ASSESSING PROGRAM TRANSLATION CAPABILITIES OF LLM ENABLED CHATBOTS

by

LUKE MARSHALL

A THESIS

Presented to the Department of Computer Science

in partial fulfillment of the requirements for the degree of

Bachelor of Science

June 2025

Contents

Acknowledgments

I want to express my sincere gratitude to my primary supervisor, Professor Zena M. Ariola, for her guidance and expertise throughout this research and thesis. I am equally grateful to my secondary supervisor, Dr. Chris Misa, whose additional perspective and technical insights greatly enhanced this work. I would also like to thank my project advisor, Anthony Dario, for his direction and ideas during the early stages of this research. Finally, I am grateful to my parents for their support and encouragement throughout my academic journey, which made this work possible.

Abstract

This thesis examines the capabilities of Large Language Model (LLM) -enabled chatbot applications for translating programming languages. With the software industry facing challenges, including an aging COBOL infrastructure and widespread memory safety vulnerabilities resulting from the use of archaic programming languages, effective code migration strategies have become essential. We evaluate nine commercial chatbot applications from leading AI companies (Anthropic, Google, MetaAI, OpenAI, and xAI) in their ability to translate a complex OCaml project to Python, TypeScript, Rust, and C++. Our methodology employs a two-dimensional framework that examines prompting strategies (direct versus assisted) and translation approaches (functionalist versus linguistic), resulting in four distinct types of translation. Through analysis of 72 direct translations and seven assisted translations, we assess both the functionality and paradigmatic consistency. Results demonstrate that assisted translations achieve 75% success rate for functionalist approaches across target languages, while direct translations show highly variable performance (0-56% success rate) with consistent failures in Rust and C++. Linguistic translations using assisted approaches successfully demonstrated paradigm reorganization, while direct linguistic translations largely mirrored their functionalist counterparts. Our findings indicate that while LLM-enabled chatbots are not yet viable for fully automated code translation due to inconsistent success rates and systematic errors, they serve as effective tools when assisting human developers, particularly for complex projects that require paradigm shifts or the careful preservation of functionality.

Chapter 1

Introduction

This thesis investigates the viability of using Large Language Model (LLM) -enabled chatbot applications for translating programming languages across two key dimensions: prompting strategy (direct versus assisted) and translation approach (functionalist versus linguistic). The functionalist approach prioritizes the exact replication of program functionality, while the linguistic approach emphasizes code structure and the use of paradigms in the target language. These dimensions yield four distinct translation types, each requiring specific methodologies for data collection and analysis.

Our research methodology centers on translating a complex OCaml program into four target programming languages: Python, TypeScript, Rust, and C++. We utilize nine different commercial chatbot applications, representing the current state of the art, including models from Anthropic, Google, MetaAI, OpenAI, and xAI. The source program's complexity provides a realistic baseline for our investigation.

The primary goal of this thesis is to assess whether LLM-enabled chatbot applications are viable for translating programming languages, comparing the effectiveness of assisted versus direct approaches and functionalist versus linguistic translation strategies. This investigation contributes to the growing body of research on artificial intelligence (AI) -assisted software development while addressing practical concerns about code migration in the software industry.

We organize this thesis as follows: Chapter 2 provides background on programming

language translation approaches and current solutions; Chapter 3 details our methodology including source program and chatbot selection, target programming languages, and translation procedures; Chapter 4 presents the analysis of translation results; and Chapter 5 offers conclusions about the viability of different translation approaches and future research directions.

Chapter 2

Background

This chapter provides the context necessary to understand the current state of programming language translation and the role of LLM-enabled chatbot applications in addressing this challenge. We begin by examining the motivation behind programming language translation, including the issues of aging infrastructure and memory safety vulnerabilities that drive the need for systematic code migration. Next, we explore different approaches to programming language translation, drawing parallels with natural language translation theory to establish the functionalist and linguistic paradigms that structure our investigation. We then survey current solutions ranging from manual translation to AI-powered tools, with particular focus on the emergence of LLM-enabled chatbot applications as potential translation assistants or direct translators. Finally, we present our approach, which combines two prompting strategies (direct and assisted) with two translation approaches (functionalist and linguistic) to create a framework for evaluating chatbot translation capabilities. This background establishes the theoretical and practical foundations for our methodology.

2.1 Motivation

Programming languages have served as fundamental tools for Computer Science, dating back to the earliest days of physical computing [1]. These are written languages that both humans and computers can understand. Programming languages

are used to compose text documents known as programs. A computer can interpret a program into a set of instructions. The computer can then execute these interpreted instructions on its hardware. The underlying theory behind programming languages has a helpful feature. If a given language can be represented by a specific abstract framework, you can use this language to solve the same set of problems as any other language that can be represented within the same framework. Many programming languages aim to compute any computable function; computer scientists can represent any of them using the same specific framework and classify them as **Turing complete** [2]. This idea means developers can write any program composed with a Turing complete programming language in any other Turing complete programming language. We limit our investigation to Turing complete languages.

Newer programming languages reflect the hard lessons learned and the experience gained from dealing with the issues of older languages. The ability to write the same program in any Turing complete programming language means that the functional abilities of some older programming languages are just as strong as those languages created more recently. The power of older programming languages has led to their continued use up to the present day. One such example is **COBOL**, a programming language created in 1959 for business computing purposes. COBOL is known for being extremely verbose and complex to structure into coherent programs. Currently, around 40% of banks in the world use COBOL as their core technology [3]. The main issue with this is the lack of new developers entering the field with knowledge of COBOL. As experienced programmers retire, there is not a readily available supply of young developers who can take their place without extensive training specific to COBOL. One solution could be to train new hires in COBOL; however, a more realistic solution is to switch to a programming language with an established community of users.

A much larger issue exists in terms of **memory safety**, as memory issues are ubiquitous in programs written in many older programming languages. In programming languages that require **explicit resource management**, such as COBOL and the C language, developers are responsible for managing computer resources within the program

using the language's corresponding syntax. This syntax can be complex and relies on fallible humans to write perfect code. Bugs in managing many computer resources can be detected early in testing, but memory errors are especially challenging to find because there may be no indication that they exist, or they may only occur in specific instances during the execution of a program. The Microsoft Security Response Center stated in 2019 that, on average, 70% of the security vulnerabilities they fix are due to memory errors [4], and this statistic had held for more than a decade prior.

Additionally, in 2024, the White House released a report detailing the benefits of using **memory-safe languages** and encouraged companies to develop roadmaps for **migrating** their code to those languages [5]. Memory-safe languages are a category of programming languages that do not rely on syntax as their *primary* means of managing memory resources. Some of these languages automatically clean up after the program during execution, while others use static checks of the program text during compilation to ensure that the program does not contain memory errors before it even runs. Code migration is the process of translating the code of a given program from one programming language to another. The solution to these memory safety issues, as outlined in the White House's report, is to migrate and therefore translate code into memory-safe languages.

To summarize, the computer science industry faces numerous challenges, including an aging COBOL infrastructure and memory safety concerns in some older languages, all of which share a common solution: migrating to a different programming language to build our systems and applications. These problems provide an opportunity to investigate the capabilities of specific programming language translation strategies. The code translation process, enabled by current commercial tools, is what we examine here.

2.2 Programming Language Translation Approaches

Given the task of translating code to a different programming language, the immediate thought may be to find parallels between translating programming languages and translating **natural languages**. Natural languages are human languages, intended

primarily to be spoken, written, and interpreted by humans. Humans have been translating between natural languages for over 5 millennia [6], and there are many ideas we can apply to translating programming languages from the theory surrounding the natural language translation process. Two such ideas hold that written language has **function** and **shape**. The function of writing is to convey meanings, feelings, and ideas through words, as well as the actions that others may take as a result. The kinds of words used and their format are what provide a piece of writing its shape, such as the words and formats used to write a formal letter versus a Haiku. To use a **functionalist approach** to translating natural languages, you must translate based on the function of the resulting writing alone [7]. To employ a **linguistic approach** to translating natural languages, you must translate primarily based on the types of words and formats used in the source and target texts, with a *secondary* emphasis on ensuring that the functionality of both is the same. For this research, we limit our use of natural language translation concepts to the functionalist and linguistic approaches to translation.

