SolMover: Smart Contract Code Translation Based on Concepts*

Rabimba Karanjai
University of Houston
Houston, USA
rkaranja@cougarnet.uh.edu

Lei Xu
Kent State University
Kent State, USA
xuleimath@gmail.com

Weidong Shi
University of Houston
Houston, USA
wshi3@uh.edu

ABSTRACT

Large language models (LLMs) have showcased remarkable skills, rivaling or even exceeding human intelligence in certain areas. Their proficiency in translation is notable, as they may replicate the nuanced, preparatory steps of human translators for high-quality outcomes. Although there have been some notable work exploring using LLMs for code-to-code translation, there has not been one for smart contracts, especially when the target language is unseen to the LLMs. In this work, we introduce our novel framework `SolMover`, which consists of two different LLMs working in tandem in a framework to understand coding concepts and then use that to translate code to an unseen language. We explore the human-like learning capability of LLMs with a detailed evaluation of the methodology to translate existing smart contracts written in `Solidity` to a low-resource one called `Move`. Specifically, we enable one LLM to understand coding rules for the new language to generate a planning task, for the second LLM to follow, which does not have planning capability but does have coding. Experiments show that `SolMover` brings a significant improvement over gpt-3.5-turbo-1106 and outperforms both Palm2 and Mixtral-8x7B-Instruct. Our further analysis shows that employing our bug mitigation technique even without the framework still improves code quality for all models.

CCS CONCEPTS

• **Computing methodologies** → **Machine translation**; **Natural language generation**.

KEYWORDS

Smart Contracts, Machine Learning, Machine Translation, Code Transpilation, LLM, Large Language Model

ACM Reference Format:
Rabimba Karanjai, Lei Xu, and Weidong Shi. 2024. SolMover: Smart Contract Code Translation Based on Concepts. In *Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware '24), July 15–16, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3664646.3664771

*Part of this work was conducted with the generous grant support from Sui Foundation Academic Research Grant

AIware '24, July 15–16, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0685-1/24/07…$15.00
https://doi.org/10.1145/3664646.3664771

1 INTRODUCTION

Recent advancements in Large Language Models (LLMs) have been transformative, marking significant progress towards being human-like. These models are being touted to have human-level intelligence, especially in comprehending and generating natural language [28, 44, 54, 75]. Natural language translation is one domain where LLMs excel, showcasing notable proficiency and effectiveness [22, 37, 48, 71, 76]. This advancement resonates with the initial goals and visions set forth by machine translation researchers in the 1960s [25, 51], raising the question of whether LLMs can mirror the translation methodologies employed by humans. This also begs the question: can we take inspiration from how these translations work to explore code-to-code translation using LLMs?

In recent times, the proliferation of decentralized ledger technologies, coupled with the smart contracts built upon them, has been particularly notable within the finance industry. For example, it has been reported that in the year 2021, Uniswap's smart contracts facilitated transactions averaging a daily volume of approximately $7.17 billion [26]. Given the surge in smart contract utilization and their pivotal role in diverse applications such as Confidential Computing [39, 60] and Decentralized Serverless Architectures [42], it becomes imperative to inquire into the efficacy of Large Language Models (LLMs) in authoring smart contracts derived from user directives, as well as the security robustness of such automatically generated contracts. So it becomes a natural question if we can take a smart contract in one language and use LLMs to translate it.

Smart contract languages are unique domain specific programming languages. Different blockchains support different languages, which means there's no one-language-fits-all solution for smart contract developers. The most popular one is Ethereum `Solidity`. Other smart contract languages include Vyper, Cairo, Move, Yul and Yul+, Clarity, to name just few. The domain specific nature of smart contract languages creates unique challenges for LLM based code translation and makes it particularly interesting as a research topic. Comparing with other general purpose programming languages, LLM based translation of smart contract languages is less studied by the research community.

In this work, we delve into the capabilities and constraints of LLMs within the realm of smart contract code transpilation. We investigate whether LLMs can mimic human translators in their approach to translate smart contracts — a specialized yet increasingly essential subset of software code. The primary focus of this work is to explore whether we can use LLMs for smart contract translation with specific correctness guarantees. Specifically, we aim to investigate whether LLMs can effectively understand a smart contract written in `Solidity`, and generate equivalent code in another smart contract language, namely Move in our case. We also look at how we can encode the smart contract concepts into the

model for a more universal code transpilation framework that can translate one coding language to another in a universal way.

To address the above, we propose our system SolMover, which is a wordplay on Solidity to Move. SolMover involves multi-step knowledge distillation to understand the solidity code and generate equivalent Move code. We convert the translation problem to a multi-step code synthesis problem with a granular task-based approach. SolMover first parses a Solidity file for functions and content, generating a high-level task based on the file. This task is then sent to the first part of our fine-tuned LLM to generate sub-tasks. The fine-tuned LLM is fine-tuned on textbooks of Move language and specifications along with Solidity. This generates sub-tasks based on the previous input. This then we pass on to a smaller codegen model trained specifically on a very small subset of Move code, which generates the translated equivalent Move code for us. We use Move Prover for verification of the generated code, as well as multi-prompting techniques to pass on the error message from move-cli to further improve the quality of the generated code. We validate our approach based on our evaluation set, as well as analyzing if the translated code has matched with the input of the specifications.

This work performs a multifaceted exploration of smart-contract translation of a low-resource language (Move) through the use of a combination of concept mining along with error-code-guided prompt engineering for regeneration. We also perform a preliminary evaluation with state-of-the-art models for the same code translation work. Our study aims to answer the following research questions:

- **RQ1: Can the LLMs learn coding concepts?** We try to encode the knowledge of the Move language in our first LLM. Are the Large Language Models capable of learning programmatic rules from textbooks? Can they associate the semantic rules described in a text with actual coding?
- **RQ2: Can the concepts be used to generate a granular sub-task from a generalized prompt?** We explore the possibility of generating task-specific granular sub-tasks based on the inferred prompt from the input contract using knowledge from **RQ1**. Are these sub-tasks useful enough for generating code?
- **RQ3: Can we generate code in a low resource language the LLMs have not been trained on?** We ask the question of our fine-tuned model, which has not been trained on a large plethora of Move code, whether we can still generate compilable Move code.
- **RQ4: Can we mitigate bugs using compiler feedback?** Can we use compiler feedback to mitigate the bugs and make the code better? To what extent do prompting techniques help the model mitigate these bugs?

