. / LoC.**	**% Equiv. Func. / LoC.**	**% Compiled Func. / LoC.**	**% Equiv. Func. / LoC.**
geo	94 / 85	68 / 59	94 / 84	66 / 57	93 / 79	62 / 51	1 / 0	0
ach	96 / 92	65 / 62	96 / 92	64 / 61	93 / 86	62 / 61	5 / 10	0
textrank	97 / 95	75 / 78	97 / 95	74 / 78	97 / 95	67 / 76	16 / 9	0
go-edlib	100 / 100	81 / 89	100 / 100	81 / 89	100 / 100	81 / 89	20 / 19	0
stats	100 / 100	73 / 61	100 / 100	73 / 61	99 / 98	54 / 52	3 / 1	0
histogram	100 / 100	63 / 43	100 / 100	63 / 43	100 / 100	63 / 43	96 / 91	0
gonameparts	100 / 100	71 / 57	100 / 100	71 / 57	100 / 100	29 / 34	29 / 14	0
checkdigit	100 / 100	86 / 85	100 / 100	86 / 85	100 / 100	76 / 74	21 / 14	0
Average	98 / 97	73 / 67	98 / 96	72 / 66	98 / 95	62 / 60	27 / 20	0

- **geo** [11]: a Go library for spherical geometry maintained by the Golang team. We translate a sub-module of this project that implements core algorithms such as edge intersection checking and vertex containment checking
- **ach** [9]: a Go library implementing a reader, writer, and validator for banking operations. We translate a sub-module of this project that implements validation logic.
- **go-edlib** [15]: a Go library for string comparison and edit distance algorithms
- **stats** [13]: a Go library implementing statistical algorithms
- **textrank** [17]: a TextRank implementation in Golang with extendable features (summarization, phrase extraction) and multi-threading
- **histogram** [14]: a Go library implementing streaming approximate histograms
- **gonameparts** [16]: provides string algorithms for parsing human names into parts
- **checkdigit** [10]: provides check digit algorithms and calculators

Details about our benchmarks are given in Table 1. For each benchmark, we report the total lines of code (excluding unit tests), the number of functions and methods defined in the project, the number of structs defined, the statement and function coverage achieved by the project's unit tests, and the number of stars and forks on GitHub.

7.1.4 ***Oxidizer*** *Parameters.* In general, we set *requery_budget* and both *max_tries* parameters high enough to the point that we reach diminishing returns (i.e., setting them higher would rarely yield better results). Specifically, we set *requery_budget* to 10, *max_tries* for Algorithm 2 to 15, and *max_tries* for Algorithm 3 to 5. We use a relatively low temperature (i.e., less random) of 0.2 for Claude 3 Sonnet to produce more reliable and deterministic results.

7.2 Results

How effective is ***Oxidizer*** *at translating entire projects?* We run **Oxidizer** on each of our benchmarks, and report results in Table 2 under the column **Full**. We report two key metrics: compilation success rate under the column **% Compiled** and I/O equivalence rate under the column **% Equiv.**. We report success rates based on two measures: the percentage of compilable/equivalent functions (denoted as **Func.**) and the proportion of compilable/equivalent lines of code, computed as $100 \times \frac{\textit{successful lines of code}}{\textit{total lines of code}}$ (denoted as **LoC.**). Importantly, the equivalence rate depends on the coverage of the test suite: functions not covered by unit tests are automatically marked as not equivalent. For example, in *histogram*, one-third of the functions did not have unit test coverage, preventing us from validating their equivalence to the translated versions.

Our results show that **Oxidizer** produces code that compiles nearly 100% of the time, which is a substantial challenge itself given the restrictiveness of Rust's type system. In addition, **Oxidizer** raises the bar in producing semantically correct code. On average 73% of functions are I/O equivalent to the source project. Notably, mocked functions are treated as both compilation failures and non-equivalent to the original function, so they do not contribute to the reported success rates – only functions that failed to compile are mocked. In Table 2, five out of eight benchmarks do not have any mocked functions. For the other three benchmarks, 4% of functions in **ach** (17 out of 369, totalling 516 LoC), 3% in **textrank** (1 out of 69, totalling 65 LoC) and 5.5% in **geo** (43 out of 780, totalling 1236 LoC) were mocked.

This shows that **Oxidizer** can significantly reduce developer effort to translate a project. Prior work [29] has shown that even a translation where only ~45% of functions are I/O equivalent significantly reduces the amount of developer effort to translate a project.