We can map functionalist and linguistic natural language translation approaches to the process of translating programming languages. The translation approaches also provide goals for the program translation process. The functionalist approach to translating programming languages focuses on **function**. The function of a program can be specified as clearly as necessary to include any expected effects, and excluding other unspecified effects. This specification can then be used as a template to create the same functionality in the translation as was prescribed for the original. The linguistic approach to translating programming languages focuses on the **shape** of the program, with a secondary emphasis on functionality. The consistent use of keywords, expressions, statements, and other language elements within the same **paradigm** provides the shape of the program. These paradigms are a kind of format and way of structuring the code, similar to how many genres of poetry prescribe a strict layout. As a secondary measure, we do not consider the functionality of the linguistic translations. While functionality is the primary focus of the functionalist translation approach, the priority is consistent paradigm use in the linguistic translation approach.

The qualities of natural languages fundamentally differ from those of programming languages in ways that make translating the latter much simpler. When translating natural languages, there are no absolute meanings, so a functionalist translation can only ever be an approximation. The meaning of a piece of natural language writing to the reader depends on many factors. Programming languages, however, are designed to be unambiguous, meaning each part of a program has an explicit meaning. The **semantics** of the language prescribe exactly how the computer should interpret a given program. Translating based on functionality can be achieved simply by replicating functionality in the target language using the corresponding syntax, which mirrors the semantics of the source. This feature of programming languages extends to the paradigms, as each has its own set of syntax elements that exist to facilitate writing programs in one or more paradigms that the language supports.

Given some source natural language, it is impossible to tell if you can translate it into any given target language. A helpful constraint on programming language translation is to only use Turing complete languages. If both source and target programming languages are Turing complete, you can use either to write the same programs. Therefore, we know that you can translate any program written in a given source language into any program in a given target language. These two characteristics, explicit meanings and complete translatability, make programming language translation much simpler. Translating programming languages remains a complex task, and various solutions are being utilized and explored to aid in the process.

2.3 Current Solutions

There are many current and proposed solutions to the problem of programming language translation. The current solutions utilize manual translation, transpilers, code editor augmentation, and AI. You can split the AI solutions between direct use of LLMs and use of LLM-enabled applications. Manual translations require a human with in-depth knowledge of both source and target languages to make a best effort at translation. The

results of this vary based on the experience of the human translator, and there are no guarantees. Most current solutions incorporate some element of technology into the translation process and extend it with human support.

Two technology-based solutions to program translation are transpilers and augmented code editors. Transpilers are applications that attempt to translate between source and target programming languages based solely on **interoperability**. The goal is therefore to get the translated program to work in an environment that only supports the target language. This goal is less ambitious than a functionalist approach, as it entails a *best effort* at replicating functionality, with an emphasis on simply being able to execute the translated program. This transpiled code can be challenging to modify, as the transpiler provides no assistance after producing the translation. Developers must manually debug the transpiled program, and the same problems arise as in manual translation [8].

Tools such as augmented code editors with various extensions, plugins, and integrations, including the *Copilot* tool from Microsoft and the *Cursor* text editor, work natively with program files to provide integrated software development support, including for translating programs. Both of these approaches include a human element in the translation process to address shortcomings in the technologies. These shortcomings are more severe than the uncertainty that comes from including a human in the process. Current research focuses on leveraging recent advancements in AI to produce fully automated translations.

The recent advancements in the field of AI have led to the attempted integration of LLMs and the applications they enable into the code translation process. The work directly dealing with LLMs is only available to those groups who have both direct access to the models and can manipulate and run them locally. These research groups include both private companies, such as Meta, and public institutions, like UC Berkeley. These research projects treat the LLM as a step in the translation process, such as with Berkeley's *LLMLIFT* [9] tool, which utilizes an LLM to produce a translation and an accompanying proof, to ensure program correctness. You can then check the proof using automated theorem provers. Initial projects in companies like Meta, which developed

their own models, leveraged their privileged access to fine-tune the training of their LLMs and customize the processing to obtain better translations. These companies have since branched into integrating other tools or translating to better-understood intermediate representations [10]. The current applications of these LLMs outside of research projects include serving as the main driver for many code editor augmentations and tools, as well as in numerous commercial chatbot applications.

In recent years, due to advancements in LLM technologies, the capabilities and sheer number of commercial chatbot applications have exploded. **Chatbot** applications have an interesting relationship with **programs**. Programs are text documents written in programming languages. Chatbot applications take in text in the form of a prompt, do some work on it, and produce a corresponding text output. For this thesis, we limit our understanding of chatbot applications to this simplified format, treating the processing portion as a black box. This understanding of chatbot applications and programs leads us to the idea of using the text of a program as input, along with a prompt to translate the code into a target language. Past research has also provided evidence that chatbot applications may be a viable option for translating code [11], but there is room to investigate more specific criteria for chatbot application code translation.

You can use chatbot applications to translate programming languages in either an assistance capacity or directly. If you use a chatbot application for **assistance**, it serves as an additional tool available to a human developer performing manual translation. A developer using a chatbot application **directly** provides the chatbot some code in a source programming language and additional context in the form of a prompt to translate the source program into some given target programming language. If a chatbot is used directly for translation and the resulting program in the target programming language performs as expected, this is akin to an automated translation. If the direct use of the chatbot application results in some code that requires restructuring or debugging by a human, this aligns with the findings of other research on the results of direct chatbot programming language translations [12]. Current research does not compare the direct approach and the assisted approach in investigations into the viability of translating

programming languages using chatbot applications, nor does it build upon the approaches of natural language translation.

2.4 Our Approach

We investigate here the viability of using LLM-enabled chatbot applications in code translations on two dimensions: chatbot prompting strategy and translation approach. The chatbot **prompting strategy** can be either *direct*, simply prompting the chatbot with an instruction to translate some given code, or in an *assistance capacity*, with some limits on the prompt content. You can categorize **translation approaches** into two main types: *functionalist* and *linguistic*. The functionalist approach prioritizes recreating the original program's functionality in the target language. The linguistic approach treats functionality as secondary, instead focusing on achieving the desired *structure* of the translated program, ensuring that the code adheres to the target language's paradigms. These dimensions enable the production of four types of translations: assisted functionalist, assisted linguistic, direct functionalist, and direct linguistic. Each of these four translation types must have its own methodology for prompting strategy, data gathering, and analysis procedures.

Direct and assisted prompting strategies form two distinct categories. A direct prompting strategy includes the entire source program code in the prompt itself, along with any additional context needed for the chatbot to understand what it is supposed to do with the given code. This additional context may be an instruction to translate the source program in a specific way or to a particular target language, or it may be extra prompts to produce a complete translation. An assisted prompting strategy restricts the types of prompts users can make, specifically disallowing requests for direct translation of any portion of the original program. The chosen prompting strategy then creates the method of data collection.

The method used for collecting data is dependent on its format, which in turn depends on the prompting strategy. The data produced through direct prompts may

include code that requires reformatting for the program to run or extra response text. The final program should exclude this extra generated text. The *raw data* from direct prompting consists of the set of all prompts and responses. You can combine the code from the responses into a resultant program translation in the target language. The *processed data* from direct prompting consists of the code from the raw data, possibly with some light restructuring to eliminate duplicate code. The data produced through an assisted prompting strategy includes the constrained prompts and responses, as well as the final translated program. Researchers should collect a representative subset of the prompts and responses along with the final assisted translations. Having a record of prompts, responses, and accumulated translations then structures the analysis.

The analysis separates translations by approach: functionalist or linguistic. Functionalist program translations can be measured by their ability to perform the functions of the original program. Linguistic translations are more nuanced, and analyzing them requires relative measures. We can compare linguistic translations to their functionalist counterparts to identify consistent differences in the use of paradigms. We can also analyze the consistency of paradigm use within individual programs and mark adherence as either uniform or variable. We then collect and analyze this data to determine the validity of the corresponding program translation type.