Move [27] was designed to work on the Libra Blockchain [45, 46], and its unique feature is crafting unique asset types (called resources) with built-in self-defense [33]. Duplication and silent disposal are out of the question; these assets can only change hands between storage spaces within a Move program. Move's rigorous type system acts as a bouncer, which checks IDs and verifies every transfer. Even with these robust shields, resources remain flexible citizens in Move code, i.e., they can nestle in data structures, act as arguments, and join the party wherever needed. In contrast to other programming languages, Move is not a completely separate language created from scratch, but developed based on Rust [52] with different built-in safety features. These security features make it even more complex to translate smart contracts written in Solidity to Move. In summary, some of the major contributions of this work are listed below:

- We propose SolMover, which takes Solidity code and translates it to Move.
- We demonstrate that even if an LLM has not been explicitly pre-trained on a code corpus, and is unable to generate compilable code, it still can learn to generate compilable code with fine-tuning and the assistance of our framework.
- We demonstrate that LLMs can encode concepts, and generate granular sub-tasks from a larger task.
- We find that using compiler errors to guide the LLMs to produce better code does produce results, but to a limited degree.
- We demonstrate that with a combination of knowledge distillation and fine-tuning using a very small amount of code, code generation for low-resource language is possible.
- Our study and approach may be generalizable to LLM based translation of other domain specific and/or low-resource languages.

To the best of our knowledge, we are the first to propose and evaluate a system to:

- use concepts and code to generate code; and
- show that it is indeed possible to generate compilable code for low-resource language using off-the-shelf LLMs, just by fine-tuning and daisy chaining.

2 THE SOLMOVER FRAMEWORK

As depicted in Figure 1, SolMover consists of multiple modular blocks. In *block 1* the Solidity smart contract given for translation is analyzed. We use a technique similar to Karanjai et al [40] to extract the function and comments from the Solidity file. The module parser and comment extractor work in tandem to extract comments from the Solidity code and use a prompt template similar to [40] to generate the initial Prompt for the task. *Block 2* consists of the LLM tasker, which uses Move Books, white papers, and tutorials (with code examples) as its source for Retrieval Augmented Search to generate sub-tasks based on the prompt generated in *Block 1*. Each of these sub-tasks is then sent to our second LLM (SolMover) fine-tuned on limited Move code [19]. These sub-tasks are then collated to make the final Move code to be part of *Block 4*. *Block 3* is the second LLM which takes each of these sub-tasks to generate the segmented codes. These code fragments are then stitched to generate the complete code candidates for the translation. In *Block 4* we use the move-cli to try to compile the translated code. This step logs the successful compilation. If a code does not compile, it logs the error message and sends it back to SolMover again to compile. SolMover tries to do it five times and logs the translation as a failure after the fifth try. *Block 5* tries to take the compilable code and verify correctness by running it through Move Prover [78]. Move Prover tries to verify the contract formally proving that the smart contract is correct within its specification. In *Block 5* we log the incorrectness specification and again pass it to SolMover to rectify the error by re-prompting with the error. This is done again five times. If the correctness holds, the program is marked as a safe

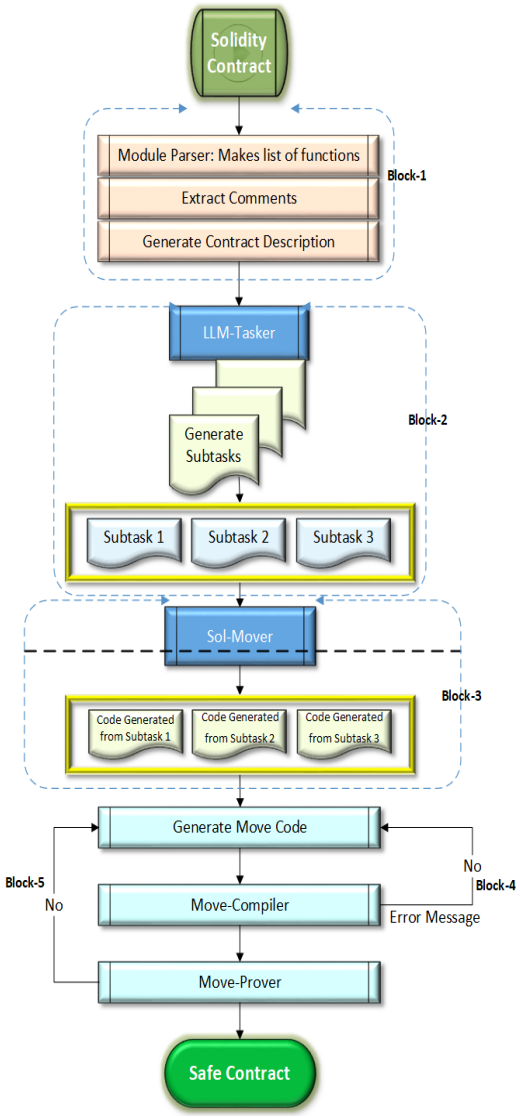

Figure 1: Framework for SolMover consists of five stages: (1) Task Creation: The parser parses the Solidity file to generate the initial prompt for the task based on comments and keywords used in the smart contract. (2) Concept Mining: We use retrieval-augmented approach to mine concepts from Move programming guides and books to generate sub-tasks based on the initial prompt. (3) Code Generation: We use the sub-tasks generated using concept distillation to generate code for each sub-task. These are then compiled into a single Move file, which will be our candidate-translated code. (4) The translated code is compiled, if it cannot be compiled, the error is sent back to SolMover with a modified prompt incorporating the error for regeneration. (5) The compiled code is passed through Move Prover to see if it can be proved, if it cannot be it again is sent to SolMover with the error message incorporated in the prompt for regeneration.

contract. Otherwise it is marked as compilable but not yet formally proved.

2.1 Code Understanding

Since we have converted our code translation work to a code generation task, we first need to understand what the original input code is doing. To do that, we developed a parser that takes a `Solidity` file and extracts comments, functions, and generates a seed candidate prompt from the `Solidity` input file that is used for constructing the prompt for the generator to use.

Comments are essential for understanding what a `Solidity` smart contract is supposed to do. Explanatory comments embedded within the code act as beacons, revealing its core logic and central purpose, which are essential to understanding the coder's original intentions and ensuring the code's accurate execution. By parsing the `Solidity` code using techniques similar to [41].

