**Julien Robert**

- Blog: [*https://oneirical.github.io/*](https://oneirical.github.io/)
- GitHub: [*https://github.com/oneirical*](https://github.com/oneirical)
- Reddit: [*https://old.reddit.com/user/oneirical/*](https://old.reddit.com/user/oneirical/)
- Zulip: [*https://rust-lang.zulipchat.com/#user/693959*](https://rust-lang.zulipchat.com/#user/693959)
- Discord: Oneirical
- Email: julien-robert@videotron.ca

# Rewriting Esoteric, Error-Prone Makefile Tests Using Robust Rust Features

**Project Size: Large**

## Synopsis

`tests/run-make` contains a heaping collection - 349 to be precise - of Makefiles containing Bash test scripts. These are designed to catch all kinds of potential errors and breaking changes that could sneak in after changes done to Rust's compiler and other utility tools. As an example, the test `core-no-fp-fmt-parse` verifies that the core library of Rust can be compiled without enabling support for formatting and parsing floating-point numbers.

The problem: Should one of these tests fail, Rust maintainers will naturally attempt to understand how these tests operate in order to fix their bugs. When opening the run-make folder, they will be met with a… rather esoteric and inscrutable wall of Bash syntax.

Rust contributors should be expected to contribute to Rust using their knowledge of Rust code, not ancient Bash hacks and workarounds. This project aims to rewrite these tests in robust, documented `rmake.rs` files, backed up by a supportive and easy to understand `run-make-support` support tool.

## The Problem

To showcase the importance of this project, allow me to demonstrate the affectionately named `branch-protection-check-IBT`:

```
all:
ifeq ($(filter x86,$(LLVM_COMPONENTS)),x86_64)
    $(RUSTC) --target x86_64-unknown-linux-gnu -Z cf-protection=branch -L$(TMPDIR) -C
link-args='-nostartfiles'  -C save-temps  ./main.rs -o $(TMPDIR)/rsmain
     readelf -nW $(TMPDIR)/rsmain | $(CGREP) -e ".note.gnu.property"
endif
```

Did you get all that? This test comes with no comment on its function beyond "Check for GNU Property Note". From what I understand, it conditionally compiles a Rust program for the x86_64 architecture with specific compiler and linker options, and then inspects the compiled binary to check for the presence of GNU-specific properties… Imagine being a contributor running into this test failing, and now needing to trudge through ancient documentation to understand what is happening here.

Information on *what these tests actually do* is sparse and not very informative. Test names are full of acronyms and only occasionally possess explanatory comments. Some of these tests also stretch out into the dozens of lines, only returning a generic statement of failure when an error is encountered and not indicating to contributors what caused the error.

There are also inconsistencies and unexpected behaviours, where oddities of the Windows operating system are accounted for using rough hacks and workarounds… And without Rust's robust type checking and error handling, any of these tests could very well assume everything is fine when things actually are not.

Take for example this `const-prop-lint` test, designed to verify that there are no object (.o) files left behind after the compilation, which could happen if the code generation process was interrupted due to an arithmetic overflow error.

```
all:

$(RUSTC) input.rs; test $$? -eq 1

ls *.o; test $$? -ne 0
```

Imagine a case where buggy code generation fails to create any files no matter what - this test will naturally pass, as it considers "no output at all" to be just as fine as "no .o files detected". Not to mention cases where "ls" could behave strangely with potential special characters…

## How To Fix It

Jieyou Xu, the mentor for this project, has already written *a minimalist version of the tool* that will let the Rust codebase swap these tests into `rmake.rs` files written in pure Rust. It can handle basic and most common Makefile functions, like passing arguments around and specifying compilation targets. However, any feature beyond barebones functionality has yet to be implemented.