How much do our proposed type-compatibility checks and feature mapping rules benefit translation? We run **Oxidizer** with type-compatibility checks disabled (but with feature mapping enabled), and with both type-compatibility checks and feature mapping disabled. The compilation success rates and I/O equivalence rates are reported under the columns **No Type Check** and **No Feature Mapping**, respectively, in Table 2.

Our results shows that our type-compatibility checks are generally helpful. They improve function equivalence rate for five out of our seven benchmarks, usually by 5–35%, and in one case by 144%. In addition, our feature mapping rules are absolutely critical for reliability. Without them, Algorithm 2 aborts for all of our benchmarks, preventing us from reliably evaluating I/O equivalence. The sub-column **% Compiled** under the super-column **No Feature Mapping** reports the percentage of functions that were successfully translated before Algorithm 2 aborted.

How much does our proposed semantics-driven phase benefit translation? We run **Oxidizer** with the semantics-driven translation phase disabled and report the compilation success rates and I/O equivalence rates under the column **No Semantics-Driven** in Table 2. Our results show that the semantics-driven phase has limited ability to help translation. In five out of eight benchmarks, the semantics-driven phase does not repair any non-equivalent functions. For the remaining three, it is able to repair 15 non-equivalent functions in *geo*, 4 in *ach* and 1 in *textrank*. When considering the cases where the semantics-driven repair does succeed, it takes an average of 2.65 queries to repair a non-equivalent function.

We hypothesize that the limited success of the semantics-driven repair stems from the fact that our feature mapping rules and type compatibility checks already produce high-quality translations that are difficult to improve upon. When such repairs are successful, they typically address straightforward expression-level issues, such as missing signs in numeric values. By contrast, unsuccessful cases are often due to incomplete unit test coverage or program non-determinism, as further discussed in Section 7.3.

*How does **Oxidizer** compare to parallel works in whole-repository translation?* To the best of our knowledge, only two works [29, 53], done in parallel to ours, tackle whole-project translation.

Neither of these tools support Go as source language, and only one [53] supports Rust as a target language, thus directly running their tools on our benchmarks would be impractical. However, certain aspects of the results reported in these works can be directly compared to ours, and we believe our work compares favorably to them.

Both sets of authors report very high success rates (100% and 99%) of getting "runnable" (i.e. syntactically valid and/or compiling) translations, which is similar to **Oxidizer** at 98%. However, **Oxidizer** is much more successful at producing validated I/O-equivalent translations. The first work [53] is unable to validate semantic correctness for the vast majority of the translated code because the test cases crash. This is mostly because they do not attempt to robustly handle semantic correctness. The second work [29] does attempt to robustly handle I/O equivalence. They report that they can validate I/O equivalence of 26% of the functions they translate on average (corresponding to 46% of the functions actually covered by unit tests), which is substantially less than our 73%. Moreover, they report that semantic validation simply crashes 25% of the time. We believe the difference in performance can be credited to our type-compatibility checks, which ensure that semantic validation does not crash, allowing us to successfully validate many more functions.

7.3 Discussion

Remaining non-compilable code. **Oxidizer** predominantly generates compilable translations. For the remaining non-compilable functions, we found that the LLM often struggles to generate code that requires deviations from the original syntax. However, this is sometimes necessary in order to satisfy Rust's strict compiler requirements. For instance, the LLM frequently translates use-after-move patterns, such as **f(x); y = x.field.Clone()**, directly into Rust equivalents, leading to compilation errors. A compilable translation would require modifying the code to clone **x.field** first and then call **f**, deviating from strict syntax similarity. We observed that these syntax-altering patterns accounted for 53% of the non-compilable functions in *ach*.

Remaining inequivalent code. **Oxidizer** achieves a high I/O equivalence rate. For the remaining inequivalent functions, our manual investigation identified semantic errors such as calling incorrect functions or using inappropriate operators. Additionally, some functions exhibit nondeterministic behavior, making equivalence validation particularly challenging. For example, in *stats*, there is a code snippet **perm := rand.Perm(length)** that generates a random permutation of the slice $[0..length)$. While **Oxidizer** correctly translates this code, we were unable to validate equivalence to the original code due to the inherent randomness.

Applicability of the approach to other languages. We believe our approach can be extended to support languages such as C and C++. Since both Go and C handle errors using error codes or return values, our feature mapping rule for error handling—which converts error return values into idiomatic Rust-style error handling using the Result enumeration—would also be beneficial for LLM-based translation from C. Similarly, for C++, feature mapping can be used to translate certain templated classes into Rust using traits and structs, much like our approach to interface translation in Go. This would allow modular translation of methods while preserving subtyping relationships. Furthermore, given the availability of serialization libraries for C/C++, type compatibility is also relevant as it could help filter out incorrect translations.

