Written by Piotr Sarnacki
on June 3, 2022

(async) Rust doesn't have to be hard

Comment on: HN | Reddit

An article titled Rust Is Hard, Or: The Misery of Mainstream Programming came out today and it's getting a lot of attention. I have a feeling that it's viewed in a wrong context, so I'd like to comment on the issue raised there.

A lot of the comments to the article are in a tone of "this is precisely the reason why I don't learn Rust, it's just too hard" or "just don't use async, it's too hard, otherwise Rust is not that bad". So I'll start with a note for all the people intimidated by the techniques the author is trying to use in the post: when writing Rust code you almost never use this kind of stuff. Especially when you're writing applications, as opposed to libraries. And especially if you're not writing anything that is traditionally systems programming, like a database or an operating system (and yes, Rust is great for non systems programming too). I really wish the author clearly pointed out that they write the article from a point of view of a library author trying to come up with generic and flexible APIs. Then maybe I wouldn't feel the need to write a rebuttal and shake my head in disbelief reading some of the comments on HN or Reddit. But here we are, I guess.

I've been coding in Rust for a few years now and although I haven't done any advanced stuff, I wrote quite a bit of async code. In fact I learned Rust doing async, I wrote pretty much only async code when using Rust at work (unfortunately only for a few months) and I still write mostly async code. In the article I'm responding to you can read:

> “Fearless concurrency” – a formally correct but nonetheless misleading statement. Yes, you no longer have fear of data races, but you have PAIN, much pain.

What's going on here? Why is the language causing so much pain voted as "the most loved language" for like 7 years in a row in the Stack Overflow survey. Is it the Stockholm syndrome? Do Rust devs like pain? Or maybe it's a huge cult: you get in and you try to lure in as many people as you can so they feel the pain too.

The truth is: this is probably not the experience most people have with Rust. It's definitely not the experience I have with Rust. In fact, it's vastly different. Which doesn't mean Hirrolot is wrong, it just means they have a different perspective and different experience, probably because of the stuff they needed to implement in Rust.

Is Rust totally painless? Of course not, I had my fair share of fights with the compiler to try to make code compile. I definitely get frustrated from time to time, but as a professional developer I can stomach *a lot* of compile time frustration. I have much less tolerance for runtime frustration. Or in other words: I prefer to spend a few hours being frustrated at a compiler rather than a few hours debugging a production issue under pressure. This is a tradeoff, for sure, so your milage may vary, but that's just where I'm standing after 17 years or so in the industry.

async code in applications

While I definitely agree async adds complexity to Rust and if you don't need it, it will be most probably easier to not use it, I don't think it's practical to outright recommend not using async at all. But the question is: what do you use async for? Most of the time it would be probably something like a web service. And then most of the time your code will look something like:

```
#[get("/todos/{id:\\d+}")]
async fn todos_show_handler(
    id: web::Path<i64>,
    pool: web::Data<PgPool>,
    routing: web::Data<RoutingService>,
) -> Result<TodoPresenter, Error> {
    let todo = sqlx::query_as!(Todo, r#"SELECT * FROM todos WHERE id = $1"#, *id)
        .fetch_one(pool.get_ref())
        .await?;

    let url = routing.todo_url(*id);
    Ok(TodoPresenter { todo, url })
}
```

This is one of the handlers from my implementation of a Todo Backend. It's what you would do in vast majority of your handlers: get an input from a request, fetch data from the database or otherwise do some kind of computation and return the result. Notice that there are no explicit lifetimes here and you need only a very limited undersanding of the borrow checker. There is shared state here (in the form of the database pool and the router "service"), but it's shared using the `web::Data` type, which is a type from actix-web very similar to `Arc`. In fact, that's also how I think a lot of async code would be written, also outside of web frameworks: instead of dealing with lifetimes just use `Arc` or `Arc<Mutex<...>>`. That's also why my first article on this blog is about `Arc` and `Mutex` - I think `Arc` should be the first thing you try when you need to share stuff between threads, not the last. Most of the time it will be fast enough, don't worry. Then only if you need more performance or you think you can pull off a better API you can try something more advanced, but otherwise? Why would you make your life harder?

If you always use `Arc`, why not just use GCed language

As my response to most of the ownership and lifetimes problems is to just slap an `Arc` or an `Rc` there, I've heard quite a lot of variations of the question: If you default to using `Arc`, won't it be slower than using a language with a sophisticated garbage collector? It's a good question and if you really put every single thing behind an `Arc`, it would probably be the case. Which would indeed defeat some of the reasons to use Rust. Not all, mind you, you would still get data races free programs and predictable memory usage, but yes, it wouldn't be optimal.

The good thing is, by saying "just use `Arc`" I'm not advocating to hide every single variable behind an `Arc`. You mostly need it for shared state. Remember the example from the previous section? It had two things behind an `Arc`, but the rest was pretty much regular Rust code. So while, yes, every request to your application would have to do at least two `Arc.clone()` operations in this specific case, the performance penalty is negligable. And unless you're writing a "close to the metal" layer 4 load balancer or something, you probably won't care. It's fine, really.