Still, it is already possible to already rewrite the most basic tests. *I have submitted a pull request* porting one such test to Rust, `core-no-fp-fmt-parse`, as to familiarize myself with the function of the codebase.

```
all:
    $(RUSTC) --edition=2021 -Dwarnings --crate-type=rlib ../../../library/core/src/
lib.rs --cfg no_fp_fmt_parse
```

This was turned into:

```
// This test checks that the core library of Rust can be compiled
// without enabling support for formatting and parsing floating-point numbers.

fn main() {
    rustc()
        .edition("2021")
        .arg("-Dwarnings")
        .crate_type("rlib")
        .input("../../../library/core/src/lib.rs")
        .cfg("no_fp_fmt_parse")
        .run();
}
```

I could see a need for commenting specific lines in these Rust ports for the tests with more complex functionalities than these low-hanging fruit ones.

To write this, I studied *Abhay Jindal*'s pull request. I do not know if they still desire to participate in GSoC, as the conversation has been left untouched for the last 2 weeks as I am writing this. If I am assigned this project in GSoC, I would naturally finish the PR they started by adding the requested test documentation.

## Why A Large Project Size?

Functionalities added to `run-make-support` should follow the *“loosely-typed API design”* proposed by Jieyou & other Rust contributors. This *should* reduce the need for constant backtracking and rewriting already-ported tests - which an excessively rigid design could cause.

This project may therefore seem rather straightforward - tinker up some functions in `run-make-support`, taking advantage of Rust generics for flexibility, port a couple of tests every day, emerge victorious and adorned in glory at the end of the summer.

I doubt it will be that simple.

The Rust project became what it is today through its robust community review process. Merging pull requests is not a simple matter, as even a seemingly trivial test can contain heaps of traps and tricks.

*Here is an example shown to me by Jieyou.* This test is composed of only four lines of code - a job easily done, surely?

```
all:
  $(RUSTC) --crate-type=staticlib nonclike.rs
  $(CC) test.c $(call STATICLIB,nonclike) $(call OUT_EXE,test) \
    $(EXTRACFLAGS) $(EXTRACXXFLAGS)
  $(call RUN,test)
```

Unfortunately, this test makes use of a static library file, which looks like `libnonclike.a` on Unix and `nonclike.lib` on Windows. Jieyou utilizes this function to take this into account.

```
if target().contains("msvc") {
  format!("{name}.lib")
} else {
  format!("lib{name}.a")
}
```

However, at the time I am writing this, his PR is still failing `i686-msvc` compilation, which means oddities related to operating systems may yet remain.

I believe this example shows how **nightmarish** some of these tests can be. And this isn’t taking into account how conflicts could arise across multiple PRs, or how ported tests might *appear* to work, when they actually have slight differences with the old tests.

The main way to check parity with former behaviour is through a `tools.mk` file with even *more* obscure Makefile syntax.

Finally, certain tests are so minimalist they might not deserve being `run-make` tests at all. `run-make` tests *execute* their compiled programs, while a different test category, `run-pass`, only cares about the compilation process. Therefore, it is worth judging the easier tests among the lot, and deciding if they should be moved away from `run-make` all-together.

**For these reasons**, I cannot predict for certain what percentage of all tests I will succeed in porting. 100% is obviously unreasonable. Teapot, the previously assigned mentor for this project, said that something in the realm of 50% would be ideal - but this still implies a *boatload* of testing, refining and debugging. 30 tests is a good “absolute minimum” - the final result should be somewhere in between 30 and 175.

So, to tip the scales in my favour, I will commit, before starting the true porting exercise, to reviewing every test and classifying them in terms of difficulty and what problems they may cause in the construction of `run-make-support`. See the **Project Timeline**.

My project objective is therefore not “port X% of tests”, but rather “port all Easy-tier tests, some harder tests, and ensure that the path is paved appropriately and that the most painful inconsistencies are studied to make the task easier for future maintainers.”

I may perhaps become one such “future maintainer” beyond GSoC!

These complications make this project a truly serious undertaking, which I believe renders it worthy of the Large tier.