However, C and C++ introduce additional complexities compared to Go, particularly due to unsafe constructs. We believe that additional program analyses can mitigate these challenges by recovering implicit type information, which can then be incorporated into the prompt. For example, while C arrays are represented as raw pointers, they do not directly map to Rust's **Vec** type. In such cases, fatness analysis can help distinguish array pointers from non-array pointers (as discussed in [62]), enabling more precise translations.

8 Related Work

In this section, we discuss closely related work from the literature under several categories.

Entire Project Translation. The most closely related work to ours are two parallel works on translating entire projects using LLMs [29, 53]. The first work [29] targets translating Java to Python. They propose a partitioning technique and I/O equivalence validation technique that is similar to ours. They also propose to build a symbolic rule-based mapping for APIs in the source language to APIs in the target language, which is somewhat similar to our feature mapping, though we argue it is less flexible than ours. In addition, their technique for I/O equivalence validation is less reliable than ours, as they often report failures to map concrete values in the source language to the target language. The second work [53] does not propose technique for semantic validation at all. Their translation approach resembles ours, but without feature mapping or type compatibility checks. The only other works known to us on entire project translation are based purely on symbolic rules. They target C to Rust, [2, 24, 62], C to Go [1], and Java to C# [12]. However, purely symbolic rule based translation is known for producing unidiomatic code [25].

Other Code Translation Works. The vast majority of other work on code translation [23, 31, 32, 51, 52, 54–56, 59–61] has focused on translating code taken from competitive programming websites [37, 50], educational websites [19, 59], or hand-crafted coding problems [21, 36]. There are a few exceptions [25, 44, 63], but they report very little success on translating code exceeding 100 lines of code. Many of these works [31, 51, 52, 54, 56] propose novel training methodologies for LLMs, which could be applied to the underlying LLM used by our approach to potentially improve performance. Others propose prompting techniques [55] and repair techniques [61], which could be applied to our approach as well. Also relevant to our work are those on automated program repair, which can be leveraged to repair I/O equivalence errors. Several recent works [35, 58] use LLMs to propose repairs, though their techniques would likely need to be adapted to repair translation errors.

Cross Language Differential Testing and Verification. While prior work has proposed techniques for cross-language differential testing [25, 29] and verification [26, 60], they do not address the critical challenge ensuring compatibility between the implementations they compare. We aim to do this with our type-compatibility checks. We drew inspiration from work in language interoperability [45], which targets the case when both languages are compiled to a shared intermediate or target language. Also closely related to code translation is translation validation [39, 49], which is most often used to validate compiler correctness. There are also many other works that focus on same-language differential testing and verification, such as those that use symbolic execution [20, 41, 43, 47] and fuzzing [28, 33, 40, 48]. We could potentially adapt these techniques to obtain stronger guarantees on I/O equivalence, but they are not directly applicable.

9 Conclusion and Data Availability

In this work, we present an approach for translating entire projects, and we demonstrate its application to the translation of Go projects to Rust. We propose two novel strategies, namely feature mapping rules and type compatibility checks, and show empirically on Go projects from Github that they improve the reliability of obtaining a compiling and I/O equivalent translation significantly. This work is the first to deliver translations of entire projects that not only compile but also pass a meaningful share of the projects' test suites. An artifact of our work is publicly available [46].

Acknowledgements

Cristina David was supported by the Royal Society University Research Fellowship URF\R\221031.

References

[1] [n. d.]. C to Go Translator. https://github.com/gotranspile/cxgo.

[2] [n. d.]. C2Rust Transpiler. https://c2rust.com/.

[3] [n. d.]. Claude. https://www.anthropic.com/index/introducing-claude.

[4] [n. d.]. Cloud SDK: Libraries and Command Line Interface. https://cloud.google.com/sdk/. Accessed: 2024-11-05.

[5] [n. d.]. Download Azure SDKs and Tools. https://azure.microsoft.com/en-us/downloads/. Accessed: 2024-11-05.

[6] [n. d.]. Eliminating Memory Safety Vulnerabilities Once and For All (DARPA). https://www.darpa.mil/news-events/2024-07-31a.

[7] [n. d.]. Gemini. https://blog.google/technology/ai/google-gemini-ai/.

[8] [n. d.]. Go standard regular expression library. https://pkg.go.dev/regexp.

[9] [n. d.]. Moov ACH. https://github.com/moov-io/ach.

[10] [n. d.]. Provide check digit algorithms and calculators written in Go. https://github.com/osamingo/checkdigit.

[11] [n. d.]. S2 geometry library in Go. https://github.com/golang/geo.

[12] [n. d.]. Sharpen – Automated Java->C# coversion. https://github.com/mono/sharpen.

