Using GPT-4 to Assist in C to Rust Translation

WEDNESDAY, SEPTEMBER 27, 2023

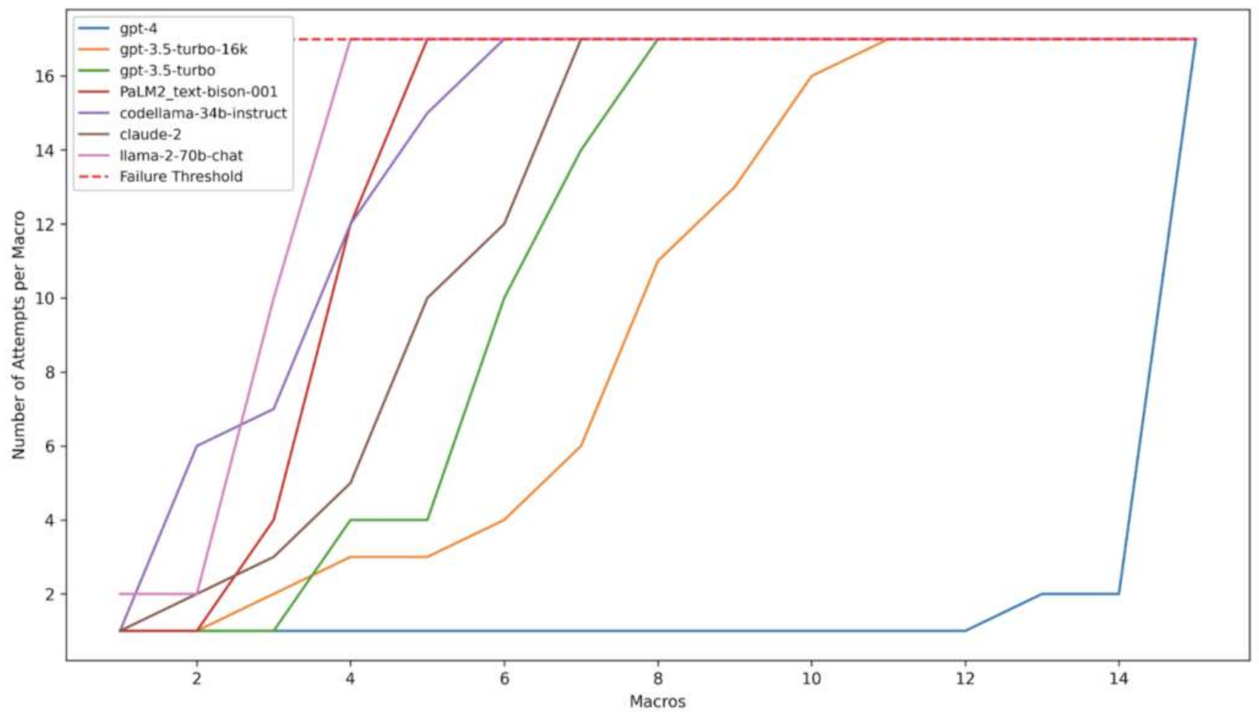
ARTIFICIAL INTELLIGENCE (HTTPS://GALOIS.COM/BLOG/CATEGORY/ARTIFICIAL-INTELLIGENCE/)

Adam Karvonen (https://galois.com/blog/author/adam-karvonen/)

At Galois, we have been experimenting with multiple Large Language Models (LLMs), including GPT-4. Part of the motivation for this is to continue increasing the accessibility and utility of our formal verification and software assurance tools. While these tools provide high assurance for critical software, they can require significant expertise or time to extract full value; LLMs hint at ways to reduce these burdens. In my last blog (https://galois.com/blog/2023/08/applying-gpt-4-to-saw-formal-verification/), I showed some initial results using LLMs to help perform formal verification by creating SAW memory safety proofs. For this blog, I produced a refactoring tool that verifiably rewrote 2900 lines of real world open source code.

We decided to focus on tasks that used the LLM to refactor code where the behavior of the program should not change. Many of these tasks exist in the translation of C to Rust and the incremental rewriting of unsafe, unidiomatic Rust to safe, idiomatic Rust. To prototype this, I wrote a Python program that used deterministic "hard logic" wherever possible and used GPT-4 as "soft logic" when necessary. Because I could check equivalence to verify the ground truth correctness of the refactoring, GPT-4 could freely rewrite code within the structure of the program, relying on this verification step to completely eliminate the risks of hallucinations.