## Statement of Commitment

I chose this project because:

- There is a clear difficulty ramp - some tests are very easy to understand, others not so much. It will be educational to do work that scales linearly in difficulty throughout the summer as opposed to being thrust into the deep end of a highly arcane and massive part of the codebase.
- It is easy and rewarding to track progress - the number of ported tests can directly be quantified, which is the complete opposite of, say, the Cryptography Rust project proposal where the entire summer could be taken to study the algorithm and finally write down a comparably small, but very thoughtful set of lines of code implementing it.
- I enjoy the “Rewrite it in Rust” meme when it genuinely has advantages and showcases the strengths of Rust. In this case, robust error handling and strong static typing will give huge advantages over Makefile scripts - allowing contributors to understand better why tests fails and how to fix their contributions.
- I already have some experience rewriting flawed, old code in Rust. See the **About Me** section!
- The *mentor’s GitHub profile picture is adorable :3*

At the time of writing this, I am planning to take an elective course over the summer, Introduction to Macroeconomics, a commitment of 3-5 hours per week. I believe this will not affect my capabilities of giving it my all on a GSoC project in a full-time schedule.

As mentioned, I have shown my ability to work with this project in *a pull request* porting a first test and editing the support API with new helper functions. At the time of writing, it has been approved by Jieyou and is awaiting leadership confirmation for merging.

## Project Timeline

### Community Bonding Period (May 1 - May 26)

All tests will be classified under 3 difficulty levels:

- **Easy** tests require very minor alterations of the `run-make-support` tool, and do not require deep documentation dives to be understood. They may already have comments or not have many flaws. The number of tests in this category could be small.
- **Medium** tests may imply the creation of significant new functionality in `run-make-support`, or may be using some harder to understand features of Makefiles. They might contain flaws and possible false positives (such as the example I gave for `const-prop-lint`) that need to be accounted for with Rust’s error handling. This will likely be the largest category.
- **Hard** tests are arcane and reliant on very niche functionality. They may require major new features in `run-make-support`, be profoundly cursed, and require discussion with my mentor and extensive documentation research. Completing a Hard test will likely pave the way forwards and make porting of future tests much easier.

Some low hanging fruit Easy-tier tests will be ported to get up to speed. The goal: getting at least one pull request merged and fully understanding the CI process and contributor guidelines of Rust.

I am also excited to get to know the members of Rust GSoC and be in the heart of one of the biggest open source projects out there, known and trusted by industry professionals and eccentric hobbyists alike. I'll journal my experience and personal discoveries in my *blog*.

Deliverables:

- A difficulty classification of all tests, subject to change, with some detail on the challenges certain tests may pose.
- At minimum one test ported and its associated PR merged.
- A blog post detailing the project and my experience so far.

**May 27 - July 12**

The heart of the project, and likely where the highest amount of tests will be ported. **Easy** and **Medium** tests will be the bread and butter of this period as I reinforce `run-make-support` to support additional features, and diagnose potential false positives, false negatives and unexpected behaviours - both those that already exist as a consequence of the Makefile implementations, and those that could arise as a consequence of the Rust port.

To get to understand the project at a deeper level, I may try my hand towards the end of June in the port of a **Hard** test. However, at the end of this period, all **Easy** tests should be completed.

Weeks 1-3 Deliverables:

- Significant progress towards the completion of Easy tests, some consideration of at least one Medium test.
- Strengthening of `run-make-support` with robust helper functions which take into account compatibility issues with different operating systems and compilers.

Weeks 3-6 Deliverables:

- Completion of Easy tests. Note that some tests may have their difficulty category swapped, within reason.
- At least one Medium test completed.
- Refinement of ported tests to ensure parity with old Makefile function and consistent behaviour across platforms.

The goal of this period is not reaching the completion of an arbitrary number of tests, but rather ensuring the establishment of a robust testing protocol with completed examples which maintain the safety and solidity values of Rust.

**July 12 - August 19**

Documentation of both ported tests and the `run-make-support` tool will be strengthened, with the goal of no longer confusing contributors who have failed `run-make` tests and are wondering why. **Medium** and **Hard** tests will receive the bulk of my efforts - as a result, the rate of time spent per ported test may increase.