Topic refers to what the Solidity code does. Mostly these are derived from the initial comment and the used functions, as we can see in the code snippet in Listing 1.

```
pragma solidity ^0.8.0;

// The sample hotel and vending Solidity smart contract
     below allows one to rent a hotel room.
// It allows someone to make a payment for a room if the
     room is vacant. After payment is made to the
     contract the funds are sent to the owner.
contract HotelRoom {
    //create an emun with 2 status so we can keep track
         of our hotel room
    enum Statuses {
        Vacant,
        Occupied
    }
    Statuses currentStatus;
```

Listing 1: Input Solidity Code Example

2.2 Concept Mining

We take cues from how human translation works [32], the concept mining requires the LLMs to first produce output that aligns with those concepts and knowledge beneficial for the transpilation work.

The concept mining involves fine-tuning an LLM on textbook data of `Move`. We follow Gunasekar et al [34] to prepare the dataset and to fine-tune the model, and [49, 77] for finding relevant text. We prepare the dataset as shown in Figure 2, where we use `Move` Books [3, 12], blogs on `Move` [11], tutorials with code snippets [13] as concepts and use code samples from the repository given in [18]. We intentionally do not use any code corpus for fine-tuning the model. For preparing the dataset we use scripts from [10] modified to work for our files.

We use Retrieval Augmented search in our architecture as described in [77] for finding out our concepts. Since our task is different than their task, our implementation is different.

We operate under the assumption of a comprehensive textbook database containing diverse text and code samples. This database is the source for D text snippets generated via document segmentation. Segmentation leverages the HTML structure, specifically **H** or **Heading** tags (or, absent these, larger text elements) and their associated text blocks. This technique extracts concept headings

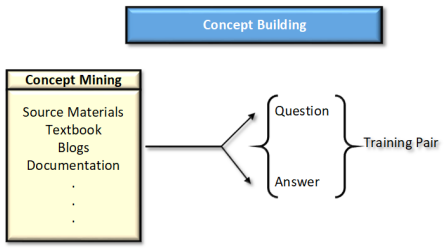

Figure 2: Concept mining. The left block signifies the source materials mined for building the plan. These are processed to create (Question, Answer) pairs, used as instruction prompts for training.

and related explanatory content. For optimized retrieval, we apply the Dense Passage Retriever (DPR) [43] to subdivide each text segment into units of consistent length. This aligns with DPR best practices for retrieval quality. A key advantage of this method is its scalability, facilitating the management of very large file sizes. The outcome is a retrieval database $C = \{c_1, c_2, ..., c_M\}$ composed of M text fragments.

We define the query keywords as a set $X = \{x_1, x_2, ..., x_k\}$. A retrieval function, denoted as $\mathbf{R} : (X, C) \rightarrow C$ operates on the query (X) and the retrieval database (C). This function identifies and returns the most similar text fragment, c_s within C. Subsequently, a generative model, **G** takes the retrieved fragment c_s and the query context (X) as input. In the context of token-level text prediction, where the predicted sequence length is restricted to a single token $n = 1$, **G** predicts the subsequent token(s), denoted as $Y = \{x_{k+1}, ..., x_{k+n}\}$. Formally, we have $P(Y) = \prod_{i=1}^{n} P(x_{k+i}|c_s, x_{1:k+i-1})$.

For document retrieval, we employ the BM25 algorithm [61], similar to its usage in ElasticSearch. BM25 is a term-based approach that leverages a bag-of-words representation. It calculates lexical similarity between the query and document fragments. In contrast, DPR models [43] utilize two bidirectional transformers, E_C and E_Q. E_C encodes each segmented text fragment within the retrieval database C and generates an index. The model extracts the representation of the special token `[CLS]` which is then used for similarity computation: $sim(q, c) = E_C(c)^T E_Q(q)$. The training process follows the approach outlined in [43, 49], incorporating batch negatives for loss calculation as described in [56]. However, we do not ignore the hard negatives like [49], and instead adopt the following loss function suggested in [43].

$$L(q, c^+, c_1^-, c_2^-, ..., c_m^-) = -log \frac{e^{sim(q,c^+)}}{e^{sim(q,c^+)} + \sum_{i=1}^{m} e^{sim(q,c_i^-)}}$$

2.3 Concept Selection

Concept Selection is the final step for our sub-task generation. Even though keywords, topics, and relevant text demonstration are beneficial for sub-task generation for our translation task, not all results will be useful. For example, the LLM may generate sub-tasks based on trivial or noisy content which will in turn guide the `SolMover` LLM to generate useless unused code snippets in the best case, and wrong code in the worst case. Hence we limit the

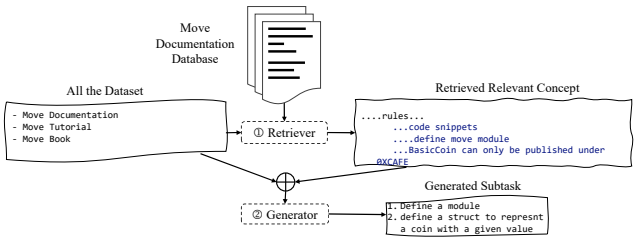

Figure 3: Concept selection.

sub-task generation based on concepts gleaned only from the text parts where suitable code snippets have also been found. Which is also passed to the `SolMover` as part of the sub-task prompts. This helps the LLM produce sub-tasks as demonstrated in Figure 3.

2.4 `SolMover` Code Generator

`SolMover` is our `Move` Code generator which takes the sub-tasks generated by the previous LLM as seen in Figure 3 and generates code in Move. `SolMover` is an instruction-trained model based on Alpaca [66]. Alpaca is an open-source instruction following LLM fine-tuned on LLAaMA [67] model with 52k Self Instruct data with user-shared conversations collected from ShareGPT [65]. We use LangChain to build a prompting framework that takes each of the sub-tasks along with any code snippet example found and generates code using the fine-tuned model. The generated response then is compiled into a single file based on the task sequence to rebuild the translated code for the original task. This is then sent to move-cli for compilation. All the compilation errors are logged and added as memory to the LangChain agent and sent back to the `SolMover` for program rectification. We run this process iteratively five times. If the candidate's response passes the compilation stage we log it as successful code translation. However, we still do not guarantee its correctness and send it to `Move` Prover, which is an automatic formal verification system for Move. The errors suggested by the prover are again sent to `SolMover` for five iterations for correction. We discuss in detail the training part of both `SolMover` and our sub-task creation LLM in Section 3.

2.5 Dataset for `Solidity` as Input Candidate

In our study, we faced the limitation of not being able to utilize pre-existing datasets like HumanEval [29], which are predominantly focused on Python programming. To the best of our knowledge, we were unable to find any existing benchmark that deals with smart contract translation or move code generation. Consequently, we initiated the creation of a novel dataset for our assessment. Our approach to verifying code accuracy involved a meticulous selection of source files from GitHub, all under open-source licenses with permissive terms.