Initial tests looked promising. However, GPT-4 is great at making promising demos, which don't always translate to practical value. I applied the approach to real world code, and it performed around 80% of the task, rewriting thousands of lines of code with a high level of assurance. Based on these results, I conclude that LLMs might be surprisingly effective in some high-assurance situations when combined with tools that can rigorously check the model's output.



(https://galois.com/wp-content/uploads/2023/09/Screenshot-2023-09-27-at-1.10.16-PM.png)

Macro creation attempts for first success for a range of macros across multiple currently available LLMs; results improve as they approach the bottom right of the graph. See article for more information.

Introduction

LLMs have both strengths and weaknesses. These weaknesses mean they cannot perform an entire task reliably without leveraging additional logic, but their strengths can complement existing tools. Because of this, I have been searching for methods to combine LLMs with existing tools, rather than having the LLM perform the entire task from end to end.

One tool that may benefit from LLM integration is C2Rust (https://github.com/immunant/c2rust), which is being developed by Immunant and Galois. C2Rust is a translator that operates on a C program's Abstract Syntax Tree (AST) to translate the program to Rust. Because the program operates on the semantics of the C language, it is able to output Rust code that compiles and runs, although the code is still unsafe and unidiomatic. This is great, and enables incremental rewrites to safe, idiomatic Rust while running unit tests. There is a lot of valuable software written in C, and enabling translation to a modern, safer language like Rust could eliminate a variety of memory safety bugs that currently plague C programs.

However, C2Rust operates purely on the semantics of the language, and as a result the translation doesn't preserve some desirable syntactic qualities, which contributes to poor readability. These desirable syntactic qualities include including comments in the Rust translation, properly translating for loops, translating C goto / switch usage into a readable Rust control flow, and using macros in the Rust output. Currently, fixing these syntactic issues requires many hours of manual effort. These issues also have a convenient property: the behavior of the program should not change at all during refactoring, which enables verification of the refactoring, which means it's much more feasible to use automated and probabilistic tools.

I decided to apply GPT-4 to the task of refolding macros into the Rust program.

Macro Refolding Overview

C macros operate on the level of string concatenation with the C preprocessor (CPP). Because C2Rust operates on the AST, it must work with the macro-expanded preprocessor output rather than the original abstracted macros. Unfortunately, this means that the abstraction of the original C macro is lost. As a concrete example, consider the following toy C macro, SQUARE_OF_DECREMENTED.

```
#define SQUARE_OF_DECREMENTED(x) ((x – 1) * (x – 1))

int call_macro(int y) {

    return SQUARE_OF_DECREMENTED(y);

}
```

C2Rust outputs the following Rust function:

```
pub unsafe extern “C” fn call_macro(mut y: libc::c_int) -> libc::c_int {
    return (y – 1 as libc::c_int) * (y – 1 as libc::c_int);

}
```

It would be very helpful if we could create a Rust macro and refold it into the function, restoring our previous abstraction. In addition, for longer C macros, this can cause the length of a Rust file to blow up in length. For a simple one line macro like SQUARE_OF_DECREMENTED, the increase in length is trivial. However, some C programs have 40 line macros that are used dozens of times, **which in one case transformed a 4,000 line C program into a 24,000 line Rust program.**

Using LLMs in an automated fashion is challenging because they cannot be blindly trusted, and it’s often difficult to automatically evaluate if a task was performed correctly. However, in the case of refactoring tasks such as macro refolding, we know that the behavior of the program should not change in any way. So, if there is a way to check the equivalence of the modified and original code, we have the flexibility to use tools that are less precise, such as LLMs, and verify their generated solution through the equivalence check.