Altogether, we prescribe each kind of program translation its own unique permutation of prompting strategy, data collection, and analysis steps. These steps inform the validity of the corresponding translation type. The translation itself represents the real-world ability to create a program, given a specific tool set and goals for the translation, either based on functionality or structure. This framework contributes to the current research by expanding the initial investigations into the capabilities of commercial chatbot applications in translating programming languages, specifically in the areas of paradigmatic code and prompting strategy.

Chapter 3

Methodology

This chapter outlines the systematic approach we developed to investigate the capabilities of LLM-enabled chatbot applications in translating programming languages. The structure of our methodology is centered around four key components that enable a comprehensive evaluation of program translation viability. First, we describe our source program selection, an OCaml networking project that serves as a representative example of a complex software project. Second, we present our selection of five commercial chatbot applications from leading AI companies, chosen to represent the current state of the art. Third, we justify our choice of four target languages (Python, TypeScript, Rust, and C++) based on paradigm compatibility. Finally, we outline our four distinct translation types: assisted functionalist, assisted linguistic, direct functionalist, and direct linguistic. Each translation kind has its own prompting strategy, data collection procedure, and analysis procedure. This multidimensional approach enables us to evaluate not only whether chatbots can translate code between languages, but also between programming paradigms.

3.1 Source Program

The source program serves as the base from which we translate using the four combinations of prompting strategy and translation approach. The choice of source program fundamentally constrains the conclusions we can make about translation methods using

chatbot applications. If we deem the translation a success or failure, we can generalize the findings from these results only to the extent that the source program itself generalizes. Given the goal of having our source program represent the vast array of possible programs, we chose a subset of files from the code repository for a networking project, created by Dr. Chris Misa at the University of Oregon. The files comprise a program that exhibits a subset of the original project's functionality. The programming language used to create the source program is OCaml, which is popular for academic projects and in the financial industry [13]. The program consists of components that build network telemetry query functions, along with a set of those query functions and a method for running them on a sample dataset. The sample data enables the simple testing of query functions in both the source program and the translation attempts, providing insight into the program's provided functionality. The code uses closures as stages in a complex data processing pipeline to track information about network packet headers. The implementation itself can be summarized as complex, requiring the use of many programming language constructs. The program employs a predominantly functional paradigm, incorporating some imperative constructs, such as mutable variables. The file itself contains numerous comments, ranging from brief explanations of functions to the author's notes on attempting to understand the code, as well as suggestions on how a function should be modified in the future. The natural disorganization that occurred during the program's development, reorganization, and exploration adds to its realism. The complexity of the source OCaml program and its realistic project allow us to draw conclusions about the capabilities of chatbot applications to translate programs between programming languages for general projects.

3.2 Selected Chatbots

We must select a representative sample of chatbot applications to understand their collective capabilities in the domain of programming language translation. Many companies now offer a wide variety of LLM-enabled chatbot applications, so we selected

a subset of each that represents the state of the art. There exist two classes of LLM, labeled by the industry as reasoning and non-reasoning. Most companies offer a choice of model between the classes to enable their chatbot application. We treat each class of LLM-enabled chatbot application equally, with no assumptions about their abilities. A third class of LLM, hybrid models, chooses between the reasoning and non-reasoning classes based on the user prompt. The companies that offer LLM-enabled chatbots do not all offer the same combination of classes and absolute number of models.

We selected five AI companies to survey. For each of the companies, we chose two models, if available, to enable the chatbot applications. The models were selected based on the recency of their release, with the goal of using a non-reasoning and a reasoning model from each company. When this exact configuration was not available, we substituted a hybrid model for either the reasoning model alone or both the reasoning model and the non-reasoning model. This replacement occurred in two instances. Anthropic only offers hybrid and non-reasoning models, so we use the hybrid model as a substitute for a reasoning model. MetaAI only provides a single hybrid model, which we use in place of the reasoning model, while we do not consider a non-reasoning model from their company. The actual LLMs chosen for each company to enable their chatbot applications are listed in Table 3.1 below. To translate programming languages using a chatbot application, each of the chosen models from each company can be used either in an assistance capacity or can be prompted directly for a translation. When translating between Turing complete programming languages, the ability to replicate the function of the source program is guaranteed. However, the shape of the translated program may differ depending on the kinds of paradigms each language supports.

Company	**Reasoning**	**Non-reasoning**
Anthropic	Claude 3.7 Sonnet	Claude 3.5 Haiku
Google	Gemini 2.0 Flash	Gemini 2.5 Pro
MetaAI	Llama 4	N/A
OpenAI	ChatGPT 4o	ChatGPT o4-mini
xAI	Grok 3	Grok 3 (with Think)

Table 3.1: Chosen LLMs to enable chatbots from each company

3.3 Selected Languages

The set of languages we investigate has an outsize impact on the kinds of conclusions we can draw based on translation results from a chatbot application. The source program in a translation will have a shape, determined by the set of paradigms used to write its code. These paradigms will come from the set of paradigms supported by the programming language used to compose the source program. Not all programming languages support the same set of paradigms. If the supported paradigms do not overlap between the source and target programming languages, this may affect the viability of a purely functionalist translation. We choose the target languages we investigate based on two constraints: they must support writing programs in the paradigms that the source language also supports, and programmers must use them for general programming. The second constraint maximizes the generalizability of the results when using a given programming language as the target for translation. Table 3.2 represents a comparison of prospective programming languages we considered for this project. Note that OO is the object-oriented paradigm. The table emphasizes the paradigms of OCaml and shows which other programming languages support overlapping paradigms. The table also lists any other paradigms that are supported by each considered programming language. We disqualified Haskell because it is not in general use, and C due to a lack of overlapping paradigm support with OCaml. We chose Python, TypeScript, Rust, and C++ as our target languages for translation.

Our target programming languages also have some interesting implementation differences, which could affect the resulting program created during a translation. We note the differences in our chosen programming languages before data is collected, so they can help inform the analysis of the translations. The most notable differences lie in the categories of execution model, type system, and resource management. The execution model is either interpreted or compiled. Errors in interpreted languages are found one at a time at runtime as the executing program encounters them. Compiled languages utilize a compiler that performs static checks to ensure the program adheres to the language's

Language	Functional	Imperative	OO	Other
OCaml	x	x	x	
Python	x	x	x	declarative
TypeScript	x	x	x	event-driven, declarative
Haskell	x			
Rust	x	x	x	generic, declarative
C		x		
C++	x	x	x	generic

Table 3.2: Programming Language Paradigms

Note: OO is Object-Oriented; an x signifies the programming language supports the corresponding paradigm

specifications and type system.

The type system for programming languages can be weak, medium, or strong. Weak type systems permit type coercion, the implicit casting of one type to another, which can lead to errors. Strong type systems disallow type coercion, but provide explicit mechanisms to cast types into other related types in a limited capacity. Medium-strength type systems fall somewhere in between, such as TypeScript, which performs static checks on the program but allows some bad practices that may still create situations where type coercion is possible.

Resource management in programming languages can be either automatic, as in garbage-collected languages, or explicit, which requires the developer to write it into the program explicitly. Explicit resource management always requires some syntax that must be written into a program in the exact correct way to avoid all resource management errors, especially memory errors. Programming language type systems vary in their approach to resource management. Some languages incorporate built-in checks that automatically verify proper resource handling, while others rely on developers to manually follow resource management conventions without compiler assistance. Table 3.3 summarizes some of these interesting programming language characteristics.

The language of the source program, OCaml, is a compiled language with a strong type system and automatic resource management. Python is an interpreted language with a weak type system and automatic resource management. TypeScript is a compiled lan-

guage with a medium-strength type system and automatic resource management. Both Rust and C++ are compiled languages with strong type systems and explicit resource management. C++ leaves resource management completely up to the developer, whereas the Rust compiler ensures proper resource use with static checks during program compilation. Overall, these languages represent a wide variety of supported paradigms and built-in features, enabling us to draw the broadest range of conclusions from our translation results.

