Compiler-Driven Development: Making Rust Work for You

Intro

Hello everyone. Today, I will be talking about what I refer to as "compiler driven development". I want to make Rust work for you. Rust has many features that make your life as a developer easier. And in combination, Rust's developer experience is one of the best and is only improving as time goes on.

■ Diagnostics

So let's start off with diagnostics. If you're just getting started with Rust, I have good news for you. Rust's diagnostics are world class. They are a powerful feature designed to guide developers towards writing correct and idiomatic code. The compiler provides detailed error messages, warnings, and suggestions that help identify and address potential issues before code is ever run. These diagnostics often include specific explanations, examples, and links to relevant documentatoin, which makes it easier to understand and resolve problems. Built-in tools like `cargo fix` work with these diagnostics by automating the process and applying suggested fixes, such as updating deprecated methods or correcting some common mistakes.

■ So first, what's the difference between a warning and an error? Well, ■ a warning is not fatal; an error is fatal, meaning that an error will actually stop compilation no matter what. ■ But that's because an error indicates a near certain issue, whereas a warning just a potential issue. ■ And as such, a warning *should* have action taken even if that's just silencing it, whereas for an error, it *must* be or your code will no longer compile. ■ So, for diagnostics, read them! They're very important and they're really good.

■ So if we have some sample code that doesn't compile, it should be pretty easy to see why. ■ But the compiler will tell you, clearly. It shows on line 5 that there is

a variant on line 3 and 4, it has a duplicate attribute. So on the bottom, it'll show "try removing one of those two attributes" and it will compile. And if you run this with `cargo fix`, it'll remove it for you.

So bad diagnostics do exist in Rust. It's not common and it is considered a bug. If you encounter bad diagnostics, open an issue. It will be taken seriously.

■ Lints

So moving on to lints. ■ So why should we actually use lints? ■ They provide significant value, catching countless bugs that may be difficult to find otherwise. That's generally on the order of the 70% of bugs that are memory safety. ■ But it also improves performance. You might be doing something that is more expensive than it needs to be, and as such, could be improved. ■ It also helps you write idiomatic code, writing Rust the way Rust should be written rather than writing as if it's another language. And it will also help performance because Rust is designed to compile Rust, of course.

■ So there are different types of lints. There are ones that are built into the compiler itself and ones that are in clippy. ■ Now as to which should be enabled, there's almost a thousand lints total and we'll probably cross that in the next few weeks. So it is difficult to know. Some lints are deprecated, some conflict with others, so obviously you shouldn't enable them all. However, there are a number that should be enabled and are enabled by default. You should not really lessen the level of lints because the defaults are there for a reason, naturally — that's why it's the default. ■ So you will want to enable a number of lints and my personal recommendation is just to go through them, honestly. Yes, it's a lot, but it's worth it. It'll take probably an hour; it's not that big of a deal. ■ So which lints would you want to enable as warn and which would you want to deny, meaning create an error and actually stop compilation? Some lints you will want to do in order to prevent memory safety issues (if you're using `unsafe`). Some are just stylistic, and you will have to make that judgement call for yourself. But personally, I find that lints that indicate a likely error you should set to eny even if they are quite strict. And again, the defaults are a great starting plcae and you should not decrease the severity without a strong reason. That's that way for a reason.

■ Rust Analyzer

So on to rust-analyzer. Aside from TypeScript's, Rust has one of the best language servers available. rust-analyzer is very powerful and has some nice features that can help you write better code and to do so more efficiently.

First of which, of course, ■ is check on save. Every time you save a file, rust-analyzer runs `cargo check` to ensure that the code compiles. This can be improved ■ by using clippy, which is Rust's built-in linter. And that will catch even more issues and enforce the best practices from the start like I had mentioned in the last slide. And it is important that using that from the start avoids the need to refactor code later on, which saves both time and effort. And you can enable clippy with rust-analyzer in the settings. If you look that up online, it's very easy to find.