Our strategy included the development of a scraper via GitHub APIs, aimed at harvesting `Solidity` code from a variety of projects. The corpus collection was guided by multiple criteria:

- Eligible projects must have at least 50 GitHub stars, indicating sufficient interest, popularity, and likelihood of code written for a human audience.

- The presence of "Solidity" as a programming tag was mandatory for further refinement.
- An ample presence of comments in the projects was essential. This facilitates the development of scalable prompts and the establishment of an experimental pipeline.

These projects underwent further processing to remove non-Solidity files, and the results form the foundational dataset for training.

3 TECHNICAL SPECIFICATION

We give details of our training regimen and model selection for both of our models for concept understanding and sub-task generation, as well as fine-tuned Alpaca for code generation.

3.1 Retrieval Augmented LLMs for Concept Understanding

To address one of the prime premises of code-to-code translation for very low-resource languages like Move, our framework solves a key bottleneck, premise selection [69]. Most of the existing LLM-based knowledge distillation [77] and code completion [61] frameworks generate the next tactic (step) based on the current step as input. However generating coherent plans and sub-tasks critically depends on premises, such as rules and code examples from an associated description.

Incorporating all the possible contexts is too large to fit into LLM's input prompt given the limited context window, and circumventing by using LangChain's agent-based memory does not yield much better results. Existing methods must memorize the association between the task state and the next relevant task based on the original task. If the context has been used in the training data to solve a similar problem, then this works, however for low-resource languages like Move that is not the case. On top of that, since Move is based on Rust, the existing pre-trained Rust code pollutes the premise by providing the wrong candidates. It does not generalize for truly novel scenarios, that require code generation based on tasks unseen in training data.

Our solution complements the memorization of LLM with explicit context selection for better concept finding. The context is extracted from books where they are defined and used. It enables us to find relevant context by augmenting LLMs with retrieval.

We also need to limit the context retrieval to a small number of contexts for it to be effective and useful. As discussed in Section 2.2 our retriever builds upon DPR but has the following two changes relevant to our purpose. First, not all context is available while searching. We first extract comments and functions that limit the accessible context based on them as keyword input in the initial prompt. That reduces the retriever's task on our data by almost 75%, simplifying the task. Secondly, DPR benefits from having hard negatives in training, i.e. irrelevant contexts that are difficult to distinguish. We instead use in-file hard negatives which sample negative contexts defined in the prompt itself.

3.2 `SolMover` Code Generation Training

For the code generator, we use Alpaca and fine-tune it using code examples as part of the instruction training dataset created as described in Section 2.4. We used a very limited set of code samples as a corpus for fine tuning. These include Fungible Tokens [6–8, 14, 20], NFT [16, 17], DeFi [5, 9, 15] as an example for creation of the dataset. The function of this fine-tuning is to create an association in the Alpaca model to have code association for the `Move` code.

4 EXPERIMENTS SETUP

4.1 Models

We experiment with four LLMs, encompassing both open-source and closed-source state-of-the-art models.

- gpt-3.5-turbo-1106: Employing the innovative RLHF methodology developed by [58], OpenAI's robust yet closed-source language model offers exceptional capabilities. We interact with this model through the official API provided by OpenAI.
- Alpaca [66]: Building upon the LLaMA model [68], this open-source, instruction-following language model was further honed using a comprehensive dataset of 52,000 examples generated through the Self-Instruct approach [73].
- Mixtral [4]: Mixtral-8x7B-Instruct, a free-to-use language model, excels at following your instructions and unleashing its creative potential. Built upon the powerful Mixtral-8x7B architecture [4], it's been meticulously trained on a custom dataset to understand and respond to diverse prompts and requests. This versatile model can generate text in various formats, translate languages, answer your questions informatively, and even write different kinds of creative content.
- Palm2 [24]: Google's PaLM 2, pushing the boundaries of language understanding, shines in tasks demanding advanced reasoning and expertise. Whether it's grappling with code and math, navigating multilingual nuances, or crafting accurate translations, PaLM 2 excels with its impressive size and cutting-edge techniques.

4.2 Comparative Methods

For our code generation task, we consider only the single-candidate method. Since this is a framework that utilizes a daisy-chained response, only the primary candidate is being considered. Within single candidate method, we consider:

- **Baseline**: Zero-shot smart contract translation with temperature set to 0 (default for remainder of experiments in this paper).
- **5-Shot** [36]: Prompting the model with five exemplars of superior quality, specifically selected from the sub-task domain and prepended to the test input, has been demonstrated to yield optimal overall performance, as per [36]. Further augmenting the number of examples, however, does not appear to produce any appreciable enhancement in outcomes.

4.3 Metrics and Benchmark

Since this work looks at code translation problems for a very low resource language `Move`. There are no existing code generation or testing benchmarks that we can use. Furthermore, the nature of the `Move` smart contract makes it very difficult to train and test separate evaluation dataset available for systematic evaluation. It also makes it less feasible to rely on public code repositories like GitHub for generating exact code examples for comparison, since this is not a code generation but rather a Code Translation Task.

Table 1: Code translation capability of four LLMs.

LLMs	Compilable Code?	Performance
gpt-3.5-turbo-1106	Y	Mixed
SolMover (Two LLM Combined)	Y	Mixed
Mixtral-8x7B-Instruct	N	Mixes different languages and generate unusable code
LLama2	N	Mixes different languages and generate unusable code
Palm2	N	For Move only generates code snippets and plan

For that purpose, we adopt the following criteria to evaluate the effectiveness of our approach: (1) Code compilability of the translated smart contract; (2) Code correctness of the translated smart contract; and (3) Bug mitigation of first candidate translation.

5 RESULTS

We ran the experiments on all four LLMs discussed in Section 4. However, both Palm2 and Mixtral did not produce any tangible output for any of our candidate translations as we can see from Table 1. Henceforth we will only report the results found in our SolMover framework and gpt-3.5-turbo-1106.

5.1 Successful Smart Contract Translation

We ran a total of 734 Solidity contracts from our collected dataset in Section 2.5 through both gpt-3.5-turbo-1106 and SolMover. Each model was allocated an identical quantity of tasks totaling 734. The performance is evaluated based on successful compilation (SC), improvements post-error feedback, and further enhancements after Move Prover feedback. We can observe from Figure 4 that:

- Initial SC rates were 204 for gpt-3.5-turbo-1106 and 313 for SolMover.
- Post error feedback, SC rates improved to 229 for gpt-3.5-turbo-1106 and significantly to 397 for SolMover.
- Subsequent to Move Prover feedback, gpt-3.5-turbo-1106 remained static at 229 SCs, whereas SolMover exhibited a marginal increase to 401 SCs.

