The Soundness Pledge

Jan 18, 2020

Lately there has been considerable drama around Actix-web, for which I'll point to Steve Klabnik's A sad day for Rust to explain. This post is an opportunity to share some thoughts I've had about soundness, Rust, and open source community.

I believe one of the most important contributions of Rust is the *cultural* ideal of perfect soundness: that code using a sound library, no matter how devious, is unable to trigger undefined behavior (which is often thought of in terms of crashes but can be far more insidious). Any deviation from this is a bug. The Rust language itself clearly subscribes to this ideal, even as it sometimes falls short of attaining it (at this writing, there are 44 I-unsound bugs, the oldest of which is more than 6 years old).

Many in the Rust community subscribe to this, but some don't. I'm *not* arguing that one is right and the other is wrong. Rather, the main argument in this post will be that there is a need for better communication, to make clear one's thinking on this point. I believe many of the problems in the Actix drama were caused by miscommunication and a mismatch in expectations.

Unsoundness is bad, unsafe is a sharp tool

I think David Tolnay's blog post on soundness bugs draws a very important distinction between "unsafe" as it is used in Rust, and "unsound". Essentially, "sound" means that all possible uses of a library avoid undefined behavior. This is quite a high standard, and in many cases goes beyond what's necessary to deliver working code. Even so, it is the standard that the Rust language (including standard library) and many libraries aspire to.

The `unsafe` keyword has a specific meaning: it is a sign that more reasoning is needed to prove that use of the code is safe. Outside unsafe blocks, the compiler essentially uses information encoded in the type system to prove that use of the code is safe. Inside unsafe blocks, certain things are permitted that are ordinarily prohibited, for example reading and writing of raw pointers. For readers who don't know Rust well, I recommend reading the Unsafe Rust chapter of the Rust book; otherwise a lot of this discussion is likely to be confusing.

There's a tendency in some parts of the Rust community to consider `unsafe` itself as bad, but I think that's missing some subtlety. Certainly in code that could just as easily be

written in safe Rust, using `unsafe` for some perceived performance gain, or, worse, as a way of silencing complaints from the borrow checker, is a bad sign. But for many uses, especially integration with libraries or runtimes designed for other languages, it is essential, and it's important to use it well. Other valid use cases include SIMD and the implementation of foundational data structures; the Rust standard library has a good collection, but is not intended to be comprehensive of all possible use cases.

Sometimes it's fairly easy to avoid soundness bugs: just don't use "unsafe" anywhere in the code, and don't depend on any other library with a soundness bug. For certain classes of problems, this is practical. Often it's framed as a tradeoff between performance and safety, but I haven't seen a lot of evidence that this is really the tradeoff. In most cases I've seen, Rust gives you the tools to achieve both, but it sometimes *does* require extra work. Many of the safety guarantees in Rust come at no runtime cost. A few others (notably array bounds checks) have some cost, but ordinary performance tuning techniques are often effective even within the safe fragment. One data point of evidence for this is pulldown-cmark, which now has best-in-class performance, but no use of unsafe (there is an optional SIMD optimization, but performance is hardly different than the default configuration, thanks to extensive tuning by Marcus Klaas de Vries).

The right way to think of `unsafe` is as a sharp tool: to be treated with respect, wielded by those who have some knowledge about how to do it well, and not to be overused. We're *not* aiming for a world in which scissors have been replaced by those plastic things that can't actually cut paper.

Binding with complex runtimes

Some applications of Rust are binaries where all of the code is Rust, or perhaps there are a couple of C or C++ libraries for specific functions. For these, reasoning about soundness is reasonably straightforward.

Another interest class is binding Rust to existing complex runtimes, in which I include (scripting) languages, UI toolkits, and other related things; I believe the soundness issues are similar.

For binding with a scripting language, a reasonable definition of soundness is that no code written in that language should be able to trigger UB in Rust code. This is quite difficult to acheive in practice, because languages are rarely designed with soundness in mind. A good example is rlua, the README of which contains a good discussion of the safety goals and some indications how hard they are to achieve. Another good snapshot of an inherently difficult soundness issue is PyO3#687. These links show significant effort and thoughtfulness of how to achieve soundness in a difficult environment.