One of the significant features of rust-analyzer is ■ inlay hints, and that's highlighted here with the yellow background and red text. So rust-analyzer shows various hints in your editor, including inferred types, which is what's shown here. Lifetime annotations, implicit dereferences, many other things that can be enabled as you choose. That can help you understand code that you aren't familiar with or simply haven't touched in a while. It also serves as a powerful learning tool for beginners by explicitly showing various parts of Rust that are often implicit. Having a type annotation may not seem significant, but if you're not familiar with the language, it will help you know what the actual type is. Even for experienced developers, these hints are useful in determining whether a piece of code is sound, showing types at various points in an iterator chain, showing generics when you're not quite sure what the type is, or just showing a lifetime.

■ rust-analyzer also has some code completion. Wouldn't we all like to type less? rust-analyzer knows the types of variables from the signatures of functions. It can show various possible completions, like here, showing the different fields. It can also automatically import an item if it's not yet in scope most of the time. This can save you the hassle of looking up the correct path. And let's face it, we all have to do that even for common items. As a bonus, you don't have to worry about typos when it does that.

■ There's also some refactoring tools, ranging from simple renaming shown here,

to more advanced options such as extracting a block of code into a standalone function. And when you do that, it will automatically determine the necessary parameters for the function as well as the return type. And aside from simple renames, in my experiences, these aren't used too often, but when they are, they save a lot of time and a lot of hassle.

■ There's also code navigation if you just want to see where code is used or where it is defined. If you see above line one where it says "one reference". If you click on that, you get this nice little pop-up. And it'll show you on line eight, which is highlighted in that popup, that is where it is used. And conversely, if you click on the `x` in VS Code, at least with control held it will bring you back to the definition. And that works across crate boundaries and in larger code bases, especially ones where that are many similarly-named structs. It's a huge time saver because you know where it brought you is actually correct and you're not looking at some completely unrelated struct with a similar name.

■ There is also code generation. rust-analyzer is capable of generating some relatively basic code, such as implementing a stubbed out trait, writing the variants of an enum in a match statement. This saves some time when writing boilerplate code. It can also help you learn what is needed to get your code to compile. And as shown in the image on the right, you can also generate getters and setters, delegate methods to a certain field, or even implement a and derive. So that's just getting started with Rust.

■ Type system

What about the type system? There's a famous saying with Rust: ■ "if it compiles, it works". Now that's not strictly true; of course there are still logical errors. But I guarantee you, if you only ever write safe Rust code, you will not get a segfault. ■ Fighting the borrow checker is a common complaint, but it's actually a feature. As I had said before, ■ entire classes of bugs are eliminated. And per both Microsoft and Google, that is 70% of CVEs. A number of bugs that are difficult to confirm the lack of in both C and C++:

- ■ Null pointer exceptions, they don't happen. There's just no null.

- ■ Use after free, ■ dangling pointers, ■ and double frees. None of them can exist because lifetimes are enforced.
- There's also ■ data races between different threads. And those can't happen because Rust enforces exclusive mutability. Only one thing can hold a mutable reference to a struct at a given time.
- Also ■ uninitialized reads, which is quite common in C, as it turns out. And that's just enforced by the type system in general. You can't use a variable if it's not defined.

■ So as I had said, entire classes of bugs are eliminated. And as a result, ■ you just don't have to check things. The compiler does it for you. As an example, you could have exhaustive pattern matching. If you add a new variant to an enum, then the compiler will error out anywhere else you're using it because you haven't yet matched on that variant. You can also leverage ownership to take advantage of invariance. While in Python, you will frequently see "with a file open" and then have a scope, in Rust, merely having the file itself or a reference to it is sufficient to prove that the file is open and that you can use it. And I would like to emphasize ■ large refactorings are acceptable with Rust. I have more than once found myself doing more than a couple thousand lines of code difference in a single pull request, and that's acceptable. It's not necessarily *recommended* because it is still difficult to review, but it is *possible*. And the reason for that is when you have so much code that the compiler is just able to know how things interact. And when it does htat, it is able to enforce numerous invariants, figure out what needs to happen in order for things to compile and will tell you what needs to change for it to compile. And to me, that's pretty impressive. I don't know of any other language that does that.