In the case of macro refolding, there is a simple way to obtain the equivalence ground truth. We can partially compile the original and modified code to an intermediate representation, such as the High level Intermediate Representation (HIR), LLVM, or assembly. If the original code and modified code produce identical HIR output, we have a high level of assurance that both versions of the code are equivalent.[1]

Here is the corresponding macro folded into the previous `call_macro` function, with every type cast and parentheses exactly placed to produce identical HIR output to the original C2Rust output.

```
macro_rules! square_of_decremented {

    ($x:expr) => {

        ($x – 1 as libc::c_int) * ($x – 1 as libc::c_int)

    };

}

pub unsafe extern “C” fn call_macro(mut y: libc::c_int) -> libc::c_int {

    return square_of_decremented!(y);

}
```

My early tests for this application looked promising. I wanted to push further than a cool demo and try this approach on rewriting a large real world file. I chose to tackle mc_tmpl.rs (https://github.com/memorysafety/rav1d/blob/cc78ff0f2bced0cce554e9a59e75fc8866245d30/src/mc_tmpl.rs), a file from the rav1d codebase. Rav1d began as a Rust port of the open source dav1d (https://code.videolan.org/videolan/dav1d) AV1 decoder, generated by C2Rust translation. The original mc_tmpl.c (https://github.com/memorysafety/rav1d/blob/main/src/mc_tmpl.c) had 953 lines of code, but mc_tmpl.rs had grown to 4303 lines, largely due to unfolded macros. It contained 20 different macros used 85 different times, and the longest macro was 45 lines in the original C. A real world test like this is a great way to get a better sense of the capabilities of LLMs.

Approach

As discussed in the overview, my approach focused on using GPT-4 as a probabilistic “soft logic” in a Python program. I used deterministic “hard logic” wherever possible, and used GPT-4 as “soft logic” when necessary. The majority of the program was hard logic performing structured tasks, such as constructing a Rust-based parser to generate a detailed index of functions, macros, and their corresponding line numbers. I also created prompt templates, and iterated over these functions and macros, inserting them into the proper place in each template before prompting GPT-4.

From experience, I knew that it was essential to break the task down into simpler subtasks wherever possible. So, I split my usage of GPT-4 “soft logic” into macro creation and macro insertion. For this proof of concept, I did allow some manual work, and I took a function using each macro, deleted all lines not associated with the macro usage to simplify the task, and stored the simplified example. This took a small amount of effort, and from my testing GPT-4 would also have been capable of performing this task, but I didn’t take the time to implement it. An example can be seen here. (https://github.com/GaloisInc/blog-saw-and-llms/blob/main/blog-c2rust-and-llms/simple_function_example.txt)

For macro creation, my program iterated over a list of macros, inserting each macro and simplified function into a prompt template to GPT-4. The program would regex the code from GPT-4’s response, partially compile it to HIR, and store the result if it was equivalent to the original code’s HIR. If an error or failed equivalence check was returned, the appropriate prompt template and associated error or difference was appended to the conversation and sent to the model. An example transcript can be seen here. (https://github.com/GaloisInc/blog-saw-and-llms/blob/main/blog-c2rust-and-llms/creation_example.txt)

GPT-4 was able to successfully create all 20 Rust macros that compiled to HIR equivalent to the original function. Producing Rust code that targets C2Rust HIR output requires creating an unusual, unidiomatic form of Rust. In 20 minutes of googling I was unable to find any examples of anyone doing something similar to this, with or without an LLM. This indicates the task is probably not present in its training data, yet GPT-4 was successful. This corresponds with other tests I've done using GPT-4, such as having it perform arithmetic in non-standard formats such as base 9.

Once the macros were created, the program then used GPT-4 to insert them into the mc_tmpl.rs Rust program. In this case, the program used a parser to create a list of C and Rust functions and macros. Then it iterated over this list, inserting the C and Rust functions and macros into the appropriate positions of a prompt template. GPT-4 then produced a JSON output, specifying which lines needed replacing and supplying the new code. The program would regex the JSON response, insert the code response into the full program, create and check equivalence of the HIR output, and select the appropriate response prompt if there was an error. An example transcript can be seen here. (https://github.com/GaloisInc/blog-saw-and-llms/blob/main/blog-c2rust-and-llms/insertion_example.txt)

The JSON output for the SQUARE_OF_DECREMENTED example would look like this:

{

"lines_to_replace": [2],

"new_rust_code": "return square_of_decremented!(y);"