The story of rlua raises a very interesting point: the Lua language was fundamentally not designed with safety in mind, and allows a number of operations that are inherently quite

dangerous. There's really only so far a wrapper can go in trying to ensure soundness. The author's more recent thinking is to expose the `Lua::new` constructor as `unsafe` and make it clear to the client that they are ultimately responsible for not loading Lua code that can provoke safety problems. This seems a reasonable approach, and I think speaks to a general principle: when it is not possible to *guarantee* soundness, it is often best to expose the functionality through an unsafe interface.

The same principle underlies the philosophical differences between the Ash and vulkano crates for providing Vulkan wrappers. The Vulkan API is fundamentally unsafe, and while vulkano goes to great lengths to try to wrap that and provide a safe interface, that creates a number of compromises, which is problematic because the point of Vulkan is performance. For more context, see this description of design decisions for vulkano and this Reddit discussion presenting Ash motivations. While both approaches are reasonable, of the two Vulkano has found it considerably more difficult to deliver on its promises, with persistent safety issues in spite of its stated goals, and other compromises that most Rust graphics programmers I've talked to find unappealing.

A closely related issue relevant to my own work is providing safe bindings for platform UI capabilities. For macOS, this is roughly the same as Objective-C language bindings. For Windows, it involves a great deal of wrapping of COM objects. Microsoft has a good blog post detailing some of their own work and the challenges: Designing a COM library for Rust. Another detailed argument is a github issue entitled COM trait methods should probably always be unsafe. That is perhaps overly pessimistic, but speaks to the difficulty of the problem.

I'll give a simple illustrative example of how safely wrapping a COM interface can be tricky: ID3DBlob, which is used to hold the results of shader compilation, among other things. The underlying COM interface has two methods, which expose a length and a pointer to the blob contents. An obvious interface would be something like this:

```
#[derive(Clone)]
struct D3DBlob(ComPtr<ID3DBlob>);

impl D3DBlob {
    fn get_mut(&mut self) -> &mut [u8] {
        unsafe {
            slice::from_raw_parts_mut(self.0.GetBufferPointer(), self.
        }
    }
}
```

Can you spot the soundness issue? It's the fact that it's possible both to clone and to get mutable access, therefore you can get two mutable references to the same contents. Leaving out one or the other would restore soundness, but both of these operations make

sense - you want a mutable reference when filling a buffer from some source, and you might want to clone to distribute the buffer to multiple threads.

A sophisticated approach to wrapping might use a form of session type, like this:

```
struct D3DBlobMutable(ComPtr<ID3DBlob>);

#[derive(Clone)]
struct D3DBlob(ComPtr<ID3DBlob>);

impl D3DBlobMutable {
    fn freeze(self) -> D3DBlob {
        D3DBlob(self.0)
    }

    fn get_mut(&mut self) -> &mut [u8] { ... }
}

impl D3DBlob {
    fn get(&self) -> &[u8] { ... }
}
```

This is a simple example. I've come across a lot more interesting stuff, involving temporary objects that have an implicit lifetime dependency, callbacks with interesting pointer lifetime requirements, and more. Many of these have resulted in unsoundness in published "safe" wrapper crates.

Because of the difficulties I've outlined above, my own preferred style is increasingly to wrap platform functions using an *unsafe* API, then verify that uses of these functions are conservative. In druid in particular, the druid-shell layer (which abstracts platform capabilities but is not a simple wrapper) has significant amounts of unsafe code, while the druid layer above it (UI logic) has absolutely none. I don't know of a better way to manage this.

Mitigating unsoundness

There are a combination of practices that can be used to mitigate unsoundness. Many of these are borrowed from the C++ world, where effectively every line of code is unsafe.

The first line of defense is code review. Here, the `unsafe` keyword helps because it narrows the focus onto sections of code that are potentially unsound. A codebase with a few small, isolated `unsafe` sections is much easier to audit than one which uses it pervasively. There are tireless volunteers in the Rust community, most notably Shnatsel, who analyze code for unsoundness and propose fixes.

There are also a number of tools available, including sanitizers and MIRI. The latter is especially good at reporting "technical" violations, as it has a deeper understanding of Rust semantics than the LLVM-based tools. However, these tools can only report violations on code paths that trigger them, and many of the soundness violations occur on exceptional paths. For these, fuzzing can help.

Even so, the ideal of soundness covers *all* uses, no matter how devious. It is possible, even likely, that dynamically testing running code will fail to uncover potential soundness issues that could arise from using the code in ways likely not anticipated by the library author. In fact, for these use cases there is likely some controversy about whether fixing the soundness issues is even worthwhile.