In this comparative analysis, the combined language model SolMover exhibited superior performance over gpt-3.5-turbo-1106 in a translation task involving 734 items. SolMover outperformed in successful compilations initially, after error feedback, and after Move Prover feedback, with the latter showing no improvement at the final stage. This indicates that the integration of two language models in SolMover significantly enhances its ability to learn from feedback and improve its output, making it more adept at handling complex code translation tasks.

5.2 Reducing Bugs with Iterative Error Feedback

As we described in Section 2.4 we try to mitigate bugs introduced by the LLMs for code generation by iterative prompting with error code as a guide. This compiler error-based prompting iterative loop runs at a maximum of five times before marking the translation case as unsuccessful. We specifically decided on five runs based on our observation that increasing the loop more, does not improve result, but at times just runs in a loop. We run this experiment on

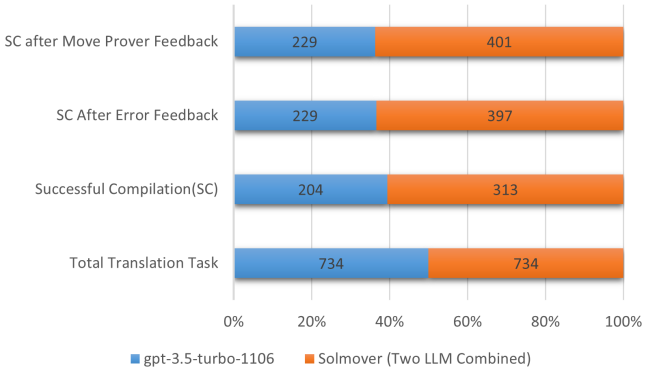

Figure 4: Successful Translations.

Table 2: Bug mitigation.

LLM	Total Translation Task	Incomplete Translation (IC)	IC After Error Feedback
gpt-3.5-turbo-1106	734	187	110
SolMover (Two LLM Combined)	734	201	134

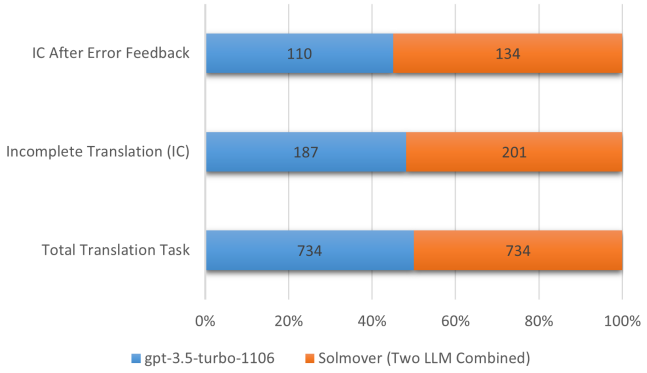

Figure 5: Bug Mitigation.

both of the LLMs to decouple the concept mining part from the bug mitigation part, and to see if just iterative prompting alone can help achieve better performance on state-of-the-art commercial LLMs too. We report the result in the Table 2. The focus here is on incomplete translations (IC) before and after error feedback is applied.

Here we first log the incomplete translations without any kind of compiler feedback. Then we log how both the compiler feedback and Move Prover feedback affect the LLM's bug mitigation capabilities.

As we can see from Figure 5 the results indicate that both models benefited from error feedback, with gpt-3.5-turbo-1106 showing a more substantial reduction in Incomplete Completions. Producing more compilable code after the feedback loop. Despite SolMover starting with more Incomplete Completions(ICs), it did not reduce its ICs as effectively as gpt-3.5-turbo-1106 post-feedback.

In the comparison of incomplete translations for two language models, `gpt-3.5-turbo-1106` and `SolMover` (Two LLM Combined), both began with a similar number of tasks. `gpt-3.5-turbo-1106` initially reported fewer incomplete translations and also showed greater improvement after receiving error feedback. This suggests that while `SolMover` had a higher starting point of incomplete tasks, it was less responsive to feedback in reducing these errors compared to `gpt-3.5-turbo-1106`.

- Pre-feedback, `gpt-3.5-turbo-1106` had 187 ICs, while `SolMover` had slightly more with 201 ICs.
- Post-error feedback, ICs reduced to 110 for `gpt-3.5-turbo-1106` and to 134 for `SolMover`.

5.3 Correctness

Our measure of code correctness here is based on the theorem prover for `Move`, `Move` Prover. When `Move` Prover can prove a contract, we mark it as safe. Otherwise, if it produces an error, we mark it as correctness not proved. We also take the same iterative approach as the compiler error feedback loop to test if that has any effect on the different compilers. As we can observe from Figure 6 that `SolMover` not only had a higher success rate in initial compilations but also showed greater improvement upon receiving feedback. Moreover, it continued to improve even after the `Move` Prover feedback, in contrast to the `gpt-3.5-turbo-1106`, which plateaued.

- The `gpt-3.5-turbo-1106` had a baseline SC of 204, which increased to 229 after error feedback and remained unchanged after `Move` Prover feedback.
- The `SolMover` model, on the other hand, started with a higher SC of 313, which then improved to 397 after error feedback and slightly increased to 401 after `Move` Prover feedback.

The comparative assessment of `gpt-3.5-turbo-1106` and `SolMover` (Two LLM Combined) reveals that the latter outperforms the former in all stages of code compilation. Initially, `SolMover` starts with a higher number of successful compilations. It continues to improve significantly upon receiving error feedback and demonstrates a modest increase even after `Move` Prover feedback, suggesting a robust ability to learn and adapt. In contrast, `gpt-3.5-turbo-1106` exhibits improvement after error feedback but shows no further enhancement post `Move` Prover feedback, indicating a potential limit to its adaptability.

6 DISCUSSION & ANALYSIS

In this section, we conduct analyses to understand the `SolMover` framework. We try to answer the research question (RQ) we started with and analyze the results reported.

