How does Rust store types at runtime?

Asked 5 years, 2 months ago Modified 5 years, 2 months ago

Viewed 2k times

5

A `u32` takes 4 bytes of memory, a `String` takes 3 pointer-sized integers (for location, size, and reserved space) on the stack, plus some amount on the heap.

This to me implies that Rust doesn't know, when the code is executed, what type is stored at a particular location, because that knowledge would require more memory.

But at the same time, does it not need to know what type is stored at 0xfa3d2f10, in order to be able to interpret the bytes at that location? For example, to know that the next bytes form the spec of a `String` on the heap?

memory **types** **rust**

Share

Improve this question

Follow

edited Sep 25, 2019 at 18:35

Shepmaster

428k ● 111 ● 1.2k ● 1.5k

asked Sep 25, 2019 at 18:23

joel

7,798 ● 4 ● 40 ● 70

1 Answer

Sorted by: Highest score (default)

9

> How does Rust store types at runtime?

It doesn't, generally.

> Rust doesn't know, when the code is executed, what type is stored at a particular location

Correct.

> does it not need to know what type is stored

No, the bytes in memory should be correct, and the rest of the code assumes as much. The offsets of fields in a struct are baked-in to the generated machine code.

When does Rust store something *like* type information?

When performing dynamic dispatch, a *fat pointer* is used. This is composed of a pointer to the data and a pointer to a *vtable*, a collection of functions that make up the interface in question. The vtable could be considered a

representation of the type, but it doesn't have a lot of the information that you might think goes into "a type" (unless the trait requires it). Dynamic dispatch isn't super common in Rust as most people prefer static dispatch when it's possible, but both techniques have their benefits.

There's also concepts like `TypeId`, which can represent one specific type, but only of a subset of types. It also doesn't provide much capability besides "are these the same type or not".

Isn't this all terribly brittle?

Yes, it *can* be, which is one of the things that makes Rust so interesting.

In a language like C or C++, there's not much that safeguards the programmer from making dumb mistakes that go out and mess up those bytes floating around in memory. Making those mistakes is what leads to bugs due to memory safety. Instead of interpreting your password as a password, it's interpreted as your username and printed out to an attacker (oops!)

Rust provides safeguards against that in the form of a strong type system and tools like the borrow checker, but still all done at compile time. *Unsafe* Rust enables these dangerous tools with the tradeoff that the programmer is

now expected to uphold all the guarantees themselves, much like if they were writing C or C++ again.

See also:

- When does type binding happen in Rust?
- How does Rust implement reflection?
- How do I print the type of a variable in Rust?
- How to introspect all available methods and members of a Rust type?

Share Improve this answer Follow

edited Sep 25, 2019 at 19:11

answered Sep 25, 2019 at 18:25

Shepmaster
428k ● 111 ● 1.2k ● 1.5k

---

so a call to `count_ones` on a u32, when written to machine code, includes some instruction to act on bytes 0, 1, 2, 3 but not byte 4? is it more the compiled method (as an example of what you refer to as "the rest of the code") rather than the variable or value that contains this info? or perhaps the info isn't there at all - rather the method is parsed in the context of a given type at compile time, and the instruction is calculated, incl the number of bytes to act on, on a per-call basis, with no reference to a type anywhere in the compiled code – joel Sep 25, 2019 at 19:44

You can see the machine code yourself (click on the dots next to "build" and select "assembly"). Each of the instructions end in `l`, which is a mnemonic for "long", a.k.a. 32-bit operations. – Shepmaster Sep 25, 2019 at 19:49

To understand the machine code, you need to understand the Rust ABI, which isn't stable. In this particular case, the argument `v` is passed in the `eax` register, which is a 32-bit register, and the return value is returned in the same register. The code does not access the memory at all.

– Sven Marnach Sep 25, 2019 at 22:02