[13] [n. d.]. Stats – Golang Statistics Package. https://github.com/montanaflynn/stats.

[14] [n. d.]. Streaming approximate histograms in Go. https://github.com/VividCortex/gohistogram.

[15] [n. d.]. String comparison and edit distance algorithms library. https://github.com/hbollon/go-edlib.

[16] [n. d.]. Takes a full name and splits it into individual name parts. https://github.com/polera/gonameparts.

[17] [n. d.]. TextRank implementation in Golang with extendable features (summarization, phrase extraction) and multi-threading (goroutine). https://github.com/DavidBelicza/TextRank/tree/master.

[18] [n. d.]. Tools to Build on AWS. https://aws.amazon.com/developer/tools/. Accessed: 2024-11-05.

[19] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2023. AVATAR: A Parallel Corpus for Java-Python Program Translation. In *Findings of the Association for Computational Linguistics: ACL 2023*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 2268–2281. doi:10.18653/v1/2023.findings-acl.143

[20] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. 2013. Regression tests to expose change interaction errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 334–344. doi:10.1145/2491411.2491430

[21] 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. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[22] Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, Rishi Poddar, and Aseem Rastogi. 2025. RustAssistant: Using LLMs to Fix Compilation Errors in Rust Code. In *International Conference on Software Engineering (ICSE)*. IEEE, 267–279. doi:10.1109/ICSE55347.2025.00022

[23] Peng Di, Jianguo Li, Hang Yu, Wei Jiang, Wenting Cai, Yang Cao, Chaoyu Chen, Dajun Chen, Hongwei Chen, Liang Chen, Gang Fan, Jie Gong, Zi Gong, Wen Hu, Tingting Guo, Zhichao Lei, Ting Li, Zheng Li, Ming Liang, Cong Liao, Bingchang Liu, Jiachen Liu, Zhiwei Liu, Shaojun Lu, Min Shen, Guangpei Wang, Huan Wang, Zhi Wang, Zhaogui Xu, Jiawei Yang, Qing Ye, Gehao Zhang, Yu Zhang, Zelin Zhao, Xunjin Zheng, Hailian Zhou, Lifu Zhu, and Xianying Zhu. 2024. CodeFuse-13B: A Pretrained Multi-lingual Code Large Language Model. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24)*. ACM, 418–429. doi:10.1145/3639477.3639719

[24] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating C to safer Rust. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 121 (Oct. 2021), 29 pages. doi:10.1145/3485498

[25] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2024. Towards Translating Real-World Code with LLMs: A Study of Translating to Rust. arXiv:2405.11514 [cs.SE] https://arxiv.org/abs/2405.11514

[26] Jack J. Garzella, Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. 2020. Leveraging Compiler Intermediate Representation for Multi- and Cross-Language Verification. In *Verification, Model Checking, and Abstract Interpretation: 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16–21, 2020, Proceedings* (New Orleans, LA, USA). Springer-Verlag, Berlin, Heidelberg, 90–111. doi:10.1007/978-3-030-39322-9_5

[27] Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. 2020. Featherweight Go. arXiv:2005.11710 [cs.PL] https://arxiv.org/abs/2005.11710

[28] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. DLFuzz: differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 739–743. doi:10.1145/3236024.3264835

[29] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Repository-Level Compositional Code Translation and Validation. arXiv:2410.24117 [cs.SE] https://arxiv.org/abs/2410.24117

[30] Suman Jain and Inderveer Chana. 2015. Modernization of Legacy Systems: A Generalised Roadmap. In *International Conference on Computer and Communication Technology (ICCCT '15)*. ACM, 62–67. doi:10.1145/2818567.2818579

[31] Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. 2023. Attention, Compilation, and Solver-based Symbolic Analysis are All You Need. *arXiv preprint arXiv:2306.06755* (2023).

[32] Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. 2023. On the Evaluation of Neural Code Translation: Taxonomy and Benchmark. In *Automated Software Engineering (ASE)*. IEEE, 1529–1541. doi:10.1109/ASE56229.2023.00114

[33] Wei Jin, Alessandro Orso, and Tao Xie. 2010. Automated Behavioral Regression Testing. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST '10)*. IEEE Computer Society, USA, 137–146. doi:10.1109/ICST.2010.64

[34] Ravi Khadka, Belfrit V. Batlajery, Amir M. Saeidi, Slinger Jansen, and Jurriaan Hage. 2014. How do professionals perceive legacy systems and software modernization?. In *International Conference on Software Engineering (ICSE 2014)*. ACM, 36–47. doi:10.1145/2568225.2568318

[35] Jiaolong Kong, Xiaofei Xie, Mingfei Cheng, Shangqing Liu, Xiaoning Du, and Qi Guo. 2025. ContrastRepair: Enhancing Conversation-Based Automated Program Repair via Contrastive Test Case Pairs. *ACM Trans. Softw. Eng. Methodol.* (March 2025). doi:10.1145/3719345 Just Accepted.