Language	**Execution Model**	**Type System Strength**	**Resource Management**
OCaml	compiled	strong	automatic
Python	interpreted	weak	automatic
TypeScript	compiled	medium	automatic
Rust	compiled	strong	explicit
C++	compiled	strong	explicit

Table 3.3: Programming Language Characteristics

3.4 Translations

We are investigating the capabilities of current commercially available LLM-enabled chatbot applications to translate code using two prompting strategies: direct and assisted, each employing a functionalist or linguistic translation approach. The intersection between each dimension is a translation type: assisted-functionalist, assisted-linguistic, direct-functionalist, and direct-linguistic. Each translation kind requires a custom methodology. Each methodology contains the actual prompting strategy, data collection, and analysis procedures.

Functionalist translations emphasize functionality over all else. The functionality of the translated program should be identical to that of the source program. For this thesis, the source program is the same for all translation attempts. You may find an in-depth analysis of the source program in Section 3.1. The prescribed functionality of the source program can be vague or strict. Our criteria are as follows:

- The source program can be compiled and run.
- The source program contains 15 query functions.
- The source program prints a representation of an internal data structure when they are not consumed in the query functions, then prints 'Done'; otherwise, it only prints 'Done'.

Linguistic translations emphasize the shape of the program and see the replication of functionality between the program and translation as a secondary goal. The shape of a program is based on its paradigm use, which is reflected in its formatting and syntax, particularly in the use of keywords. The use of paradigms within a single program can be variable or uniform. Uniform paradigm use means that the same mixture of paradigms is consistently applied throughout a program; multiple paradigms are allowed, but they must be applied uniformly across the code. Variable paradigm use means the program does not adhere to a uniform mixture of paradigms throughout the code. We can compare the use of paradigms in corresponding functionalist and linguistic translations, according to the chatbot and prompting strategy used to create the translations, to identify any systematic differences between the two populations.

3.4.1 Prompting Strategies

Assisted Assisted translation prompting strategies are dictated by the needs of the human translator in the moment, so they cannot be specific. The assisted translation process is essentially a manual translation process with the help of a chatbot. Thus, it is difficult to maintain any single strategy. Instead, we choose to provide general limitations on the kinds of prompts that a translator may submit to the chatbots. These limits are as follows:

- No additional chatbots or models are allowed outside those in Table 3.1.
- No additional AI tools are allowed, such as Microsoft Copilot.
- Code is limited to small pieces written in the target language.

The assisted translation prompting strategy adheres to the limitations outlined above and focuses on helping the human translator achieve the desired program, given the chosen translation approach. The process of assisted functionalist translation is limited to understanding the minimal, highest-level functionality of the code line by line. Each piece in the translation utilizes features of the target language that are as semantically equivalent to those of the original language as possible. Therefore, the prompts provided to the chatbot should focus on replicating functionality from OCaml in the target language.

The process of assisted-linguistic translation involves understanding the minimal, highest-level functionality of the code line by line, as well as the paradigms used in the original program. Each piece of the translation reflects a possible reorganization into another paradigm and aims to produce a best effort at replicating the original program's functionality. In this case, the prompts given to the chatbots should focus on implementing a specific paradigm for the program translation. The assisted translation approach utilizes the chatbot as a tool to help a human translator with questions they may have in the moment.

Direct The direct prompting strategy is dictated by simplicity. To be viable for direct use in programming language translations, producing either a functionalist or linguistic program translation with a given chatbot should be as simple as asking for one. For this reason, we use minimalist prompts and focus on mapping the ideas behind each translation approach to the **base prompt**. The base prompt is the first prompt given to a chatbot in a single series of prompts and responses. If the chatbot does not provide an entire translation, additional prompts are used to direct the chatbot to give the rest of the translation. This procedure describes how we enable the chatbot to produce a program translation directly.

Given the goal of a functional or linguistic translation when using a chatbot to translate a program directly, we must explicitly communicate this through our prompts. To produce a direct functionalist translation, we must embed the functionalist viewpoint itself in a declarative prompt. We approximate the ideas of a functionalist translation

by prompting the chatbots with a minimalist instruction to translate the code from the original to the target language. To produce a direct linguistic translation, we must again embed the viewpoint in a declarative prompt. We approximate this by prompting the chatbots with a minimalist instruction to translate the program from the original to the target language, with the added context that only the natural paradigms and idioms of the target language should be used in the translation. This added context should provide a target for the translated program's paradigm, hopefully making the chatbot produce a more linguistically accurate translation. We then give the prompt, along with the code to translate, to the chatbot through its web interface.

For either translation approach, there may exist superior prompting strategies; we simplify here. This simplification is due to our criteria for direct translation viability being akin to automation. In an automated translation, a simple command can be given to produce a translation. If extra instruction is provided in the prompt, the translated program should reflect special attention paid to the aspects of the program translation mentioned in the prompt. The prompting strategies then inform the data collection and analysis procedures.

3.4.2 Code Translation Procedures

This section outlines the procedures for collecting translated code into program files, resulting from either assisted or direct prompting strategies. Gathering the code into a program is simple for assisted translations, as they are already in files as part of the translation process. Gathering the code resulting from a direct translation can be more complex.

Assisted The assisted translations, functional and linguistic, are collected within the code repository for this project[1] as they are written. Some example prompts and responses are also collected. The program collection process for assisted translations is simply creating a set of files to write the translation, and then updating those as the translations progress. These files can then be used for dynamic and static analyses.

[1]Project code repository: `https://github.com/lukemarshall2222/undergrad-research.git`

Direct For direct translations, any translated code produced by the chatbot is gathered into the corresponding file type for the given target language with minimal interference, such as deleting redundant code. If portions of a translation are provided more than once, the most recent version is kept. Non-code extraneous response text is not included in the program files. If the chatbot response prescribes a specific file structure, this will be used to organize the given translated code. The entire chat of prompts and responses for each translation is included in the repository for this project, in the same directory as its corresponding assembled translation. The direct translations can then undergo the same dynamic and static analyses as the assisted translations.

3.4.3 Result Analysis

Once the translated programs are created and accumulated, we begin the analysis. The analysis in this case involves gathering data on the program translations. The data collection is categorized according to the translation approach. A functionalist translation should create programs whose function aligns with the specifications in Section 3.4. The linguistic approach should create programs that pay special attention to the use of paradigms. This can be demonstrated through either producing a program that shows a large paradigmatic difference or contains a more uniform use of a given set of paradigms compared to its functionalist translation counterpart. The functionalist translations enable us to judge the linguistic translations based on their relative paradigm use.

Functionalist Results for functionalist translations are produced from attempts to run the code. For direct translations, we attempt to run the code up to ten times. If an attempt causes a simple error (<15 minutes to fix), it is corrected, and another attempt is made. Otherwise, the error is labeled as complex, and the analysis for that translation is ended. We also end the analysis after ten attempts to run the code have been made. The results of this analysis procedure are presented in Appendices A.1-4. For each attempt, the reasons for any errors are recorded along with the fixes if they are simple errors. We also track the number of query functions from the source program that

are translated and the percentage of these functions that function correctly.

For assisted translations, attempts are continually made to run the code throughout the translation process to produce a fully functioning program in the target language, although this is not always successful. This analysis must be conducted for each chatbot and model used to produce a translation. We only created one assisted translation per target programming language, but they are analyzed using the same methodology. This produces a table of results, including the number of functions provided, the number of functions that worked, and the success rate of the provided functions, as in Tables 4.1-2.

Linguistic Results for the linguistic translations are produced from observations about paradigm use in corresponding functionalist and linguistic translations. Each programming language we use supports multiple paradigms. The paradigm use can be noted by the shape of the program, which is defined mainly by the keywords and other syntax used to write the program, as well as formatting in some cases. The syntax used to recognize specific paradigms in the translated programs is mainly specific to the target language.