■ Design patterns

There's also design patterns. So you will occasionally see these. Some of them are quite common, some of them not so much.

First, ■ the newtype pattern which I would say is more frequent. And what this is, is a simple wrapper aorund an existing type, whether it be an integer, a struct, whatever. And this is used ■ to either enforce, restrict, or both the functionality of

the inner type. And this is necessary because of Rust's orphan rule. You cannot implement a trait or a function or anything of that sort for a type that you don't own. And of course, you can't *remove* a function for something that you do not own. ■ And this can be used to enforce different types for different use cases as well. So if you have a user in a datbase, it's stored as an integer, almost always. ■ If you wrap that in a struct and then store that construct as a field, now if somebody just tried to assign that ID to the field, they can't. The compiler will stop them because it is expecting a user ID, not an integer. You would have to manually construct the user ID, which hopefully if you're using the interfaces correctly, is not possible.

■ ■ You can also avoid leaking sensitive information. This sounds odd, but if you have a password, you can just wrap it in a new struct and implement `Debug` for it where the password isn't exposed, whereas the default implementation for a string just prints it out. That's not too useful when you're logging things and you have a security issue, which is going to happen.

■ ■ There's also the builder pattern. I won't be going into detail for how this is implemented too much, but sometimes you will see the type, which is `Foo`, and then a number of methods chained with some arguments, and then `build`. Sometimes `build` won't even be present. That's all enforced by the compiler. It knows which fields have been initialized, which ones haven't, and if those ones that haven't are actually needed or if they have default values. And if you do that, it all compiles down to nothing. It just initializes the fields directly and that's it.

■ ■ One more design pattern is typestate. Say we have a drone; you want it to take off and land. We're not flying yet — we're not that complicated. So if you have a drone that's on the ground and you told it to land, what would you expect it to do? Well the question just doesn't make sense because it's already landed. So instead, just make it so that it cannot land. And you can do this by providing a generic argument to `Drone` and then only providing the `take_off` method for that. And notably, we are taking `self` by value and not by reference. That way the previous state of `Drone` does not exist after it is called. And note that we also return a "hovering" `Drone` so that you can use it afterwards. And vice versa, you have a hovering `Drone`, you can then land it, and you have returned a landed

`Drone`. And on the bottom line, if you initialize a landed drone and try to call land, like I had mentioned, it just won't compile. At all. Now this is a simple example, of course, but this design pattern can be leveraged for a number of other things. You could extract this to dozens of fields, which I have personally done. It's complicated, but it is doable and it does work. It will prevent you from doing something you shouldn't.

■ Pitfalls

Now, there are some pitfalls with Rust, as with any language. ■ First and foremost, `unwrap` and `expect`. They cause panics. You might not want that; you probably don't, in fact. Now, these methods should not be avoided entirely. They are in the standard library for a reason. They should be used with *caution* in favor of proper error handling, such as returning the error from the method.

■ There's also lifetimes, which are famously viral. Once a lifetime is introduced, it is difficult to impossible to remove it. If you have it stored as a field, you pretty much have to have it as a static lifetime. Many times it is simpler to clone the data, which would let you have it be owned and will avoid the lifetime altogether. That is particularly useful when prototyping. If you're not sure if the overhead is worth it, just do it. Premature optimization is the root of all evil, of course. Wait until you know that there is a performance concern before optimizing away the clones. A number of times, the compiler is smart enough to optimize it away for you, so you don't even have to worry about it.