6.1 Concept Distillation

One of the unique aspects of the framework is to try to encode concepts into an LLM using retrieval-augmented search methods. As we see from Figure 2, our search method can produce sub-tasks. If we look closely at the results in Figure 6. Contract Correctness and Figure 5. Bug Mitigation, we will notice that the iterative method has been applied to both `SolMover` and the gpt-3.5-turbo-1106 model. However, the increases in performance and decrease in non-compilable code synthesis are primarily observed in our framework. prompting us to look at our translation recipe of generating code using sub-tasks, and generation of the said sub-tasks using Concept Distillation.

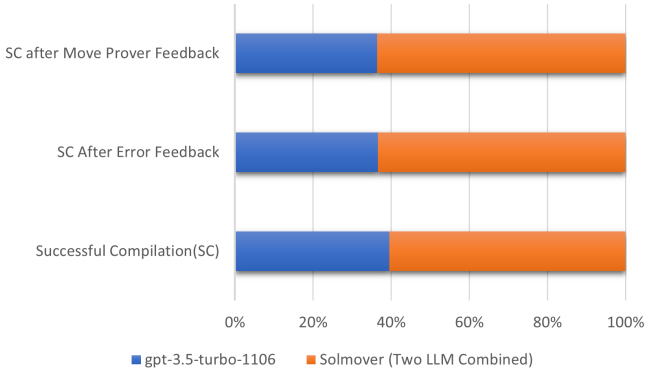

Figure 6: Contract Correctness.

> **RQ1 & RQ2** *Can the LLMs learn coding concepts? & Can the concepts be used to generate a granular sub-task from a generalized prompt?*
> We were able to empirically observe an increase in successful code translation for the model prompted by sub-tasks generated by the concept. Leading us to believe LLMs can encode associations resembling concepts. However, we will not actually call it "concept", but context association.

6.2 Smart Contract Translation

One of the primary goals of this paper is to explore whether one can translate smart contract code from one language (`Solidity`) to another smart contract language (`Move`). This is considered difficult by the fact that there are very few Move codes available in GitHub and not suitable for training a large language model. We also looked at different ways that we can improve the quality of the translated code by reducing bugs introduced by the LLMs. Our results are depicted in Figure 4. Successful translations give us confirmation that we can successfully translate an existing smart contract code written in `Solidity` to `Move`, and also beat the existing state of the art.

> **RQ3** *Can we generate code in a low-resource language that the LLMs have not been trained on?*
> Yes. Our empirical result suggests that without being trained in a specific language, an LLM can still translate a smart contract written in that language to a low-resource one with the help of sub-tasks and minimal fine-tuning.

Our iterative method of using both the compiler and `Move` Prover feedback decreases bugs in the translated code and increases code correctness, as we can see from Figure 6 (Contract Correctness) and Figure 5 (Bug Mitigation). We also observe that there is negligible improvement when it comes to the `Move` Prover feedback.

RQ4 *Can we mitigate bugs using compiler feedback?*
Yes. Our empirical result suggests that iterative prompting with compiler feedback does help reduce the number of uncompilable codes. This is true for both of the models, and our framework is more effective.

7 RELATED WORK

The related work can be categorized broadly as just Code Transpilers and LLM based solutions.

Rule-based Transpilers. A rule-based transpiler, like skilled interpreters, bridges the gap between programming languages. Instead of relying on guesswork, they follow predefined rules, ensuring accurate and predictable code transformations. This enables collaboration and code reuse, allowing users to seamlessly move their Python code to Java [63] or Java to Python [53]. While these tools offer precision and flexibility, they're not mind-reading magicians. Setting up the rules can require some programming expertise, and complex language features might need manual adjustments.

Statistical ML-based Transpilers. Statistical ML-based transpilers employ machine learning to translate between programming languages. Unlike rule-based methods, they learn from vast datasets of paired translations, identifying patterns and relationships to predict translations for unseen code. A phrase-based approach to code migration was introduced by Nguyen et al. [55], while Karaivanov et al. [38] enhanced this methodology by incorporating the grammatical structure of the target language and custom rules. Aggarwal et al. [21] applied a similar approach to convert Python2 to Python3, utilizing sentence alignments. Additionally, bidirectional transpilers, such as the one proposed by Schultes [64] for Swift [2] and Kotlin [1], have been explored. Ling et al.'s CRustS transpiler [47] stands out for its ability to reduce unsafe expressions in the target language through the implementation of extra transformation rules.

Transformer and Other ML-based Code Translation Tools. Transformer models, known for their usage in natural language translation, are applicable for code translation as well. The encoder-decoder architecture introduced by Vaswani et al. [70] in Transformers has had a profound impact on natural language (NL) translation, capturing intricate contextual relationships among words. This breakthrough influence extended to the realm of software engineering (SE), sparking a renewed interest in crafting specialized language models tailored for the unique challenges of code translation tasks. Different Transformer flavors like CodeBERT [31]and CodeGPT [50]are being brewed, some focusing on understanding code, others on generating fluent target language. Wang et al. [74] introduced CodeT5, an encoder-decoder Transformer incorporating code semantics derived from developer-assigned identifiers, resulting in more precise and domain-specific translations. Ahmad et al. [23] explored a unified Transformer model, PLBART, trained through denoising autoencoding, enabling versatile performance across both natural language (NL) and programming languages (PLs). Researchers are even experimenting with incorporating developer hints and exploring alternative architectures like tree-to-tree models. Chen et al. [30] pushed beyond the traditional sequence-to-sequence paradigm by introducing a tree-to-tree neural network tailored specifically for program translation.

LLM-based Methods Using Compiler/unit Testing. Roziere et al. [62] introduced an unsupervised code translation method that utilizes self-training and automated unit tests to ensure the equivalence of source and target code. It is noteworthy that they employ unit tests to construct a synthetic parallel dataset for model refinement; however, these tests are not integrated into the loss calculation during training. In a different context, Wang et al. [72] leverage RL with compiler feedback for code generation, again distinct from a supervised learning setting. Pan et al [35] did a comprehensive study on the existing LLMs for code translation. On the other side, Orlanski et al [57] explored the area of how low-resource coding languages are being affected in the LLM era.

Since our source and target both programming languages are smart contracts, providing bug free experience is one of the goals of the project. Pan et al [59] shows how code translation can introduce bugs in the translated code. While our work differs from the translation frameworks evaluated in that work, *Block 4* and *Block 5* strives to ensure the translated code should not introduce more bugs, as described in 5.2. However, this is an area we believe needs more exploration for code safety in LLMs.

