RiiR Considered Harmful: Just Bind to C Libraries

Preston Carpenter

1. Abstract

There is a trend in the Rust community to rewrite popular C libraries in Rust. For sufficiently complicated libraries this is very difficult to do as well as the original. Even if it is a success it will split the ecosystem: there must be a choice for developers to contribute to either the original C library or the Rust rewrite. Instead, safe bindings to the C library should be made which will take much less effort and allow the Rust community to share in the successes of the larger programming community.

2. Introduction

There is little doubt that Rust, in its current form, has reached a sufficient mass to be considered "mainstream". Firefox is actively being rewritten using Rust with major subsystems such as Servo already being used successfully in production. The language has also garnered attention from some of the industry's biggest players: Google[1], Facebook[2], and others. Some companies have even bet their future on the language, the most notable being Chucklefish which is using Rust to write their next game.[3]

However, this shift into the mainstream now requires a shift in project management. Before, the primary goal was to spread Rust as far as it can and be as accessible to everybody by being overly inclusive. This work is still ongoing, but the main focus should now shift to stewardship over the existing project. This means changing some processes that have been in place for a while. Though the Rust 2019 Roadmap has not yet been published multiple call-in blog posts have called for this necessary change.[4][5][6] The Rust community is now ready for a change in how the project is governed.

One aspect that has been overlooked so far is the philosophy behind "RiiR" (Rewrite it in Rust). It has generally been ignored outside of mild admonishment in the forums when its advocacy strays too close to breaking the "No Zealotry" rule. The most extreme cases of this are when random projects are harassed to rewrite their entire project in Rust simply because it's "better". This type of zealotry has been properly excluded and is not part of the average Rustacean's habitus.

There is, however, a more subtle form of this RiiR philosophy that has not been dealt with which threatens the stability of the community. This is the case where, instead of bothering other projects to rewrite their code in Rust, Rust programmers go ahead and take on the work themselves as a separate project. In isolation, this is not an issue as programmers are free to work on what they like in the open source community. However, in aggregate this encourages RiiR as a solution when interfacing with existing ecosystems is too difficult.

3. Rewrite it in Rust vs Rewrite it for Rust

Before continuing, a distinction must be made between rewriting a library *in* Rust vs rewriting *for* Rust.

The former is a purely technical statement regarding implementation level details of the library itself. Like existing C libraries, it should not matter to consumers of the library what the language is written in so long as it can be interfaced with in a standard way. This standard way happens to be the C calling convention, but since it's such a simple convention any language can expose it as an interface.

However, rewriting a project *for* Rust implies that this decision was made because of a shortcoming in Rust to interoperate with a system that is not its own. If a complex, but well-designed, library is written in C then it is desirable to be able to use it in Rust. Technically speaking, this is always possible to do thanks to Rust's FFI system. However, in order to encode the library's interface in a memory safe way at scale immense effort might need to be undertaken. Because of this effort many would-be binding writers instead opt to rewrite the complex library in Rust.

4. An argument for C bindings

Other languages routinely use bindings to gain the benefits of C libraries without the added cost of reimplementing the functionality in the host language. A prime example of this is the Python ML library Tensorflow.[7]

The argument can be made that languages like Python need to rely on bindings because C provides something that those languages do not have, usually speed of execution, whereas Rust has all the benefits of C plus more (namely safety and ergonomics).

However, even if a RiiR library could be strictly better than its C counterpart the amount of time and effort that must be expended is immense. This effort can be worth it, but it should only come after a C library fails to be either fast, fully featured, or well designed enough. As much effort as it is to write C bindings for Rust it is much, much easier to do so than redoing sometimes several hundred years worth of work to end up in nearly the same place.

With that said, writing C bindings is still difficult. It takes a different style of programming to make successful C bindings. The purpose of the bindings are to be sound, but that can be difficult to do in a way that is performant and easy to use.

5. Difficulties in writing C bindings

There are two major difficulties that are encountered when writing C bindings: wrapping the library safely and keeping the API ergonomic while keeping it safe.

The first is ensuring that the bindings are memory safe. Unlike other languages, this means that using the C library it must not be possible to do break any of the memory usage rules used to encode the safe Rust memory model. Though Rust gives you the tools to accomplish this, the work is both repetitive and difficult. For simple APIs it might be sufficient to use a simple abstraction that fits well into the ownership model. However, most APIs are anything but simple and require more complicated mechanisms to ensure memory is not misused.

The second problem is how to balance this safety with ergonomics. Ergonomics in this case refers to how easy it is to use the library as a user. In certain cases, the ergonomics of the Rust bindings can be better than the C bindings. This is thanks to the stricter type system and more explicit memory management. Combined, these two systems can make it apparent how the library is supposed to be used and make it very difficult, or even impossible, to misuse it.

With these difficulties in mind, it is easy to see why it is more interesting from a technical perspective to RiiR a library in Rust rather than writing bindings. Writing bindings is error-prone, completely manual work that often goes unrewarded especially when compared to the efforts put into the library itself. Safety never has to be considered if the library is written entirely in safe Rust and all effort can instead be placed in creating a useful and ergonomic library. In the binding author's world instead of providing a new, fully legitimate service using Rust in a potentially original setting the work is reduced to the most bureaucratic of work: memory management.