The syntax that serves as indicators that a given paradigm is being used in each of the target programming languages are listed in Tables 3.4-8. If the syntax corresponding to a given paradigm is seen in a translation, this paradigm is noted in the results. If the paradigms are used consistently throughout the program, this is recorded as uniform paradigm use, even if a mixture of multiple paradigms is used. If the paradigms are limited to specific portions of the program, this is recorded as variable paradigm use. We record the paradigms used in a translation and the uniformity of their distribution for both functionalist and linguistic translations of a given chatbot and target language. We may make conclusions about the difference a linguistic prompt makes in actual paradigm use compared to the corresponding direct functionalist translations. This analysis procedure can also be applied to comparing translations resulting from the assisted translation strategy.

OO	Functional	Imperative
class	let	ref
method	fun	:=
	→	!
	| >	if/then/else

Table 3.4: OCaml Paradigm Syntax

OO	Functional	Imperative	Declarative
class	def	for	comprehensions
self	lambda	while	generator expressions
	map, filter, reduce	if/elif/else	

Table 3.5: Python Paradigm Syntax

OO	Functional	Imperative	Generic
class	⇒	for	<T>
this	function	while	
dot operator	map/filter/reduce	if/else	
	forEach	switch	
	const		

Table 3.6: TypeScript Paradigm Syntax

OO	Functional	Imperative	Generic
struct	fn	for	<T>
enum	Fn/FnMut/FnOnce	while	type
impl	map/filter/fold/reduce	if/else	<‘a>
trait	collect	mut	
self	iter		
dyn	match/case		
where	Option<T>		
	Result<T>		

Table 3.7: Rust Paradigm Syntax

OO	Functional	Imperative	Generic
class	auto	for	template
struct	function pointers	while	typename
this	lambdas	switch	
new		if/else	
delete			

Table 3.8: C++ Paradigm Syntax

Chapter 4

Result Analysis

After the translation procedures are complete, translated programs exist for each chosen chatbot, chosen programming language, translation approach, and prompting strategy. Each of these programs must be analyzed according to its corresponding analysis procedure. In this section, we walk through the analysis of each kind of translation: assisted-functionalist, assisted-linguistic, direct-functionalist, and direct-linguistic. We also note some surprising results that we encountered during the analysis.

4.1 Functionalist Translations

Functionalist program translations are created with the goal of recreating the functionality of the source program with a target programming language. These translations can be made either with an assisted or direct prompting strategy. The assisted prompting strategy involves prompting chatbots with questions that focus on recreating the functionality of the source program in the target programming language. The direct prompting strategy involves a simple direct prompt to translate the given program from OCaml to a chosen target language. Once they are created, they may be tested against the specified functionality of the source program.

4.1.1 Assisted

Using the assisted prompting strategy and functionalist translation approach, we create a translated program for each of the chosen target languages. We then note the number of query functions provided and how many of them we are able to use to recreate the original functionality from the source program. Table 4.1 presents a summary of the results, including the success rate achieved with the provided query functions.

	OCaml	**Python**	**TypeScript**	**Rust**	**C++**
Programs translated	N/A	1	1	1	1
Functions provided	15	15	15	15	15
Functions working	15	15	15	15	0
Success rate	N/A	100%	100%	100%	0%

Table 4.1: Translation Results by Programming Language

4.1.2 Direct

Using the direct prompting strategy and functionalist translation approach, we create a translated program for each of the chosen target languages and each of the chosen chatbots. We then note the number of query functions provided and how many of them we are able to use to recreate the original functionality from the source program. Table 4.2 contains a summary of the results. The table expresses a key limitation of the direct translation approach we use. Because the code is generated through a non-standardized web interface, some query functions may be lost during the production or organization of the translated program code. We define the success rate as the percentage of provided query functions that successfully recreate the original functionality. A more detailed stepwise summary of the attempts made to execute the program translations can be found in Appendix A.

	OCaml	Python	TypeScript	Rust	C++
Programs translated	N/A	9	9	9	9
Functions provided	15	110	116	103	122
Average functions provided	N/A	12	12	11	13
Working functions	15	52	65	0	0
Success rate	N/A	47%	56%	0%	0%

Table 4.2: Translation Results by Programming Language

4.2 Linguistic Translations

Linguistic program translations are created to translate the code not only to another programming language, but also possibly to another set of paradigms. The translation could also reorganize the code to express the same paradigms, albeit with varying uniformity. Functionality is a secondary consideration in this translation approach, so we only measure the use of paradigms. In analyzing each translated program, we note the paradigms employed in both functionalist and linguistic translations. These paradigms are primarily recognized by the syntax used, as demonstrated in Tables 3.4-8. We also note the distribution of paradigm use throughout the program. If any of the paradigms used in the program are isolated to discrete sections of the code, this is variable paradigm use; otherwise, it is uniform paradigm use. The complete analysis is recorded in Appendix B.1-4. This analysis enables the comparison of functionalist and linguistic translations, as well as the relative measure of paradigm use in linguistic translations.

The procedures for analyzing assisted and direct linguistic translation results are the same. The measures can only be completed when both a functionalist and a linguistic translation are available for a given prompting strategy and chosen programming language. Due to time constraints, we were unable to create a linguistic C++ translation; therefore, this aspect is excluded from the discussion, except to highlight the limitations of the assisted translation methodology. This method of analysis is a limited relative measure, hoping to inform on the viability of using chatbots for linguistic program translations.

4.3 Notable Results

When using new tools, it is normal to encounter a few bugs. In the case of LLM-enabled chatbots, these bugs may manifest as pure logical decision-making bugs or hallucinations. On the purely logical decision-making side, we noticed some peculiar function names in certain translations, primarily related to a specific aspect of the source OCaml file. In the OCaml programming language, you can define infix operators easily. Most other programming languages, including those we use as target languages here, allow at most operator overloading, and otherwise do not include defining infix operators in their functionality. Almost every chatbot-direct translation, functionalist and linguistic, replaced the custom operator from the original OCaml with a function that replicated its behavior. The Gemini chatbot, enabled by their 2.0-Flash model, took the task of translation very literally and translated the original custom operator, denoted **@=>**, to a function named **at_equals_greater_than**. Hallucinations can also be an issue. For a week during the direct prompting procedure, the OpenAI chatbots would only produce the local weather report when prompted to translate the source program. The non-reasoning Grok 3 model also used the Chinese word for "New York" randomly in a function call. For a program translation to be useful, it must be readable and function correctly. Pure logical decisions and hallucinations, such as those we observed in our small investigation, reduce the utility of translated programs and diminish the viability of chatbot translations.

4.4 Discussion

Our investigative methodology yielded interesting results, both in terms of actual determinations on the viability of programming language translation using LLM-enabled chatbot applications and in highlighting their limitations. The results of the assisted-functionalist translation methodology, which utilized a chatbot to help translate code with the primary goal of replicating the source program's functionality, showed promise, as we achieved our goal with a 75% success rate across the four chosen target languages.

We were able to replicate this success rate in the assisted-linguistic translations. Our translated programs exhibited remarkable differences in format and paradigm use when different paradigms from those in the source program were selected for translation. This differs significantly from the results produced in the direct linguistic translations, which mostly mirror their functionalist counterparts. The repository that holds the original project, including the subset we used as our source program for translations, is a private repository. At one point in the translation process, the chatbot enabled by Google Gemini 2.5 Pro generated a link to the original project as an example of something similar to the code I provided. This is a significant problem, as it highlights our inability to determine what training material these models are allowed access to. Finally, the most considerable point against direct chatbot translations remains the inconsistent success of the translations simply in compiling, let alone executing and reproducing the specified functionality of the source program. This means direct chatbot translations as a whole are not yet viable for program translation.

Within the direct functionalist results, noticeable differences exist in the abilities of chatbot applications to translate between two groups of programming languages. If the chatbots achieve any success in translating the source program into Python and TypeScript, those programs are likely to be mostly complete and functioning. If not, the translations likely do not even function at all. All of the C++ and Rust direct functional translations were unsuccessful in replicating the functionality of the source program. The line between Python and TypeScript denotes the distinction between interpreted and compiled languages, with TypeScript also incorporating a type system. Crossing this line did not appear to affect the chatbot translation performance. The line crossing from TypeScript to Rust and C++ represents the distinction between languages that implement automatic resource management and languages with explicit resource management. This appears to be the increase in language complexity that causes the chatbot the most trouble. However, it is unclear whether the reason is actually the complexity of explicit resource management.

