November 12, 2024

Rust needs an official specification

Can we currently reason about Rust code with absolute certainty? Not really, but we should be able to. In this article, we dive into the reasons why it may be time for a Rust specification.

A simple C++ question

If you know a little bit of C++: can you tell me what the output of the following program is?

```
#include <stdio.h>

struct Foo {
    ~Foo() { puts("Goodbye!"); } // runs when Foo is destructed
};

int main() {
    Foo _tmp;                    // a Foo object is constructed
    puts("Hello");
}
```

This is not a trick question! The answer is of course *"Hello"* followed by *"Goodbye"*. But is this always the case, no matter which C++ compiler is used, or which optimization flags?

The answer is a resounding **yes**! Someone versed in C++ can explain that the `puts` function in the destructor `~Foo` will be run at the closing `}` of the `main` function. No "if", "but", or "well, it depends."

Why is this so? It's not because we have tested this program extensively on every C++ compiler, but because *the language rules of C++ say so*.

And we can look at them! Let's take out our ~~bible~~C++ Standard and open it at chapter 6, section 7.5.3 (*Automatic storage duration*), third verse, and behold:

If a variable with automatic storage duration has initialization or a destructor with side effects, an implementation shall not destroy it before the end of its block nor eliminate it as an optimization, even if it appears to be unused, except that a class object or its copy/move may be eliminated as specified in 11.10.5

Okay, maybe that was not so easy to read. That's because it is written in 'standardese': a precise style of language riddled with technical jargon used in documents that define formal standards. But like a legal contract written in 'legalese', once you become comfortable with it, you can start to unpack the concepts within it.

For example, the above mentions a variable with "automatic storage duration" (`_tmp` has this), and "a destructor with side effects" (performing I/O using `puts()` definitely is a very noticeable side effect). There is an explicit reference to another article with an exception, but it doesn't apply here---ergo, `_tmp` shall not be destroyed before the end of its block.

You may argue that I made some claims in the last paragraph, such as `_tmp` having "automatic storage duration" or that the exception does not apply---if so challenged, I'm sure I can find articles in the C++ standard to back those claims. And we can probably continue this until we reach claims that are self-evident (or at least beyond a reasonable doubt). But let's stop here and not do a full exegesis. For a C++ expert, the above claims should be clear and independently verifiable.

Of course there also exist situations where it is not entirely clear what a C++ program will do. Often that is deliberate (compilers need to have wiggle room to allow for optimizations). Every so often, it is caused by an omission in the Standard, and then these can be reported to the authorities who maintain it: the C++ Committee. If they agree it is an omission, it will get labelled as a Defect Report, and they will then discuss and debate it, and ultimately come up with a clarification of the text.

A simple Rust question

If you know a little bit of Rust: can you tell me what the output of the following program is?

```
struct Foo;

impl Drop for Foo {
    fn drop(&mut self) {
        println!("Goodbye"); // runs when Foo is dropped
    }
}
```

```
fn main() {
    let _tmp = Foo;      // a Foo object is constructed
    println!("Hello");
}
```

This looks the same as the C++ case, right? And if I run it, it sure *looks like* it is the same: *"Hello"*, then *"Goodbye"*. And the documentation for the `Drop` trait seems to suggest that's the intended behaviour. **But**, surprise... if I simplify the name of the variable in line 10:

```
    let _ = Foo;
```

the output *will change* into *"Goodbye"* followed by *"Hello"*! Hang on, what's going on here?

I tried consulting the Rust Reference on destructors but didn't see a clear answer. I asked someone with over 8 years of Rust experience, who was equally surprised but gave me an "it depends" rationalization (paraphrased):

The compiler is allowed to drop objects when it can determine they are no longer needed, even if that's before they go out of scope.

But that didn't feel right to me. I stumbled on another section of the Rust Reference which contained a hint: changing `_foo` into `_` is not simply a change in the *name* of a variable, but something else entirely. But I didn't read an answer how that affects the moment the `Foo` object gets dropped.

So what is going on here:

1. The compiler does something surprising.
2. Developers disagree what the exact meaning of the program code is.
3. The Rust Reference cannot be used to adjudicate the case: we have to connect dots and draw from personal insight. Or in other words, guess.

Okay, I hear you sigh: *who cares* about this trivial toy example. But if there's already room for doubt here, what if you start writing serious Rust code which involves the borrow checker, multiple lifetimes, and `unsafe` code? In some cases you will find clear documentation in the Rust documentation, in the Rustonomicon, or the Rust unsafe coding guidelines. But not always. And then what exactly is the *authority* of any of those documents?

How a Rust Language Specification would help us

I want to stress that I am **not** attacking the supposed "difficulty" of Rust with this example. That's just hogwash. I might actually be in a minority here, but I think that Rust is a cleaner, more easily understood and friendlier language than most languages---and certainly **much** easier to understand than modern C++.