An exciting research area for the future is formal techniques for proving Rust programs correct, such as Ralf Jung's work. This is most likely to yield positive results for lower level data structures; it should finally be possible to implement a doubly-linked list efficiently and without anxiety.

Dependencies

Unfortunately, the soundness of a crate is really the worst of the crate itself and all transitive dependencies. Further, the number of transitive dependencies can become quite large; a typical application will have several hundred. There are a number of efforts to try to audit dependencies for various issues, including the use of unsafe but also license compatibility, etc. The crev project is particularly interesting, as it focuses on human review, but cryptographically strong verification that the review is what it claims to be.

The pledge

I propose the following as the basic form of the soundness pledge:

"The intent of this crate is to be free of soundness bugs. The developers will do their best to avoid them, and welcome help in analyzing and fixing them."

I think many, if not most, Rust libraries would subscribe to such a pledge, but also that there is a sizable minority who would not. And that's fine. It's definitely a tradeoff, and there is *effort* involved in achieving these goals. Or, perhaps the library author doesn't agree with my assertion that it's possible to attain top-tier performance using only safe Rust. Most Rust libraries are provided with as a volunteer effort.

Any library taking this pledge would basically need to ensure that all its dependencies have done so, or establish an equivalent level of trust through other means.

As mentioned above, I think it's fine for a library *not* to subscribe to this pledge. Then, third-party auditors will know not to waste their time seeking soundness problems, and

potential users will know what to expect. Both those who value performance over safety, and safety over performance, will see a clear statement of priorities that can inform their decision whether to take on the dependency.

Skin in the game

Higher levels of the soundness pledge might involve some form of commitment of effort to attain the goal: independent review, extensive use of fuzzers and sanitizers, and, hopefully at some point, formal proofs of soundness.

This is a whole 'nother discussion, but I think these higher commitments of effort should come with some kind of material incentive. Though there are exceptions, most open source Rust development is done as a volunteer effort, a labor of love. It is not fun responding to reports of vulnerabilities in one's code, or having a feeling of responsibility for security issues in users of the code. I think there are opportunities here to make the incentives match up more closely with the value provided.

Implications outside Rust

I've been talking extensively about Rust here for obvious reasons, but I believe reasoning about soundness and safety is valuable in other languages as well. Some other language cultures are way ahead of the game, particularly Java and JVM languages (provided you don't do JNI, ugh!), as the language itself provides very strong soundness guarantees, even in the face of race conditions and the like.

It *is* possible to write safe C and C++ code. It just takes a *lot* of effort, using the tools I've outlined above such as sanitizers. Certain codebases might already be considered honorary pledges, I'm thinking SQLite, as their testing methodology in particular is likely to catch most of the soundness bugs that motivate the use of safe Rust.

But ultimately I think some of the most productive interactions around soundness may result from the attempt to write safe Rust wrappers for other runtimes. I have to wonder how different Vulkan might be if it had been informed by an effort to build safe Rust wrappers without significant runtime costs, and how much that might have made it easier to base WebGPU on it.

Similarly for COM. Many soundness issues are discussed in informal documentation, such as whether a method is "thread-safe" or not. The Rust concepts of Send and Sync are much more precise ways of talking about behavior in multithreaded concepts, and together with questions of whether `Clone` is allowed and which methods are `&self` vs `&mut self`, plus the cases where it's impossible to avoid the need for `unsafe`, the type signature for a Rust wrapper tells you a *lot* about how to safely use a COM interface. Just doing an audit of various COM libraries to document the corresponding safe Rust types

would be a major effort, but one that I think could pay off.

Conclusion

While almost everyone would agree that soundness is better than unsoundness, different people will have different priorities for how important it is, especially as a tradeoff for coding effort and performance. Most of the Rust community highly values soundness, but even within the community there is significant variation. I've proposed a "pledge," a statement of intent really, which I hope could be used to clearly communicate the level of priority for Rust projects.

Right now, I'm inviting discussion rather than proposing it as a formal badge. Perhaps it's not really a good idea, or has downsides I don't foresee, or perhaps the idea needs tweaking. But I think it's worth talking about.

Discussion on /r/rust and Lobsters.

Raph Levien's blog

Raph Levien's blog
raph.levien@gmail.com

raphlinus
raph

Blog of Raph Levien.