As noted in Section 4.3, fundamental issues hinder direct chatbot translations.

These issues include hallucinations, as random generated content cannot be allowed in code. Network problems, which hindered my ability to produce program translations, are also crucial in determining viability, as the system must be both available and reliable to be viable. Our results demonstrate the viability of assisted translations given some room for human fallibility, along with the impracticality of direct chatbot translations.

Chapter 5

Conclusion

5.1 Summary

This thesis examines the viability of LLM-enabled chatbot applications in translating programming languages, exploring both direct and assisted translation approaches from functionalist and linguistic perspectives. The research addresses the growing need for code migration in the software industry, driven by issues such as aging COBOL infrastructure and memory safety concerns in older programming languages. Using a complex OCaml academic networking project as the source program, we evaluate translations to Python, TypeScript, Rust, and C++ using nine different commercial chatbot applications. Our methodology creates four distinct translation types: assisted functionalist, assisted linguistic, direct functionalist, and direct linguistic. Results demonstrate that assisted translations achieve high success rates (75% for functionalist approaches), while direct translations show inconsistent performance with success rates ranging from 0% to 56%. The findings indicate that while LLM-enabled chatbots are not yet viable for fully automated code translation, they serve as effective tools when assisting human developers in the translation process, particularly for academic computer science projects.

5.2 Verdicts

Current research in the area of LLM-enabled programming language translations is mainly limited to direct-functionalist translations. LLM-enabled applications have been around long enough that it is time to examine how their limitations can be overcome with the assistance of a human developer. This is akin to the assisted translation portion of our investigation, specifically addressing how LLM-enabled applications assist a human developer in accomplishing a standard task. We must also investigate the capabilities of these applications on their own throughout their evolution, as in the direct translation portions of our investigation, to track their progress. Finally, we must also realize that programs can be translated not only between programming languages but also between paradigms, and this may be necessary to integrate a translated program successfully into an existing code base.

Overall, we can combine our findings with those of the broader field to draw some conclusions about the viability of different translation processes for translating academic computer science projects between popular academic languages. LLM-enabled chatbot-assisted functionalist translations are a viable method of program translation. Since human developers can manually translate programming languages, and adding a chatbot does not diminish this ability, this result is unsurprising. The additional insight our research provides is that translating in a linguistic approach also appears to be a viable option. Assisted translations have their downsides, primarily due to the human developer, as this process is significantly slower. This is exemplified by our inability to produce a working C++ functionalist translation or an attempt at a linguistic translation, as we ran out of time. The LLM-enabled direct chatbot translation process was not viable for either functionalist or linguistic program translations. This bore out in the data, which showed that the success rate, the highest with TypeScript at 56%, is not consistent enough for any of the programming languages we investigated to be used on their own without the help of a human developer to fix complex bugs in the resultant translated programs. This is replicated in other research, some with more success than us in producing functioning translations [14], but all of that shows the direct LLM-enabled translation process is

not yet viable due to inconsistent success. Thus, the programming language translation process is viable for either functionalist or linguistic program translations when an LLM-enabled chatbot application assists a human, but not yet when the chatbot is directly prompted for the translation.

5.3 Future Directions

Linguistics and translation theory have extensive historical catalogs that may be helpful in future translation approaches. A baseline amount of knowledge about programming languages is available through their documentation; however, chatbot assistance provides the immediate feedback necessary for active learning while completing a project. This could be a direction for education research, involving students in more challenging projects than they typically complete, either on a programming language basis or a project difficulty basis, and utilizing chatbot assistance for faster-paced education. Regarding the continuation of this research, many more complex prompting strategies could be investigated, such as how well chatbots can translate between specific paradigms for the same language.

Future work must incorporate a realistic analysis strategy. We produced realistic results by attempting to translate a subset of a real academic computer science project. The current model of using simplistic benchmarks does not account for the complexity of real programs and large codebases. The applications of this research reach into the software development field, where developers are actively migrating their code to other programming languages. Our research shows that these developers are better off using chatbots for assistance in translating programming languages, rather than relying on direct translations.

Appendix A

Direct Functionalist Translation Analysis Results

A.1 Python Translations

Company and Model	Steps	Functions Supplied	Working Functions
Anthropic Claude 3.5 Haiku	• First attempt: Incorrect keyword argument in function call; fixed the keyword • Second attempt: Incorrect keyword in every constructor for a class; renamed within the class definition to replace the original keyword • Third attempt: data representation mismatch for the main data structure; complex error • Complex error found after 22 minutes	9	0
Anthropic Claude 3.7 Sonnet	• First through sixth attempts: Incorrectly used custom Tuple data structure instead of built-in tuple in type hints; replaced with tuple • Seventh attempt: Function expects a location to write data, instead given the print function; replaced with standard location • Eighth attempt: The five given functions work as expected	5	5

Company and Model	**Steps**	**Functions Supplied**	**Working Functions**
Google Gemini 2.0 Flash	• First attempt: Incorrect type hinting syntax; deleted the type hinting • Second attempt: Incorrect name used in function body to access one of its parameters; replaced name in function body in all these cases (18 total) • Third attempt: Incorrect use of a built-in data structure as a key to a hash table when it cannot be hashed; complex error • Complex error found in 10 minutes	15	0
Google Gemini 2.5 Pro	• First attempt: The first function produced a different output than expected. The code itself runs many sets of data through a set of functions to replicate a realistic scenario; simplified function calls allow for standard testing • Second attempt: The 15 supplied functions work as expected	15	15
MetaAI Llama 4	• First attempt: Incorrectly used custom Tuple data structure instead of built-in tuple in type hints; replaced with tuple • Second attempt: Incorrect use of a built-in data structure as a key to a hash table when it cannot be hashed; complex error • Complex error found in 7 minutes	11	0
OpenAi ChatGPT 4o	• First attempt: The 13 supplied functions work as expected	13	13
Open-Ai ChatGPT o4 mini	• First attempt: Incorrect use of an object instance instead of the type hint; replaced with the type hint • Second attempt: Four of the supplied functions work as expected, the other 11 incorrectly use built-in data structure as a key to a hash table when it cannot be hashed; complex error	15	4

Company and Model	**Steps**	**Functions Supplied**	**Working Functions**
xAI Grok 3	• First attempt: Incorrect use of default parameters; deleted the default • Second attempt: Incorrect use of function application with incomplete argument lists; complex error • Complex error found in 23 minutes	15	15
xAI Grok 3 (with Think)	• First attempt: Incorrect use of default parameters; deleted the default	12	0
Assisted	N/A	15	15

A.2 TypeScript Translations

Company and Model	**Steps**	**Functions Supplied**	**Working Functions**
Anthropic Claude 3.5 Haiku	• First attempt: Incorrect use of an operator in multiple instances; removed the extra operators • Second attempt: Incorrect use of a data structure instead of one of its values; changed to return just the value instead • Third attempt: The 15 supplied functions worked as expected	15	15
Anthropic Claude 3.7 Sonnet	• First attempt: The five supplied functions do not work as expected, output is not produced; complex error • Complex error found in 21 minutes	5	0
Google Gemini 2.0 Flash	• First attempt: Incorrect use of an if statement; fixed the usage • Second attempt: Incorrect use of basic syntax and custom data collections; complex error • Complex error found in 12 minutes	15	0
Google Gemini 2.5 Pro	• First attempt: required library needs downloaded, the code contains a comment on how to download the library, I followed the directions; no changes made to the code itself • Second attempt: Incorrect import syntax; replaced the syntax • Third attempt: The 15 functions worked as expected	15	15
MetaAI Llama 4	• First attempt: built in data structure used with Python syntax; replaced with the correct syntax • Second attempt: Incorrect use of data structure functionality; complex error • Complex error found in 15 minutes	11	0