Additionally, *this page of the Rust developer guide* will be updated to reflect the changes to run-make tests.

Weeks 7-9 Deliverables:

- Usage of established testing protocol to progress towards the completion of a Hard test with careful consideration of the subtle ways in which functionality could be broken.
- Completion of at least one other Medium test.

Weeks 9-12 Deliverables:

- Completion of a Hard test, with the goal of setting a precedent highlighting some of the biggest traps and inconsistencies which other Hard tests may share.
- Overall, at minimum 30 tests completed in a robust, idiomatic and stable fashion.
- Update of Rust developer guide to reflect completed work.
- Documentation on the encountered pitfalls throughout this project, so that future maintainers can avoid them should they port some of the remaining tests.

**August 19 - August 26**
A final write-up on all completed work, and potential future steps, will be submitted to GSoC. I will personally write a more informal review of my learning and enjoyment of the project on my blog.

## About Me

I am a 21 years old, final year undergraduate at Concordia University (Québec, Canada), in Systems and Information Biology, a program which no one has ever heard about. With each passing day, I increasingly suspect I might be the only one following it. At the intersection of computer science and biology, it is "officially" designed to prepare me for an academic career in bioinformatics.

Unofficially, however, I grew tired of academic-style script programming in high level languages… The field has little regard for stability and maintainability, as well as constant abstraction of lower-level computer science concepts. I felt less like a programmer, and more like someone stitching together magical black boxes of code and marvelling at the pretty graphs coming out.

It's only when I started seriously learning Rust that I realized the truth - a career in biological research was not the future I wanted for myself. Rust was, for me, the ideal bridge between the high and the low level - guiding me down and down the rabbit hole, turning my love for dynamic typing into disdain, all while holding my hand through the watchful attention of its rigid compiler. Not as daunting as C++ and not as verbose as Java, Rust made me realize that I like solving puzzles on my computer, and that it is what I want to do with my career.

With Rust, I have built the following:

- *A rewrite from C++ to Rust of RGBFIX*, a command line interface program designed to correct file corruption and broken headers in Game Boy ROM files. Part of the greater *RGBDS* project, this 1-week contribution introduced me to the world of open source and had me work with a fantastic mentor and reviewer, *Evie*!
- *The program that won me the CS Games 2024's Operating Systems category*, a server which receives packets from a file crawler, parses their hexadecimal data and writes files into the target OS, with a strong focus on robust error handling - as not all packets are guaranteed to be without errors! This challenge had a 3 hour time limit, and you may read about the *competition itself here*. I believe I owe a major part of my victory to Rust's robustness - circumstances made it very difficult for each contender to test their code, and Rust's "if it compiles, it works" pseudo-guarantee essentially secured success.
- *An experiment with neuroevolution in Bevy*, where I tested the ability of independent agents to adjust their neural networks' weights across many generations through a genetic algorithm, and eventually solve simple tasks like painting as many walls as possible in a 2D environment. It also takes advantage of Rust's signature "Fearless Concurrency", as it uses Bevy's ability to concurrently show the user a display of the current generation while training future generations in the background.
- *A small Bevy puzzle game made in 48 hours*, and currently a *much bigger traditional roguelike game, also in Bevy*. Speaking of Bevy, *I wrote a blog post* outlining some of the biggest roadblocks I

encountered working with it across these 3 projects, and *got some amazing feedback* from the community.

I live in the province of Québec, in Canada, and my timezone is EST. According to his GitHub, the mentor for my proposed project, Jieyou, is 4 hours ahead of me.

## Conclusion

Some may say that rewriting infrastructure is "boring" when other organizations are out there suggesting to GSoC contributors AI-powered technowizardry exploiting the latest trends.

Personally, I wouldn't have it any other way. Rust is, to me, a place of passion for sturdy technology, accepting anyone who can build things no matter how many academic titles they have after their name. Whenever I see any developer using Rust, I immediately assume they engrossed themselves in the tech sector for reasons reaching further than just cold hard cash.

I'd be truly honoured to give back after all the awesome projects Rust enabled me to create.