■ There's also unsafe code. *Some*, but not even most, of what I have said can be bypassed by unsafe code. It is not easy to do for some things; others, it's quite easy. You need to be careful when you write unsafe. You have to document *why* you are writing unsafe code, and `clippy` can enforce that for you. There is a lint that will make you put a safety comment on every single unsafe block in your entire code base, and it could fail compilation if you set it to `deny`. And also, `unsafe fn` currently creates an implicit `unsafe` block for the entire body. This is going to change in the 2024 edition, but in the meantime, all code that has an `unsafe fn` — the entire inside, you could be using something unsafe and not even realize it because the compiler doesn't tell you.

■ And of course, usability. Rust is a complex language, after all. It's difficult to learn. I don't think anybody could reasonably disagree with that. There's nothing wrong with using more basic features of the language until you're more comfortable with it. Don't feel pressured to use Rust's features right away. It's better to write *working* code that you understand than *questionably* code that you don't. That's especially important when working with others. You don't want someone to come up to you, ask you how your code works, and you can't explain it. If you're writing a library, especially consider the usability of your APIs. Some APIs might theoretically work and solve the problem, but are frustrating to the end user to use it correctly. Striking the right balance is very difficult to achieve. It is worth striving for. In short, keep it simple. Do the least you need to do to solve the problem.

■ Example: provenance

Now, for probably the most complicated thing *I* have done, leveraging the compiler, is provenance. Provenance, to put it simply, is the history of how something is owned, at least in Rust. For my personal experience, I was trying to write a parser. And with that parser, I was trying to enforce that the remaining input was, in fact, the remaining input. That it originated from the original input. That's harder than it sounds. And I didn't even realize this, that I even had a bug, until I was writing tests after I had written a few thousand lines of code. I quickly wrote a test and realized I was right. It didn't work. You could easily read from the end and return the beginning. I didn't want to allow that. So I came up with an idea, and I had to dig through quite a bit of papers, including a master's thesis, which was the end inspiration ■ for this.

By inroducing provenance, that's the `'prov` lifetime there next to `parse_bytes`. By merely introducing that one lifetime and changing the input type from a byte slice to a byte object with the input and provenance lifetimes, I could enforce it. And that is the extent of the public API that a library author has to know about. A downstream user who's just using the parser doesn't know about it at all.

■ So bytes, very simple. It's a thin wrapper around a slice of bytes. The variance, this is the more complicated subject that you can find in the Rust book. This is a

cononical way to write an invariant lifetime, basically meaning the lifetime can neither be extended nor shortened in any situation. So knowing that bytes is *functionally* the same as a byte slice, we'll ■ get rid of that.

■ Instead, we're implementing `Bytes`. The key here is that the only way to construct it is unsafe. And in doing so, the programmer has to enforce that the provenance is what it says. But if you're doing that, all is good! You can have `split_at`, which takes it as a referece and returns the remaining input at the beginning as an ordinary byte slice, and then returns the remaining input as another `Bytes` object. Which, okay, that doesn't actually enforce anything. Anybody can just create it.

In short, ■ this is the only method that actually creates a new byte slice from scratch. On the third line of this exapmle for the parameter `f`, you have `impl` for the provenance. If you're not familiar with that syntax, I don't blame you. It's quite niche. It's not used very often. All you really need to know is that the user can't provide that lifetime; the compiler has to. And as a result, if you call it in a very specific way, in this situation via a closure, then the compiler will generate a new lifetime for every invocation of that method. So if I call the method, you're all set. That's enough to prove that because it was called, following a series of steps through the trait system, closures, lifetimes, provenance — all of that combined results in being able to enforce that the output originates from the input *and* that it is a trailing subset or the entire string that was put in. So, that's pretty complicated, I'm not going to lie.

■ Outro

But what's what I have for today. Hopefully you learned something whether you're just starting out in your journey with Rust or work on the language itself. I know when I initially asked about the provenance, there was a lang team member that said it would be wonderful to know how it worked. That was the first comment I got. It was not very promising to hear that.

Hopefully this talk makes Rust work for you and that you're able to use the compiler to your advantage. On the screen is my contact information and more. I'll

be around on Discord to answer any questions. Thank you.