Company and Model	**Steps**	**Functions Supplied**	**Working Functions**
OpenAi ChatGPT 4o	• First attempt: Unknown complex error in one of the functions; deleted the problem function • Second attempt: 9/10 supplied functions produced the expected output, one function found to have a complex error	10	9
Open-Ai ChatGPT o4 mini	• First attempt: argument list out of order in function call; switched the order. Object used in place of a function; replaced with the function • The 15 supplied functions worked as expected	15	15
xAI Grok 3	• First attempt: Incorrect fields in custom objects in all 15 supplied functions; complex error • Complex error found in 24 minutes	15	0
xAI Grok 3 (with Think)	• First attempt: Incomplete argument lists in function calls; replaced the missing item • Second attempt: 11/15 supplied functions produced the expected output	15	11
Assisted	N/A	15	15

A.3 Rust Translations

Company and Model	Steps	Functions Supplied	Working Functions
Anthropic Claude 3.5 Haiku	• First attempt: * • Second attempt: 7 errors, 27 warnings. Unused variable names and imports, unnecessary mutability annotations, re-importing items; deleted extra names, imports, and annotations. Complex type mismatch. Fixed as many problems as I could locate, but requires another compilation to track progress. • Third attempt: 1 error, four warnings. Incorrect import of unused items and complex type mismatch • Fourth attempt: Chained mutability and memory management errors revealed once a type annotation error was corrected; complex error • Complex error found in 41 minutes	10	0
Anthropic Claude 3.7 Sonnet	• First attempt: * • Second attempt: 59 errors, three warnings. Incorrect use of variables out of scope. Possible complex errors using unimplemented object functionality. • Third attempt: 19 errors, 18 warnings. Incorrect mutability annotations, breaking memory management constraints, sharing values that should not be shared, and using unimplemented functionality on objects. • Complex error found in 36 minutes	15	0
Google Gemini 2.0 Flash	• First attempt: * • Second attempt: 2 errors. Incorrect delimiter usage, added the delimiters. • Third attempt: 11 warnings, 45 errors. Unused imports and variables, incorrect naming conventions. Complex errors in breaking memory management constraints, sharing values that cannot be shared, and using references as values. • Complex error found in 14 minutes	15	0

Company and Model	**Steps**	**Functions Supplied**	**Working Functions**
Google Gemini 2.5 Pro	• First attempt: 34 errors, eight warnings. Complex errors were found in breaking memory management constraints and sharing values that cannot be shared. • Complex error found in 5 minutes	10	0
MetaAI Llama 4	• First attempt: * • Second attempt: 48 errors, 11 warnings. Complex errors were found in breaking memory management constraints, sharing values that cannot be shared, using illegal operations, and using unimplemented methods on objects. • Complex error found in 8 minutes	9	0
OpenAi ChatGPT 4o	• First attempt: 4 warnings. None of the required functions were supplied, and no replacement of functionality was given • Complex error found in 2 minutes	0	0
Open-Ai Chat-GPT o4 mini	• First attempt: * • Second attempt: 1 error. Incorrect documentation comment; deleted the comment. • Third attempt: 16 errors, three warnings. Complex errors were found in breaking memory management constraints, sharing values that cannot be shared, and using unimplemented methods on objects • Complex error found in 23 minutes	15	0
xAI Grok 3	• First attempt: * • Second attempt: 93 errors, two warnings. Complex errors were found in breaking memory management constraints, sharing values that cannot be shared, using unimplemented functionality on objects, and using the wrong kinds of objects. • Complex error found in 4 minutes	14	0

Company and Model	**Steps**	**Functions Supplied**	**Working Functions**
xAI Grok 3 (with Think)	• First attempt: * • Second attempt: 12 errors, one warning. Complex errors were found in breaking memory management constraints and using the wrong kinds of objects. • Complex error found in 11 minutes	15	0
Assisted	N/A	15	15

* For attempting to execute the Rust code, I use the included Cargo build system. For this to work, the program files must be in a directory that contains a Cargo. toml file and a src subdirectory, or else an error will result. The Toml file includes the program's dependencies, and the src subdirectory contains all the actual code files and directories. If the chatbot response prescribes a layout for its program files, this structure will be followed for the first attempt, with corrections made to the layout if the build system constraints are not met initially.

A.4 C++ translations

Company and Model	Steps	Functions Supplied	Working Functions
Anthropic Claude 3.5 Haiku	• First attempt: 20 errors, three warnings. An object used throughout the file is never defined; complex error • Complex error found in 5 minutes	10	0
Anthropic Claude 3.7 Sonnet	• First attempt: 20 errors, 44 warnings. Importing items from libraries that do not contain them, using the wrong syntax for a given object assignment, or incorrect typing syntax. Complex errors are the assignment syntax and incorrect use of the class constructor • Complex error found in 14 minutes	15	0
Google Gemini 2.0 Flash	• First attempt: 1 error. Unknown library imported; deleted the import • Second attempt: 20 errors, eight warnings. Importing items from namespaces that do not contain them, using an undefined object; complex error • Complex error found in 7 minutes	15	0
Google Gemini 2.5 Pro	• First attempt: 21 errors. Using items with incorrect namespaces, prescribed a set of files and a command to compile them, but it does so incorrectly, such that each file may or may not know if imported items are defined; complex error • Complex error found in 11 minutes	15	0
MetaAI Llama 4	• First attempt: 1 error. Imports an uncommon library; complex error. • Complex error found in 3 minutes	12	0
OpenAi ChatGPT 4o	• First attempt: 20 errors, nine warnings. Using objects that it does not define, casting types incorrectly; all are complex errors • Complex error found in 4 minutes	12	0

Company and Model	**Steps**	**Functions Supplied**	**Working Functions**
Open-Ai ChatGPT o4 mini	• First attempt: 1 error. Imports an uncommon library; complex error • Complex error found in 3 minutes	15	0
xAI Grok 3	• First attempt: 20 errors, 12 warnings. Using objects it does not define, using items with incorrect namespaces • Complex error found in 3 minutes	15	0
xAI Grok 3 (with Think)	• First attempt: 20 errors, 13 warnings. Using objects that it does not define, using incorrect syntax • Complex error found in 5 minutes	13	0
Assisted	N/A	15	0

Appendix B

Direct Linguistic Translation Analysis Results

Below is the analysis of the results for the linguistic direct translations.

B.1 Python Translations

Company and Model	**Results**
Anthropic Claude 3.5 Haiku	• Functionalist translation: imperative, functional, minimal OO • Program shape: variable • Linguistic translation: imperative, functional, OO • Program shape: variable
Anthropic Claude 3.7 Sonnet	• Functionalist translation: imperative, functional, OO • Program shape: variable • Linguistic translation: imperative, functional, OO • Program shape: variable

Company and Model	**Results**
Google Gemini 2.0 Flash	• Functionalist translation: imperative, functional, OO • Program shape: variable • Linguistic translation: imperative, functional, OO • Program shape: variable
Google Gemini 2.5 Pro	• Functionalist translation: imperative and OO • Program shape: uniform • Linguistic translation: imperative and OO • Program shape: uniform
MetaAI Llama 4	• Functionalist translation: imperative, functional, OO • Program shape: variable • Linguistic translation: imperative, functional, OO • Program shape: variable
OpenAi ChatGPT 4o	• Functionalist translation: imperative, functional, OO • Program shape: variable • Linguistic translation: imperative, functional, minimal OO • Program shape: variable
Open-Ai ChatGPT o4 mini	• Functionalist translation: imperative and functional • Program shape: variable • Linguistic translation: imperative, functional, OO • Program shape: variable
xAI Grok 3	• Functionalist translation: imperative, functional, OO • Program shape: variable • Linguistic translation: imperative and functional • Program shape: variable
xAI Grok 3 (with Think)	• Functionalist translation: imperative, functional, OO • Program shape: variable • Linguistic translation: OO, imperative, functional • Program shape: variable