}

Results

GPT-4 was able to automate approximately 80% of the task. It is difficult to quantify the exact number. GPT-4 was able to successfully create all 20 macros, which is the more difficult and time consuming part of the task. I didn't fully flesh out the equivalence checking capability of my program, which meant that it was only able to give GPT-4 feedback on 60 out of 85 macro usages.[2]

GPT-4 was able to successfully insert 46 out of the 60 possible macro usages into the Rust code. Overall, the length of the file (https://github.com/GaloisInc/blog-saw-and-llms/blob/main/blog-c2rust-and-llms/mc_tmpl_final.rs) decreased by 1,600 lines and 2,900 lines were deleted or rewritten. It's interesting that it was less successful at the easier subtask. One plausible explanation is that asking it to output machine readable commands is an unusual task that it hasn't seen very often in training. Performance on this task may improve by giving GPT-4 a more natural output format, or by fine tuning it on examples of the output format.

There are a variety of ways that this approach can be improved. The first would be to focus on more sophisticated and robust equivalence checking. When examining the failures of GPT-4, it produces valid code that is 90% of the way to perfectly matching the HIR output but cannot quite match it perfectly (https://github.com/GaloisInc/blog-saw-and-llms/blob/main/blog-c2rust-and-llms/creation_failure_example.txt). Some macros have up to 60 type casts in one usage, some nested in parentheses several levels deep (https://github.com/GaloisInc/blog-saw-and-llms/blob/main/blog-c2rust-and-llms/mc_tmpl_final.rs#L351). Perfectly matching this output may be a task not well suited for LLMs, akin to how GPT-4 struggles to count the number of occurrences of "n" in mayonnaise. An added bonus of better equivalence checking is that we could verifiably simplify many of the macros and make the code more readable.

However, because GPT-4 usually got 90% of the way there in failure cases, it's also very plausible that simply waiting for the next generation of models may significantly improve accuracy as well. If there is another jump in capability similar to GPT-3.5 to GPT-4, that alone may be enough to fix many of the current failing examples. In addition, because of the ground truth feedback with this task we can use GPT-4 to construct a labeled dataset for fine tuning the GPT-4 base model, which should be available this year (https://platform.openai.com/docs/guides/fine-tuning). Finally, it is always possible that an improved prompt engineering approach could also improve results as well.

Takeaways

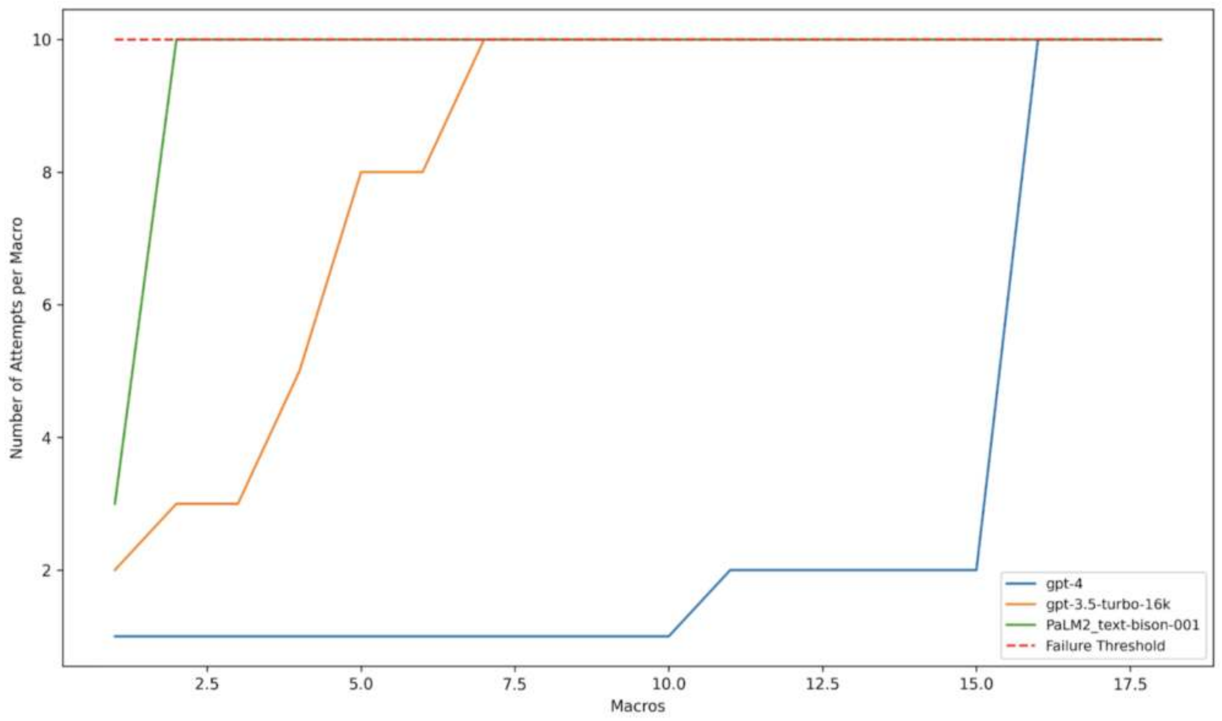
This research demonstrates the potential for sufficiently capable models like GPT-4 to produce practical value on complex coding tasks when used properly. It would be very unwise to use current LLMs to autonomously push code without safeguards, so this autonomous use is currently constrained to domains with automatic verification. However, there is a broad category of refactoring tasks where the behavior of the program should not change in any way, such as translating C to Rust or incrementally rewriting unsafe Rust to safe Rust, and this technique could be applied anywhere it's possible to check the equivalence of the old and new programs. Galois has developed tools that can formally verify equivalence between complex pieces of real-world code: for example, our Software Analysis Workbench (https://saw.galois.com/tutorial.html) (SAW) and Crux (https://crux.galois.com/) formal verification tools.