[36] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 943, 15 pages.

[37] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, Vol. 1. https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/c16a5320fa475530d9583c34fd356ef5-Paper-round1.pdf

[38] Jacob Matthews and Robert Bruce Findler. 2007. Operational semantics for multi-language programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, 3–10. doi:10.1145/1190216.1190220

[39] George C. Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 83–94. doi:10.1145/349299.349314

[40] Shirin Nilizadeh, Yannic Noller, and Corina S. Păsăreanu. 2019. DifFuzz: differential fuzzing for side-channel analysis. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 176–187. doi:10.1109/ICSE.2019.00034

[41] Yannic Noller, Corina S. Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. 2020. HyDiff: hybrid differential software analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1273–1285. doi:10.1145/3377811.3380363

[42] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei

Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[43] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a doubt: testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 1181–1192. doi:10.1145/2884781.2884845

[44] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 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* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 82, 13 pages. doi:10.1145/3597503.3639226

[45] Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. 2022. Semantic soundness for language interoperability. In *Programming Language Design and Implementation (PLDI 2022)*. ACM, 609–624. doi:10.1145/3519939.3523703

[46] Brandon Paulsen. 2025. *Artifact for Scalable, Validated Code Translation of Entire Projects using Large Language Models*. doi:10.5281/zenodo.15242640

[47] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed incremental symbolic execution. *SIGPLAN Not.* 46, 6 (June 2011), 504–515. doi:10.1145/1993316.1993558

[48] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. NEZHA: Efficient Domain-Independent Differential Testing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 615–632. doi:10.1109/SP.2017.27

[49] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems (LNCS, Vol. 1384)*. Springer, 151–166. doi:10.1007/BFB0054170

[50] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. arXiv:2105.12655 [cs.SE] https://arxiv.org/abs/2105.12655

[51] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1730, 11 pages.

[52] Baptiste Roziere, Jie M. Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging Automated Unit Tests for Unsupervised Code Translation. arXiv:2110.06773 [cs.SE] https://arxiv.org/abs/2110.06773

[53] Momoko Shiraishi and Takahiro Shinagawa. 2024. Context-aware Code Segmentation for C-to-Rust Translation using Large Language Models. arXiv:2409.10506 [cs.SE] https://arxiv.org/abs/2409.10506

[54] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2023. Code Translation with Compiler Representations. arXiv:2207.03578 [cs.PL] https://arxiv.org/abs/2207.03578

[55] Zilu Tang, Mayank Agarwal, Alexander Shypula, Bailin Wang, Derry Wijaya, Jie Chen, and Yoon Kim. 2023. Explain-then-translate: an analysis on improving program translation with self-generated explanations. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. Association for Computational Linguistics, 1741–1788. doi:10.18653/v1/2023.findings-emnlp.119

[56] Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. 2024. StructCoder: Structure-Aware Transformer for Code Generation. *ACM Trans. Knowl. Discov. Data* 18, 3, Article 70 (Jan. 2024), 20 pages. doi:10.1145/3636430

[57] David Tolnay. 2024. Anyhow. https://github.com/dtolnay/anyhow.

[58] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1482–1494. doi:10.1109/ICSE48619.2023.00129

[59] Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. CodeTransOcean: A Comprehensive Multilingual Benchmark for Code Translation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 5067–5089. doi:10.18653/v1/2023.findings-emnlp.337

[60] Aidan Z. H. Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. 2024. VERT: Verified Equivalent Rust Transpilation with Large Language Models as Few-Shot Learners. arXiv:2404.18852 [cs.PL] https://arxiv.org/abs/2404.18852

[61] Xin Yin, Chao Ni, Tien N. Nguyen, Shaohua Wang, and Xiaohu Yang. 2024. Rectifier: Code Translation with Corrector via LLMs. *CoRR* abs/2407.07472 (2024). doi:10.48550/ARXIV.2407.07472 arXiv:2407.07472

[62] Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang. 2023. Ownership Guided C to Rust Translation. In *Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part III* (Paris, France). Springer-Verlag, Berlin, Heidelberg, 459–482. doi:10.1007/978-3-031-37709-9_22

[63] Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. Multilingual Code Co-evolution using Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 695–707. doi:10.1145/3611643.3616350

Received 2024-11-15; accepted 2025-03-06