B.2 TypeScript Translations

Company and Model	**Results**
Anthropic Claude 3.5 Haiku	• Functionalist translation: functional and imperative • Program shape: uniform • Linguistic translation: functional and imperative • Program shape: uniform
Anthropic Claude 3.7 Sonnet	• Functionalist translation: functional and imperative • Program shape: uniform • Linguistic translation: functional and imperative • Program shape: uniform
Google Gemini 2.0 Flash	• Functionalist translation: functional and imperative • Program shape: uniform • Linguistic translation: functional and imperative • Program shape: uniform
Google Gemini 2.5 Pro	• Functionalist translation: functional and imperative • Program shape: uniform • Linguistic translation: functional and imperative • Program shape: uniform
MetaAI Llama 4	• Functionalist translation: functional and imperative • Program shape: uniform • Linguistic translation: OO, imperative, and minimal functional • Program shape: variable
OpenAi ChatGPT 4o	• Functionalist translation: functional and imperative • Program shape: uniform • Linguistic translation: N/A; did not receive a translation for this trial • Program shape: N/A

Company and Model	**Results**
Open-Ai ChatGPT o4 mini	• Functionalist translation: functional and imperative • Program shape: uniform • Linguistic translation: functional and imperative • Program shape: uniform
xAI Grok 3	• Functionalist translation: functional and imperative • Program shape: uniform • Linguistic translation: functional and imperative • Program shape: uniform
xAI Grok 3 (with Think)	• Functionalist translation: imperative • Program shape: uniform • Linguistic translation: functional and imperative • Program shape: uniform

B.3 Rust Translations

Company and Model	**Results**
Anthropic Claude 3.5 Haiku	• Functionalist translation: imperative and OO • Program shape: variable • Linguistic translation: imperative with minimal OO • Program shape: variable
Anthropic Claude 3.7 Sonnet	• Functionalist translation: imperative and functional with minimal generic • Program shape: variable • Linguistic translation: imperative and functional with minimal OO • Program shape: variable
Google Gemini 2.0 Flash	• Functionalist translation: imperative, OO, functional • Program shape: variable • Linguistic translation: imperative and functional with minimal OO • Program shape: variable
Google Gemini 2.5 Pro	• Functionalist translation: OO, imperative, and functional • Program shape: variable • Linguistic translation: OO, imperative, and functional • Program shape: variable
MetaAI Llama 4	• Functionalist translation: imperative and functional with minimal OO • Program shape: variable • Linguistic translation: imperative and functional with minimal OO and generic • Program shape: variable
OpenAi ChatGPT 4o	• Functionalist translation: OO and imperative • Program shape: variable • Linguistic translation: OO and imperative • Program shape: variable

Company and Model	**Results**
Open-Ai ChatGPT o4 mini	• Functionalist translation: imperative and functional with minimal generic • Program shape: variable • Linguistic translation: OO, imperative, and functional • Program shape: variable
xAI Grok 3	• Functionalist translation: imperative and functional • Program shape: uniform • Linguistic translation: imperative and functional • Program shape: uniform
xAI Grok 3 (with Think)	• Functionalist translation: imperative and functional with minimal OO and generic • Program shape: variable • Linguistic translation: imperative and functional • Program shape: variable

B.4 C++ Translations

Company and Model	**Results**
Anthropic Claude 3.5 Haiku	• Functionalist translation: imperative in OO • Program shape: uniform • Linguistic translation: keywords are OO, shape has some OO but is mostly functional • Program shape: variable
Anthropic Claude 3.7 Sonnet	• Functionalist translation: imperative and functional • Program shape: uniform • Linguistic translation: imperative and functional • Program shape: uniform
Google Gemini 2.0 Flash	• Functionalist translation: imperative with minimal OO use • Program shape: variable • Linguistic translation: imperative, generic, and OO • Program shape: variable
Google Gemini 2.5 Pro	• Functionalist translation: OO in imperative and OO in functional • Program shape: variable • Linguistic translation: keywords and shape are mixed imperative in OO • Program shape: uniform
MetaAI Llama 4	• Functionalist translation: OO in imperative • Program shape: uniform • Linguistic translation: OO in imperative, functional • Program shape: variable
OpenAi ChatGPT 4o	• Functionalist translation: imperative and functional • Program shape: variable • Linguistic translation: OO with minimal imperative • Program shape: variable

Company and Model	**Results**
Open-Ai ChatGPT o4 mini	• Functionalist translation: imperative and functional • Program shape: variable • Linguistic translation: imperative and functional • Program shape: variable
xAI Grok 3	• Functionalist translation: imperative, with minimal functional and generic • Program shape: variable • Linguistic translation: imperative in OO • Program shape: uniform
xAI Grok 3 (with Think)	• Functionalist translation: imperative in OO • Program shape: uniform • Linguistic translation: imperative, OO • Program shape: variable

References

[1] J. E. Sammet, *Programming Languages: History and Fundamentals* (Prentice-Hall Series in Automatic Computation). Englewood Cliffs, NJ: Prentice-Hall, 1969, pp. 5, 8. [Online]. Available: https://hcs64.com/files/Sammet%20-%20Programming%20Languages%20-%20History%20and%20Fundamentals.pdf.

[2] J. C. Mitchell, *Concepts in Programming Languages*. Cambridge, UK: Cambridge University Press, 2002, p. 14. [Online]. Available: https://homepages.dcc.ufmg.br/~camarao/lp/concepts.pdf.

[3] Luxoft, *How come cobol-driven mainframes are still the banking system of choice?* Mar. 2024. [Online]. Available: https://www.luxoft.com/blog/why-banks-still-rely-on-cobol-driven-mainframe-systems.

[4] M. S. R. Center, *A proactive approach to more secure code*, Accessed: [Your access date], Jul. 2019. [Online]. Available: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/.

[5] Office of the National Cyber Director, "Back to the building blocks: A path toward secure and measurable software," The White House, Tech. Rep., Feb. 2024. [Online]. Available: https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf.

[6] L. G. Kelly, "History of translation," in *The Dictionary of the Middle Ages*, New York: Scribner, 1979.

[7] UKEssays, *Contributions of functionalist approaches to translation*, Nov. 2018. [Online]. Available: https://www.ukessays.com/essays/translation/contributions-of-functionalist-approaches.php?vref=1.

[8] D. Venn and K. O'Neill, "The hidden costs of source-to-source translation: Debugging and maintenance challenges," *ACM Transactions on Programming Languages and Systems*, vol. 42, no. 3, pp. 1–28, 2020.

[9] S. Bhatia, J. Qiu, N. Hasabnis, S. A. Seshia, and A. Cheung, "Verified code transpilation with llms," in *Advances in Neural Information Processing Systems*, 38th Conference on Neural Information Processing Systems (NeurIPS 2024), vol. 37, Curran Associates, Inc., 2024. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2024/file/48bb60a0c0aebb4142bf314bd1a5c6a0-Paper-Conference.pdf.

[10] M. Szafraniec, B. Rozière, H. Leather, F. Charton, P. Labatut, and G. Synnaeve, "Code translation with compiler representations," *arXiv preprint arXiv:2207.03578*, Apr. 2023. [Online]. Available: https://arxiv.org/pdf/2207.03578.

[11] R. Pan, A. R. Ibrahimzada, R. Krishna, *et al.*, "Understanding the effectiveness of large language models in code translation," *arXiv preprint arXiv:2308.03109*, Aug. 2023. [Online]. Available: https://www.researchgate.net/publication/372961610%5C_Understanding%5C_the%5C_Effectiveness%5C_of%5C_Large%5C_Language%5C_Models%5C_in%5C_Code%5C_Translation.

[12] R. Pan, A. R. Ibrahimzada, R. Krishna, *et al.*, "Lost in translation: A study of bugs introduced by large language models while translating code," *arXiv preprint arXiv:2308.03109*, Aug. 2023. [Online]. Available: https://arxiv.org/abs/2308.03109.

[13] Y. Minsky and S. Weeks, "Caml trading – experiences with functional programming on Wall Street," *Journal of Functional Programming*, vol. 18, no. 4, 2008. DOI: 10.1017/S095679680800676X.

[14] H. F. Pan, S. Yee, A. Chakraborty, N. Nagasamudram, D. Oliveira, and M. W. Godfrey, *Towards translating real-world code with llms: A study of translating to rust*, 2024. arXiv: 2405.11514 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2405.11514.