The dynamic dispatcher

Hirrolot tries to implement a dynamic dispatcher in Rust. Not only that - it takes a closure as an argument. This is one of the things that is just hard to do in Rust. There are a few things like that, you might have heard about linked lists already. It's just one of the tradeoffs. So, yes, if you are planning to create an API that holds async closures and does zero extra allocations on top of it, you're in for a bad, bad time. Fortunately, we can compromise, it's not like your manager will come to you and tell you "we need this feature for yesterday. and remember, no allocations and no `Arc`!". Or actually they might, but in this case you're probably working in embedded or on something very specific that would be challenging to write even without Rust.

If I had to implement something like that, I would do it this way:

```rust
use async_trait::async_trait;

#[derive(Debug)]
struct Update;

#[async_trait]
trait Handler {
    async fn update(&self, update: &Update) -> ();
}

#[derive(Default)]
struct Dispatcher {
    handlers: Vec<Box<dyn Handler>>,
}

impl Dispatcher {
    pub fn push_handler(&mut self, handler: Box<dyn Handler>) {
        self.handlers.push(handler);
    }
}

// example handler
struct Foo {}
#[async_trait]
impl Handler for Foo {
    async fn update(&self, update: &Update) -> () {
        println!("Update: {:?}", update);
    }
}

#[tokio::main]
async fn main() {
    let mut dispatcher = Dispatcher::default();
    let handler = Box::new(Foo {});

    dispatcher.push_handler(handler);
}
```

Instead of using closures I would make a `Handler` trait. No `Arc`, no lifetimes, no worries. Of course there are potential issues here:

1. I'm using `async_trait` here (which Hirrolot doesn't like, but I don't mind), so a handler needs to be `Send`. If this is an issue you could also implement a `!Send` version by using `#[async_trait(?Send)]`
2. It needs more boilerplate than the closure version or even the `fn` version
3. We need to wrap handlers in a `Box`
4. The interface uses `&self`, so if you need `&mut self` you would have to change it

And probably more. But does it matter? If you're writing application code (which most of the Rust coders do), I don't think so. All of those things are minor annoyances and after you write the actual handlers you can adjust the API as needed. These problems would only matter in any meaningful way if you were writing a library that has to work for many different scenarios for as many users as possible. So we get back to what I wrote in the beginning - problems of library authors are different than problems of application developers.

Also this is a theoretical problem. If it was a practical problem to be solved the API I come up with might be vastly different, but with the amount of information I have I don't think it matters much.

Library code vs application code

(this section was added after posting the initial version)

One of the interesting comments I got in response to this post was asking about a distinction between library code and application code. Usually when writing applications you tend to extract duplicated code and the generalized version is often something you could release as a library. Does it mean that it doesn't happen in Rust. It does, but I think the constraints for internal libraries are usually less strict than for external libraries. It's also easier to compromise - you roughly know what you will be using the code for, so you can write it in a way that works for your use case. And if you don't get it right the first time? It's not a big deal if all of the users of your library are in the same company.

I know some of you will frown on this. If you don't get it right it means updates and updates are costly! Yes and no - I find it that in Rust refactoring and updating code is much easier than in other languages I know - the type system will guide you through it.

No problem then?

After reading the article this far you may be thinking I'm just an ignorant who doesn't see any issues with Rust. A fanboy that would write anything to defend his favourite language. That's definitely not the case. Well, at least the part about writing anything to defend the language. I *am* a fanboy and Rust *is* my favourite language.

I am aware of the issues with Rust. I know it sucks for library authors like Hirrolot and I wish things were progressing faster. If we had GATs and Polonius and many other awaited features, stuff like this would be hopefully easier. But it takes a long time to implement these kind of features and I doubt we will get any of that very soon.

The issue is, I think, that without proper context, Hirrolot's article is discouraging people from learning Rust, especially with the following sentiment in there:

> Still think that programming with borrow checker is easy and everybody can do it after some practice?

This is doing a lot of harm, because now beginners reading this might think: oh no, if this is how Rust works, I better quit now, I will never understand the language at this level. And it's sad, because most people won't have to. If you're a beginner in Rust you don't need to understand everything. You don't need to write generic flexible APIs that will work for everyone and every workload. You don't need to write zero allocations code.

Can't I be a library author?

(this section was added after posting the initial version)

While this article tries to show that Rust is managable for app developers, it might paint a grim picture for library developers. It's indeed harder to write Rust libraries than it is to write libraries in other languages. Getting the API write usually means you must think about types, consider allocations, sometimes deal with advanced lifetimes or traits. But it is doable too, you may need to make compromises, though.

As you saw in both Hirrolot's article and mine, you can design a relatively simple API for a dispatcher handling updates. It may not be an ideal API. It may have some constraints that it wouldn't have in other languages, but it is doable. Again, it's a tradeoff.

Summary

If there is one thing I want you to get out of this article it's this: Rust doesn't have to be very hard. It is a hard language, especially comparing to languages like Javascript or Go, but it's not unachivable. And it will only get easier over time. Writing Rust in 2022 is a vastly different experience than writing Rust when it hit 1.0 version and the Rust team is working hard on making it more approachable.

So if you're wondering if it's a language for you I hope you give it a try!

If you like this post please consider following me on Twitter!