8 GENERALIZATION AND FUTURE RESEARCH

In this work, we have evaluated the feasibility of LLM based smart contract translation using a novel approach that teaches the LLMs to understand coding concepts and then use that to translate code to an unseen language with scarce code snippets available for training or fine-tuning. Although our focus in this paper is translation from Solidity to Move, the framework and results may have practical implications for other low-resource or domain specific programming languages. It is an interesting future research direction to investigate whether the approach works for other low-resource language pairs. In addition, the research points to a new research direction of LLM - compiler co-design where compiler feedback can be optimized for effective LLM code-generation and translation.

9 CONCLUSION

In this work, we have proposed `SolMover`, a composite two-LLM-based framework that encodes textbook knowledge in one to generate a granular sub-task for the other to generate code based on that. We have evaluated the framework for the code translation task, involving an extremely low-resource code `Move` as a target language for smart contract creation. We have shown how our framework can take existing `Solidity` smart contracts and translate them to generate move smart contracts. Our contributions include introducing a way to encode concepts into an LLM, and showing how it can help LLM translate code into a non-trained source language. We also evaluate our iterative compiler error feedback loop to show that it can help mitigate bugs in the translated code.

REFERENCES

[1] [n. d.]. Kotlin Programming Language: Concise. Cross-platform. Fun. https://kotlinlang.org/.

[2] [n. d.]. Swift: The Powerful Programming Language that is Also Easy to Learn. https://developer.apple.com/swift/.
[3] 2022. *The Move Language - The Move Book.* https://move-book.com/
[4] 2023. *Can you feel the MoE? Mixtral available with over 100 tokens per second through Together Platform!* https://www.together.ai/blog/mixtral
[5] 2023. *DeFi.* https://github.com/MystenLabs/sui/tree/main/sui_programmability/examples/defi
[6] 2023. *Diem.* https://github.com/0LNetworkCommunity/libra-legacy-v6/blob/main/language/diem-framework/modules/Diem.move
[7] 2023. *fungible tokens.* https://github.com/MystenLabs/sui/tree/main/sui_programmability/examples/fungible_tokens
[8] 2023. *Gas.* https://github.com/0LNetworkCommunity/libra-legacy-v6/blob/main/language/diem-framework/modules/0L/GAS.move
[9] 2023. *GitHub - Elements-Studio/starswap-core: The swap project on Starcoin such as Uniswap a Sushiswap.* https://github.com/Elements-Studio/starswap-core
[10] 2023. *hf-codegen.* https://github.com/sayakpaul/hf-codegen/blob/main/data/prepare_dataset.py
[11] 2023. *Introduction - Move Patterns: Design Patterns for Resource Based Programming.* https://www.move-patterns.com/
[12] 2023. *Introduction - The Move Book.* https://move-language.github.io/move/
[13] 2023. *move · GitHub.* https://github.com/move-language/move/tree/main/language/documentation/tutorial
[14] 2023. *move/language/documentation/examples/experimental/basic-coin at main · move-language/move · GitHub.* https://github.com/move-language/move/tree/main/language/documentation/examples/experimental/basic-coin
[15] 2023. *move/language/documentation/examples/experimental/coin-swap at main · move-language/move · GitHub.* https://github.com/move-language/move/tree/main/language/documentation/examples/experimental/coin-swap
[16] 2023. *NFT.* https://github.com/MystenLabs/sui/tree/main/sui_programmability/examples/nfts
[17] 2023. *starcoin-framework/sources/MerkleNFT.move at main · starcoinorg/starcoin-framework · GitHub.* https://github.com/starcoinorg/starcoin-framework/blob/main/sources/MerkleNFT.move
[18] 2023. *Sui Basics - Sui Move by Example.* https://examples.sui.io/basics/index.html
[19] 2023. *sui · GitHub.* https://github.com/MystenLabs/sui/tree/main/sui_programmability/examples/fungible_tokens
[20] 2023. *Token.* https://github.com/starcoinorg/starcoin-framework/blob/main/sources/Token.move
[21] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. 2015. *Using Machine Translation for Converting Python 2 to Python 3 Code.* Technical Report. PeerJ PrePrints.
[22] Sweta Agrawal, Chunting Zhou, Mike Lewis, Luke Zettlemoyer, and Marjan Ghazvininejad. 2023. In-context Examples Selection for Machine 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, 8857–8873. https://doi.org/10.18653/v1/2023.findings-acl.564
[23] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.* Association for Computational Linguistics, Online, 2655–2668. https://doi.org/10.18653/v1/2021.naacl-main.211
[24] Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. 2023. Palm 2 technical report. *arXiv preprint arXiv:2305.10403* (2023).
[25] Yehoshua Bar-Hillel. 1960. A demonstration of the nonfeasibility of fully automatic high quality translation. *Advances in computers* 1 (1960), 158–163.
[26] Jeff Benson. 2021. Uniswap Trading Volume Exploded by 450% to $7 Billion. Here's Why. https://decrypt.co/63280/uniswap-trading-volume-exploded-7-billion-heres-why
[27] Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi, Stephane Sezer, Timothy A. K. Zakian, and Runtian Zhou. 2019. Move: A LanguageWith Programmable Resources. https://api.semanticscholar.org/CorpusID:201681125
[28] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. *ArXiv preprint* abs/2303.12712 (2023). https://arxiv.org/abs/2303.12712
[29] 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. (2021). arXiv:2107.03374 [cs.LG]
[30] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-Tree Neural Networks for Program Translation. *Advances in Neural Information Processing Systems (NeurIPS)* 31 (2018).
[31] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020.* Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139
[32] Daniel Gile. 2009. *Basic concepts and models for interpreter and translator training.* Number 8 in Benjamins Translation Library. John Benjamins, Amsterdam.
[33] J.Y. Girard, Y. Lafont, and L. Regnier. 1995. *Advances in Linear Logic.* Cambridge University Press. https://books.google.com/books?id=ROEf2h5FvD4C
[34] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks Are All You Need. *arXiv preprint arXiv:2306.11644* (2023).
[35] Sindre Grønstøl Haugeland, Phu H Nguyen, Hui Song, and Franck Chauvel. 2021. Migrating monoliths to microservices-based customizable multi-tenant cloud-native apps. In *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA).* IEEE, 170–177.
[36] Amr Hendy, Mohamed Abdelrehim, Amr Sharaf, Vikas Raunak, Mohamed Gabr, Hitokazu Matsushita, Young Jin Kim, Mohamed Afify, and Hany Hassan Awadalla. 2023. How good are gpt models at machine translation? a comprehensive evaluation. *ArXiv preprint* abs/2302.09210 (2023). https://arxiv.org/abs/2302.09210
[37] Wenxiang Jiao, Wenxuan Wang, Jen tse Huang, Xing Wang, and Zhaopeng Tu. 2023. Is ChatGPT A Good Translator? A Preliminary Study. In *ArXiv.*
[38] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based Statistical Translation of Programming Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software.* 173–184.
[39] Rabimba Karanjai, Zhimin Gao, Lin Chen, Xinxin Fan, Teweon Suh, Weidong Shi, and Lei Xu. 2023. DHTee: Decentralized Infrastructure for Heterogeneous TEEs. In *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC).* 1–3. https://doi.org/10.1109/ICBC56567.2023.10174906
[40] Rabimba Karanjai, Edward Li, Lei Xu, and Weidong Shi. 2023. Who is Smarter? An Empirical Study of AI-Based Smart Contract Creation. In *2023 5th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS).* 1–8. https://doi.org/10.1109/BRAINS59668.2023.10316829
[41] Rabimba Karanjai, Edward Li, Lei Xu, and Weidong Shi. 2023. Who is Smarter? An Empirical Study of AI-Based Smart Contract Creation. In *2023 5th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS).* IEEE, 1–8.
[42] Rabimba Karanjai, Lei Xu, Nour Diallo, Lin Chen, and Weidong Shi. 2023. DeFaaS: Decentralized Function-as-a-Service for Emerging dApps and Web3. In *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC).* 1–3. https://doi.org/10.1109/ICBC56567.2023.10174945
[43] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906* (2020).
[44] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. 2022. Holistic evaluation of language models. *ArXiv preprint* abs/2211.09110 (2022). https://arxiv.org/abs/2211.09110
[45] Libra Association Members. 2019. *An Introduction to Libra.*
[46] Libra Association Members. 2023. *The Libra Blockchain.* https://mitsloan.mit.edu/shared/ods/documents?PublicationDocumentID=5859
[47] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R Cordy, and Ahmed E Hassan. 2022. In Rust We Trust: A Transpiler from Unsafe C to Safer Rust. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings.* 354–355.
[48] Hongyuan Lu, Haoyang Huang, Dongdong Zhang, Haoran Yang, Wai Lam, and Furu Wei. 2023. Chain-of-Dictionary Prompting Elicits Translation in Large Language Models. *ArXiv preprint* abs/2305.06575 (2023). https://arxiv.org/abs/2305.06575
[49] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722* (2022).
[50] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv preprint arXiv:2102.04664* (2021).
[51] Elliott Macklovitch. 1995. The future of MT is now and Bar-Hillel was (almost entirely) right. In *Proceedings of the Fourth Bar-Ilan Symposium on the Foundations of Artificial Intelligence. url: http://rali. iro. umontreal. ca/Publications/urls/bisfai95.*