Every language has things that trip up programmers, regardless of proficiency. And in many cases, you can simply *test* your software to a sufficient degree to gain confidence that it does what you think it does.

But often you want a stronger guarantee, which is something an *official language specification* can give us:

- When you *really* want to understand why the compiler does what it does, to be more productive in the future.
- When you don't control (or even know about) the target platform (or compiler) which will be used to run your code.
- When you want to know if your code will behave the same if you upgrade to the newer version of the compiler.
- For developing external code analysis tools such as Coverity. Sure, we have clippy (a really useful linting tool) and rather specific analysis tools such as Miri and KANI. But all of these are heavily dependent on `rustc` in some way.
- To allow for competing, alternative compilers such as gccrs, avoiding vendor lock-in, without the risk of fragmenting the language into dialects.
- When you are writing `unsafe` Rust: if you have to walk a tightrope without a safety net, you had better know where to attach the rope.
- To work with components written in different languages. In this case, you need to be able to reason about the FFI.
- When you are writing safety-critical software and you have to comply with certain safety standards. Standards which, in turn, demand the conformance of your tools with certain other standards. Which presupposes the existence of said standards.

The Ferrocene specification

For Rust, the closest thing we have to a specification right now is the Ferrocene Language Specification. It technically only applies to the Ferrocene compiler, but it **can** explain the program that was shown earlier. Let's dive into some standardese.

The form `let _tmp = ...` uses an identifier pattern (5.1:1), but `let _ = ...` uses an underscore pattern (5.2:4). The former introduces a binding (5.3:1), but the latter doesn't.

This in turn means they have different drop scopes: the binding introduced by `let _tmp = ...` is dropped at the end of the block in which it occurs (15.8:19), but the temporary holding the value created in `let _ = ...` is simply dropped when execution moves beyond the let statement itself (15.8:11).

That explains it clearly enough for me: the moment a value is dropped can be predicted when using the Ferrocene compiler.

And I can tell you right away, the same holds for "official" Rust that it is currently based on---the source code for Ferrocene is available. We have looked at it.

(Note that the references above refer to Ferrocene Language Specification that was based on Rust v1.81.)

Types of specification

When it comes to specifications, it is important to ask the question: "who and by whose authority?"

Let's continue with the Ferrocene Specification: here, a commercial company (our friends at Ferrous Systems) makes promises about a Rust compiler that *they* ship. It is not *officially* promulgated by the Rust Project.

It is also *descriptive* rather than *prescriptive*: whatever they write has to follow what the Rust compiler does. They don't make the rules, they only document and cross-reference them very precisely. If they wanted to change anything, they would need to go through the Rust RFC process like anybody else.

On the other end of the spectrum sit C and C++. These are *officially standardized*, which means they have a language specification that is codified in an *international standard*, maintained by the International Standards Organisation. They are *prescriptive*: if Rust were standardized like this, the Rust Project's Language Team and Library Team would essentially be subsumed by an ISO Working Group. And as Mara Bos explains, that does mean giving up some control.

There is also a middle ground, as illustrated by Java. Here there is one dominant compiler vendor, and they also provide the language specification. This is *descriptive* for them, but *prescriptive* for anybody that wants to make an alternative Java implementation. In a sense there is a *de facto* standard: Java is whatever Oracle says it is.

Life cycle of a programming language

Programming languages evolve, and so do specifications.

C evolved in stages at Bell Labs during the early 1970s under the control of a small team of pioneers (including Ken Thompson and Dennis Ritchie), who also gave the world UNIX.

Their lab experiment escaped, and thrived in the outside world. Near the end of the 1970s, it was codified in a book by Brian Kernighan and Dennis Ritchie. That language we now call "K&R C". Eleven years later, the language was amended and standardized by ANSI---becoming an official American standard---giving us "ANSI C". This in turn was adopted with minimal changes by ISO in 1990---becoming an *official international standard* that has since seen updates in 1999, 2011 and 2018.

As C matured and became standardized, control passed from the original creators to a committee with a regulated, open process. For example, when ANSI/ISO C was being standardized, the committee was already considering adding features that Dennis Ritchie opposed. About the 1999 version, Ritchie is quoted as saying:

> *I was satisfied with the 1989/1990 ANSI/ISO standard. [...] I'm less ecstatic about the C99 standard, but don't denounce it.*

(Which is a nice example of damning with faint praise.)

Innovation in C since 1989 has happened much more slowly than is the case in Rust right now. In a sense, that is also the purpose of having a standard: creating a stable and reliable platform for building software where new features *don't* appear every three fortnights. It also gives gravitas: "K&R" C looks completely archaic and obsolete today, but "ANSI/ISO" C is still relevant 30+ years later.