Like many automation tasks, it was fairly easy to get to ~80% accuracy, and the last 20% would have required significant additional effort. I have found this is the case with using GPT-4 in a variety of tasks. I believe with the present generation of models, it's more practical to design tools that bolster productivity rather than aiming for complete automation. If I was going to further develop this proof of concept, I'd focus on enhancing its usability as an interactive tool, especially when GPT-4 falters. Even when not entirely accurate, GPT-4 always provides a strong starting point. Additionally, for many programming tasks there isn't a good source of ground truth feedback and in those cases interactive tools are much more practical than full autonomy.

I have also found that fine tuning the prompts and workflow these models are used in can go a surprisingly long way without any finetuning of the base model. In both macro creation and macro insertion, my initial attempt started with correctly handling around 20% of the cases. With around 6 main iterative improvements in each case, accuracy increased to 100% for macro creation and 75% for macro insertion. Prompt engineering can be highly effective and allows for rapid iteration.

These improvements were not tweaking one word or phrase. In general, if you are tweaking one word hoping for better results, you're probably doing something wrong. Instead, these improvements involved steps like creating higher quality few shot examples, adding a chain of thought to my examples guiding the model in a particular direction, adding intermediate steps to the chain of thought, using my parser to add more context to the prompt, using a more natural output format, and using the program to further break down the problem into smaller subtasks. For comparison, you can view my initial prompt (https://github.com/GaloisInc/blog-saw-and-llms/blob/main/blog-c2rust-and-llms/first_creation_prompt_attempt.txt) and final prompt (https://github.com/GaloisInc/blog-saw-and-llms/blob/main/blog-c2rust-and-llms/final_creation_prompt.txt) for macro creation.

I logged the results when creating and inserting macros with GPT-4, and also ran several other models on the same benchmark, including Google's Palm 2, Facebook's Llama 2, Anthropic's Claude 2, and GPT-3.5.[3] Figure shown at the start of this article. A few findings stand out. The first is that, as I've seen before, GPT-4 is significantly better than all other available LLMs for complex coding tasks and it enables many exciting new use cases. Secondly, LLMs are very general. A single prompt with a few shot-example worked, to a varying extent, on 7 different models made by 4 different companies. Results for JSON output specifying where these macros need to be reinserted are in the figure below.



(https://galois.com/wp-content/uploads/2023/09/Screenshot-2023-09-27-at-1.15.01-PM.png)

Attempts required to successfully specify macro insertion locations via a JSON format; bottom right is best.

Tests like these could also be used to address the problem that benchmarking and evaluating Large Language Models is notoriously difficult. Leaderboards (https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard) have been created with various benchmarks, but many have complained that models that perform well on benchmarks do much worse in real world tests. Using a diverse set of real world evaluations, such as the ability to create macros that pass a machine evaluation, offers a path towards more useful benchmarks and evaluation.