ps.
[52] Nicholas D Matsakis and Felix S Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.
[53] Troy Melhase, Brian Kearns, Ling Li, Iuliux Curt, and Shyam Saladi. [n. d.]. `java2python`: Simple but Effective Tool to Translate Java Source Code into Python. https://github.com/natural/java2python
[54] Shima Rahimi Moghaddam and Christopher J Honey. 2023. Boosting Theory-of-Mind Performance in Large Language Models via Prompting. *ArXiv preprint* abs/2304.11490 (2023). https://arxiv.org/abs/2304.11490
[55] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical Statistical Machine Translation for Language Migration. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*. 651–654.
[56] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748* (2018).
[57] Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. 2023. Measuring the impact of programming language distribution. In *International Conference on Machine Learning*. PMLR, 26619–26645.
[58] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.
[59] 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 *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 866–866.
[60] Karanjai Rabimba, Lei Xu, Lin Chen, Fengwei Zhang, Zhimin Gao, and Weidong Shi. 2022. Lessons Learned from Blockchain Applications of Trusted Execution Environments and Implications for Future Research. In *Workshop on Hardware and Architectural Support for Security and Privacy* (Virtual, CT, USA) *(HASP '21)*. Association for Computing Machinery, New York, NY, USA, Article 5, 8 pages. https://doi.org/10.1145/3505253.3505259
[61] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.
[62] Baptiste Roziere, Jie Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. TransCoder-ST: Leveraging Automated Unit Tests for Unsupervised Code Translation. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=cmt-6KtR4c4
[63] Saladi. [n. d.]. `py2java`: Python to Java Language Translator. https://pypi.org/project/py2java/
[64] Dominik Schultes. 2021. SequalsK – A Bidirectional Swift-Kotlin-Transpiler. In *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*. IEEE, 73–83.
[65] ShareGPT. 2023. ShareGPT: Share your wildest ChatGPT conversations with one click. Available at: https://sharegpt.com/.
[66] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.
[67] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
[68] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
[69] Josef Urban. 2004. MPTP–motivation, implementation, first experiments. *Journal of Automated Reasoning* 33 (2004), 319–339.
[70] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *Advances in Neural Information Processing Systems (NeurIPS)* 30 (2017).
[71] David Vilar, Markus Freitag, Colin Cherry, Jiaming Luo, Viresh Ratnakar, and George Foster. 2022. Prompting PaLM for Translation: Assessing Strategies and Performance. *ArXiv preprint* abs/2211.09102 (2022). https://arxiv.org/abs/2211.09102
[72] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. 2022. Compilable Neural Code Generation with Compiler Feedback. In *Findings of the Association for Computational Linguistics: ACL 2022*. 9–19.
[73] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560* (2022).
[74] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685
[75] Hao Wu, Wenxuan Wang, Yuxuan Wan, Wenxiang Jiao, and Michael R. Lyu. 2023. ChatGPT or Grammarly? Evaluating ChatGPT on Grammatical Error Correction Benchmark. *ArXiv preprint* abs/2303.13648 (2023). https://arxiv.org/abs/2303.13648
[76] Biao Zhang, Barry Haddow, and Alexandra Birch. 2023. Prompting Large Language Model for Machine Translation: A Case Study. *ArXiv preprint* abs/2301.07069 (2023). https://arxiv.org/abs/2301.07069
[77] Jianyi Zhang, Aashiq Muhamed, Aditya Anantharaman, Guoyin Wang, Changyou Chen, Kai Zhong, Qingjun Cui, Yi Xu, Belinda Zeng, Trishul Chilimbi, et al. 2023. ReAugKD: Retrieval-augmented knowledge distillation for pre-trained language models. (2023).
[78] Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark Barrett, and David L. Dill. 2020. The Move Prover. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 137–150.

Received 2024-04-05; accepted 2024-05-04