6. Smithay: A case study in rewriting Wayland for Rust

In the Wayland ecosystem there has been a rise in framework libraries for smaller compositors to take advantage of common code that all compositors must ultimately implement to be useful. Usually this code is low-level code to acquire resources from the kernel or to perform an action that all compositors would ultimately want to do (e.g. implementing copy-and-paste between windows). Of the frameworks only the Rust-written Smithay is not written in C. It is also the least complete.

Part of the reason it is so incomplete is that Smithay has required that the authors RiiR Wayland in Rust. Their library, wayland-rs, started out as bindings to the Wayland client and server libraries. It still offers C bindings however it also offers an option to use a version they made entirely in Rust. This version comes with a significant downside: it cannot interact with any C library that uses Wayland:

> The people who have already discussed this matter know that abandoning the C wayland libraries is really cutting oneself from many interactions. It is most notably required by OpenGL, but also any other C library that you'd want to use and that interacts directly with wayland like GStreamer for example.[8, paragraph 2]

Despite this massive limitation this is the default version used by wayland-rs. The authors are willingly cutting themselves off from the entire existing ecosystem so that their implementation is "pure" Rust.

What language a program is written in doesn't matter to users, so long as the program is reasonably performant, well designed, and doesn't crash too often. By making "written in Rust" a selling point it conflates implementation details with actual perceivable features. Having less memory vulnerabilities is a good thing, and Rust definitely provides that. However, there are other considerations that must be made when choosing a program. A program or library that is 100% memory sound, but cannot interoperate with the rest of the ecosystem, is useless.

7. wlroots-rs: using Rust with Wayland

Currently, the most popular Wayland framework is wlroots. It is used by a many small, upcoming compositors and has bindings in several languages.[9]

wlroots-rs, the Rust bindings for wlroots written by me, does not aim to rewrite wlroots but instead write safe, ergonomic bindings to the library. wlroots is very complicated and very big; it has over 60,000 lines of C code[10] and is steadily growing. Rewriting it would be a huge undertaking that would not be worth the effort, so bindings are the obvious solution.

Only part of the library has been wrapped safely in Rust and the effort required to do so was immense. It took a non-trivial amount of time to invent a safe API using the type system and borrow-checker that didn't allow unsafe use of the pointers exposed by the library. It took almost as long tweaking the API to make it ergonomic to use. Despite how difficult it was to do, wlroots-rs did not require reimplementing any part of wlroots and continues to benefit bug fixes, security patches, and new features that are added to wlroots. Many of these features come nearly for free as the safety framework that is already in place makes it almost trivial to integrate these new features into the Rust bindings.

wlroots-rs, despite having less effort put into it compared to Smithay, has more features and is more usable today because it relied on a well tested, well designed C library. Working with the existing ecosystem, even if it meant giving up the "purity" of being written entirely in Rust, led to a better engineered library.

8. Conclusion

Each language frequently reinvents the wheel when it comes to popular libraries. This is understandable if existing implementations were good enough then the language wouldn't exist in the first place so a certain amount of rewriting and reimplementing of old ideas with new styles is expected.

For all languages though, there comes a point where programmers realize they need to throw their hands in the air and accept that they are unable or unwilling to reimplement an API. For Rust that point seemingly hasn't been reached in the community and seemingly never will be. Though it's difficult to disengage the idea that all C code is worse since Rust aims to replace it, it is bad engineering to not admit that at a certain point it is impractical to rewrite the world just for the sake of it.

It is critical that this attitude changes in the community and the best way this can be done is the leadership setting an example by curbing this rampant meme. It requires a massive shift in the community policies and will potentially alienated current members. However, like the would-be-members that don't engage because they disagree with the code of conduct, it is better to not have members that would reject basic engineering practices and instead resort to zealotry.

References

[1] "chromiumos/platform/crosvm - Git at Google", Chromium.googlesource.com, 2019. [Online]. Available: https://chromium.googlesource.com/chromiumos/platform/crosvm/. [Accessed: 10- Mar- 2019].

[2] "facebookexperimental/mononoke", GitHub, 2019. [Online]. Available: https://github.com/facebookexperimental/mononoke. [Accessed: 10- Mar- 2019].

[3] The Rust Project Developers, "Chucklefish Taps Rust to Bring Safe Concurrency to Video Games", 2018.

[4] J. Turner, "The Fallow Year, my Rust2019 post", 2018. .

[5] G. Hoare, "Rust 2019 and beyond: limits to (some) growth.", 2019. .

[6] S. Klabanik, "Thoughts on Rust in 2019", 2018. .

[7] https://www.tensorflow.org/ Tensorflow. Google, 2019.

[8] V. Berger, "Wayland-rs 0.21: Pure rust implementation", Smithay Project, 2018. .

[9] "Projects which use wlroots", GitHub. [Online]. Available: https://github.com/swaywm/wlroots/wiki/Projects-which-use-wlroots. [Accessed: 10- Mar- 2019].

[10] "swaywm/wlroots", GitHub, 2019. [Online]. Available: https://github.com/swaywm/wlroots/blob/master/README.md. [Accessed: 10- Mar- 2019].