Of course, C has also been for a long time "what it needed to be". It is a more or less finished language, but not a dead one. But if we briefly look at C++, things are not that different: standardization in 1998 slowed down its progress as compiler writers were struggling to implement all of it. And even though in C++11 (and beyond), so many changes have been made that C++ has almost morphed into a different language, the pace of innovation is **much** slower, and also much less *gradual* than currently allowed by the 6-week release cycle of the Rust compiler. If Rust were to work this way, Rust Editions would be the way new features get added to the language.

When you really *need* a specification

In my view, having specifications/standards is great for systems programming, or when you want to write portable software. Good examples are POSIX for interacting with a UNIX-like operating system, or IEEE-754 which describes how floating point arithmetic should behave. We need basic guarantees everybody can rely on.

But sometimes it is more than simply a "really great to have". For safety-critical infrastructure, industry has agreed upon certain norms for risk management such as the Safety Integrity Levels (SIL) standardized in IEC 61508, or ASIL standardized in ISO 26262. For example, level SIL-4 means that the chance of an "on-demand failure" is less than 1:10000. It's pretty hard to quantify such risks for software to begin with, but it becomes impossible without solid foundations.

For software, there are also the Evaluation Assurance Levels (EAL). These describe how rigorously software should be tested and documented, and consist of various components combined into packages: from EAL1 to EAL7.

To achieve EAL4, for example, it is demanded that *"well-defined developer tools"* are used, and for EAL4, these have to comply with *"implementation standards"*. *Well-defined* is defined as:

These are tools that are clearly and completely described. For example, programming languages and computer aided design (CAD) systems that are based on a standard published by standards bodies are considered to be well-defined. Self-made tools would need further investigation to clarify whether they are well-defined.

So for EAL4, a specification is useful, but perhaps not absolutely necessary. On the other hand, EAL5 explicitly states that:

[This requirement is] to ensure that all statements in the source code have an unambiguous meaning. [...] implementation guidelines may be accepted as an implementation standard if they have been approved by some group of experts (e.g. academic experts, standards bodies). Implementation standards are normally public, well accepted and common practise in a specific industry, but developer-specific implementation guidelines may also be accepted as a standard; the emphasis is on the expertise.

In other words, without a clear and authoritative specification, Rust cannot be used to achieve EAL5.

But don't we, as Rust advocates, want more of Rust's type and memory safety, **especially** in safety-critical applications?

Bring on the Rust Language Specification!

The most authoritative documentation for Rust is the Rust Reference, but it is not a true specification (as I tried to gently illustrate at the beginning of this blog). What's more, in some areas, such as what the precise aliasing rules in `unsafe` Rust are, what rules work best is still being figured out.

Two years ago, Mara Bos wrote that she thought Rust needs a (descriptive) *specification*, but doesn't need a (prescriptive) *standard*. She posits that the latter isn't necessary since:

1. Rust already has an open process with RFCs, so having an ISO Working Group doesn't add much transparency.
2. Stability and reliability of Rust is currently adequately guaranteed by using "Rust Editions", putting experimental features under feature flags, and because of the existence of Crater.

I almost entirely agree. Rust has a vibrant community and is a rapidly-but-gradually evolving language, that has stability without stagnation as a principle. As I've tried to show by analogy with C/C++, "going full ISO" would slow things down; leading to a large set of updates being rolled out every 5 years or so; and wresting control away from a process that is currently doing a fine job.

There's no need to change that... yet. But in ten years, I might feel differently. Perhaps a way can be found to preserve "stability without stagnation" within the ISO committee process. Who knows? Ten years is a long time. (As we say in Dutch: "who lives then, cares then").

In the shorter term, having a Rust Language specification that is "official" (i.e. has the imprimatur of the Rust Project) is absolutely necessary to use Rust for what it is *ideally positioned to do*: safety-critical software. The Ferrocene Language Specification does not have this status.

And we could use a bit more stabilization and standardization in other areas as well. Rust's interoperability with other languages such as C++ could be easier. The Rust Standard Library (the word 'standard' is interesting here!) is also small, which leads to Rust programs often needing many dependencies. Maybe we need to build a consensus about what crates are 'almost standard'? Or maybe Rust even needs an extended standard library.

This is not merely a technical challenge, it's also a political one. A language specification functions as a contract between supplier and users, it's not a theoretical document. Some hard decisions will need to be made and everybody will be unhappy about *something* whenever it gets set in stone. Perfect is the enemy of good. The Rust Language Specification will be no different.

One interesting example will be the future of Ferrocene. Since it was made with automotive and industrial applications in mind, its specification might be lacking in some areas. On the other hand, it is already there, it is open source, and being used today! At the very least it demonstrates that a usable Rust Specification *can* be made, and that it shouldn't need to take many years to create a first version.

In any case, this ball is rolling! RFC3355 was accepted one year ago, and a specification team is at work. We are watching!

Marc
Software engineer
marc@tweedegolf.com

rust