My negative views on Rust

This is a little summary of my current thoughts on Rust. I wonder whether I'll look back in five years and see if my views have changed.

The Good

Rust's macros are very good. They act like Lisp's macros, unlike Haskell's.

The fact that Rust has type-classes ("traits") and sum types ("enums") and pattern matching is very attractive. Making orphan instances package-wide rather than module-wide is a very good decision.

I also like its treatment of records.

Its standard library does a few things well, like handling strings as UTF-8.

Distinguishing mutability has its advantages, it's easy to see that a function is "morally" pure–if not actually pure–and that's good for reading.

Take Sugar?

Rust's use of magical sugar constructs, where the compiler will automatically insert dereferences and copies and clones and drops for you has an initial appealing "it's all simple underneath" quality to it, but in practice this leads to bad compile errors: The worst kind of compile error is the one where the compiler is complaining about something that it generated for you, rather than something you explicitly wrote.

This can be compared with Haskell's use of monads, that provide syntactic sugar. The more magic you introduce, the harder it is for newbies to learn and to talk about.

Fetishization of Efficient Memory Representation

I've watched people on calls that are a couple years into Rust spend 20 minutes attempting to understand why their perfectly reasonable code doesn't fit into Rust's tight memory restrictions.

I've also been told, by people with white in their hair, with an air of misty-eyed revelation, that once you "get" Rust's memory model of the stack and the heap,[1] that things just all fit together wonderfully.

This touches on another topic I'd like to write about elsewhere: the difference between practice and theory and how users of languages like Rust and Haskell that make big promises and require big sacrifices don't seem to distinguish the difference.

It's not the technology that's working poorly, it's that you're using it wrongly.

In practice, people just want to be able to write a tree-like type without having to play Chess against the compiler. I predict that garbage collectors will become popular in Rust eventually.

This is both Rust's main goal–be like C, but safe–and also its main downside. People waste time on trivialities that will never make a difference.

Unsafe

The use of unsafe is a little disturbing, because every library features it. But it's not much different to using an FFI. I don't see this is a big downside.

The Rewrite Fallacy

I see a lot of "we rewrote X in Rust and it got faster" posts. I think that if you rewrite anything from scratch with performance in mind, you'll see a significant performance improvement. I'm suspicious of how much Rust itself is needed versus the developers having some performance discipline.

The "Friendly" Community

All new language communities are nice. When things don't matter much, people have no reason to get mad.

As soon as people have a stake in something, that's when things heat up and tempers come out. You get a stake in a programming language by writing a lot of code in it, or by building a business on it. When you have a stake in how a language works, you're highly sensitive to changes that will make you do more work than needed, or will limit your goals.

I've seen this play out for Haskell. Around 2007, when I started with Haskell, the community was friendly as anything, evangelic, open. People praised it for this. Everyone just felt blessed to be able to use this exciting language. Today, it's more like any other community. What happened? People started using Haskell for real, that's all.

Rust is going through the same thing, much more rapidly. Is it a reason to avoid Rust? No. But a "nice" community isn't a reason to join an upcoming language, either.

Async is highly problematic

Rust's choice to exclude a runtime/scheduler blessed and built-in to the language means they had to develop alternative strategies in the language itself. This is not turning out well.

Coloured Functions is a powerful metaphor for the incompatibility between async and synchronous functions and the awkward situations that mixing them introduces. Some blog posts have attempted to downplay the situation by focusing on technical aspects of specific syntactic forms. That doesn't really matter, though, because the reality is much simpler: Async code is easier to use when dependencies are in an async form.

People will choose libraries that are async over libraries that are not. However, maintainers that have written good, maintained code, are also resistant to adopt async.

- One conversation I've overhead:

    > Person 1: Another difference between diesel and sqlx is that diesel is not async yet and from looking at the issues it doesn't seem to be priority yet.
    >
    > Person 2: That sounds like a major issue

As the saying goes, the proof of the pudding is in the eating of the pudding.

Async introduces long, heated discussions.

The problem for Rust is that its users want a runtime, but want the option of not having one. The result is a mess.

Generally, I think Go, Erlang and Haskell made the better choice here. A garbage collector and green threads make programmers more productive.

When combined with iterators, I think understanding such code is quite difficult.

As a general purpose language

I feel like Rust is self-defined as a "systems" language, but it's being used to write web apps and command-line tools and all sorts of things.

This is a little disappointing, but also predictable: the more successful your language, the more people will use your language for things it wasn't intended for.

Conclusions, if any

I won't be using Rust for any of my own personal projects. But it's used at my job, so I'm sort of forced to have an opinion about it.

But I wouldn't mind using it as a replacement for single-threaded C if I just use the standard library. That would be fun.

---

1. Having done my fair share of C code, there's nothing new here for me.↩

Read more posts →