Conclusion

These experiments demonstrate that GPT-4 can make valuable and autonomous contributions to complex coding tasks when a source of ground truth feedback exists. A similar approach could be applied in a variety of ways when translating legacy languages like C into modern, safer languages like Rust, or when doing incremental refactors to improve software quality. However, many tasks don't have this ability, especially anything that involves creating new software. For the next step of this research, we're going to continue exploring practical uses of LLMs, but moving over to problems where any notion of ground truth requires human oversight or partnership. We're excited to continue using new machine learning technologies to augment human programmers.

I would like to thank Stuart Pernsteiner for his feedback, advice, and general Rust expertise.

Appendix

I did some tests with GPT-4 to perform other rewriting tasks on C2Rust output. I had promising results on inserting comments (https://chat.openai.com/share/b116b1cc-3720-4a9c-b78a-bd2f77378faf) into the Rust output and rewriting C2Rust while loop outputs (https://chat.openai.com/share/76ce4121-8de6-427b-bcca-533baab26e65) back into the original for loops.

Additional model evaluation details: These results should be treated as early stage and limited in scope. This evaluation gave every model 15 attempts to correctly write the macro, and stopped on the first success. Because the benchmarks track the number of attempts required for the first success, rather than the number of successful attempts over a high number of trials, there is an element of luck involved. In addition, I first tuned the prompt to work with GPT-4. When running each model, I spent between 15 minutes and 1 hour inspecting model transcripts and tuning the prompt to address the most common shortcomings, and I ensured each model was returning properly formatted responses that were picked up by the regex. However, I had spent several hours tuning the prompt on GPT-4. Palm2-text-bison-001 was used instead of code specific Palm2 models because in my testing, it was the only model able to follow my formatting requirements for use in an automated tool. Llama2 and GPT-3.5 were not used for macro insertion because their 4,000 token input limit was too low for many functions. They also required a slightly shorter prompt to fit some macros into their context length.

Footnotes

[1] For lower level representations like LLVM, we have a lower level of assurance, because it's possible that a future Rust compiler may produce different LLVM.

[2] The primary reason it wasn't fully fleshed out is that some macros used the `paste` crate, which required extra steps to emit intermediate representations. I didn't take the time to add that capability to this proof of concept, and instead I did the extra steps by hand for those examples.

[3] See appendix for additional details and experimental limitations.

PREVIOUS (HTTPS://GALOIS.COM/BLOG/2023/09/THE-IMPACT-OF-PROVABLE-SECURITY-AWS-AND-SUPRANATIONAL/)

NEXT

Galois News

Galois Releases the Swanky Suite of Rust Libraries for Secure Computation (https://galois.com/news/galois-releases-the-swanky-suite-of-rust-libraries-for-secure-computation/)
PRESS RELEASE

Galois Releases CAMET Base Pack 1.6.1 with Enhanced Capabilities and Stability Improvements (https://galois.com/news/galois-releases-camet-base-pack-1-6-1-with-enhanced-capabilities-and-stability-improvements/)
PRESS RELEASE

(HTTPS://GALOIS.COM/NEWS/)

Portland, OR

421 SW 6th Avenue, Suite 300
Portland, Oregon 97204
(https://www.google.com/maps/place/Galois,+Inc./@45.520811,-122.678081,17z/data=!4m6!1m3!3m2!1s0x54950a04159ece0f:0x36857895c75e27d7!2sGalois,+Inc.!3m1!1s0x54950a0415

Arlington, VA

901 N Stuart Street, Suite 501
Arlington, Virginia 22203 (https://goo.gl/maps/pxFK95q48t32)

Minneapolis, MN

111 Third Avenue South, Suite 350
Minneapolis, MN 55401 (https://maps.app.goo.gl/csQrxYhmMPHDXLZJA)

Dayton, OH

444 E 2nd Street
Dayton, Ohio 45402 (https://goo.gl/maps/cRzPpKEF6eD2)

T 503.626.6616 (tel:15036266616)
F 503.350.0833

contact@galois.com (mailto:contact@galois.com)