Rust generics

non-deterministic

🍑

Lukáš Hozda & Luukas Pörtfors

2025-11-20

Braiins Systems s.r.o

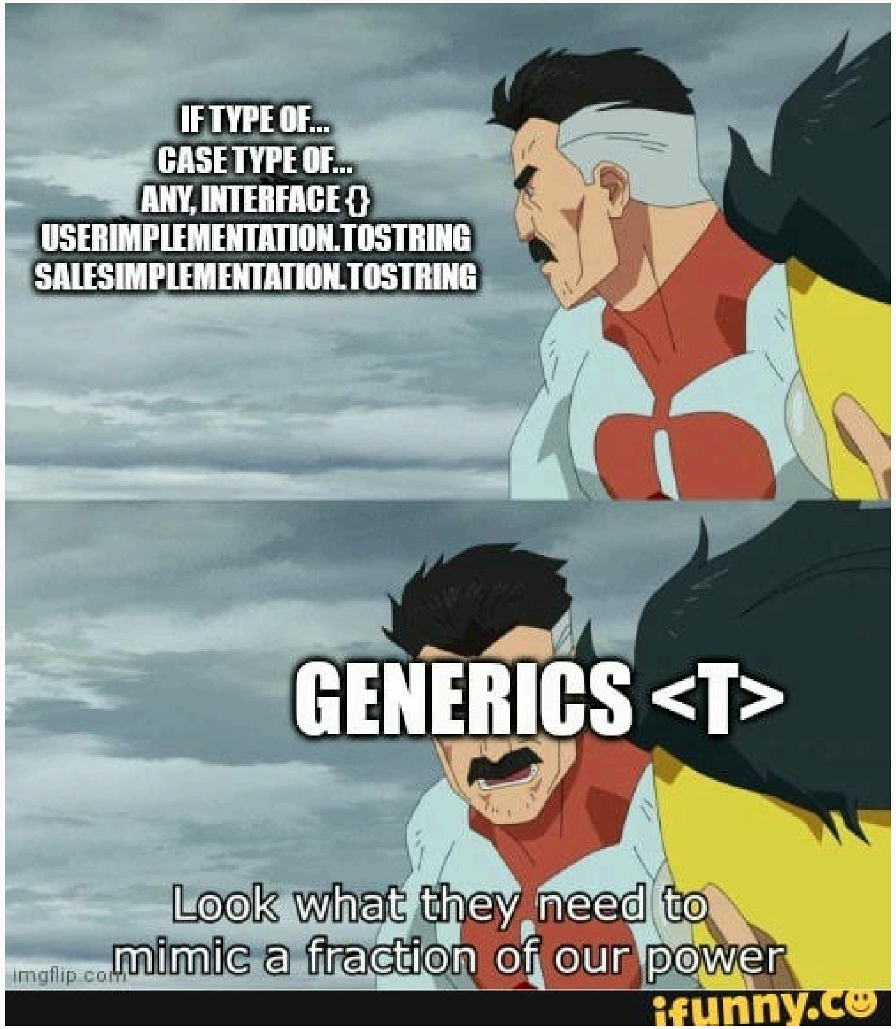
Generics and Traits in Rust

What are Generics?

- Allows writing code that works with multiple types
- Similar to templates in C++
- Zero-cost abstraction
- Compiler generates specialized code for each type
- Static dispatch

```rust
// Generic function
fn identity<T>(x: T) -> T {
    x
}

// Generic struct
struct Point<T> {
    x: T,
    y: T,
}
```


IF TYPE OF...
CASE TYPE OF...
ANY, INTERFACE {}
USERIMPLEMENTATION.TOSTRING
SALESIMPLEMENTATION.TOSTRING
GENERICS <T>
Look what they need to mimic a fraction of our power
imgflip.com
ifunny.co

Constrained Generics

- Rust doesn't have classes or inheritance (in the OOP sense)
  - instead we have *traits* (type classes)
- Traits define behavior for types
- Constrain generic types with trait bounds
- Specify what capabilities a type must have

```rust
// T must implement Display trait
// NOTE: The other way to do this:
// fn print_value(value: impl Display) {
fn print_value<T: Display>(value: T) {
    println!("{}", value);
}

// Multiple trait bounds
fn compare<T: PartialOrd + Clone>(a: T, b: T) -> T {
    if a > b { a } else { b }
}
```

Trait Objects

- Dynamic dispatch - available for *dyn compatible* traits
- Runtime polymorphism
- Use dyn keyword (is `!Sized`)
- Allows storing different types in same collection

```rust
trait Animal {
    fn make_sound(&self);
}

fn animal_sounds(animals: &[Box<dyn Animal>]) {
    for animal in animals {
        animal.make_sound();
    }
}
```

Default Trait Implementations

- Traits can provide default method implementations
- Types can override or use default behavior

```rust
trait Logger {
    fn log(&self, message: &str) {
        println!("Default log: {}", message);
    }
}

struct FileLogger;
impl Logger for FileLogger {
    // Uses default implementation
}
```

Derive Macros

- Automatically implement common traits
- Reduces boilerplate code
- Many standard library traits supported

```rust
#[derive(Debug, Clone, PartialEq)]
struct Person {
    name: String,
    age: u32,
}
```

ME
IT'S IMPOSSIBLE TO MOVE INTO A FUNCTION BECAUSE OF LARGE GENERIC TYPE PARAMETERS AND TRAIT BOUNDS
macro_rules! foo_bar { ...
imgflip.com

Generic Associated Types

- Allows defining associated types in traits
- Provides more flexibility in generic code

```rust
trait Container {
    type Item;
    fn contains(&self, item: &Self::Item) -> bool;
}

impl Container for Vec<i32> {
    type Item = i32;
    fn contains(&self, item: &i32) -> bool {
        self.contains(item)
    }
}
```

Trait Bounds in Method Definitions

- Constrain method type parameters
- Ensure methods only work with types that meet requirements

```
impl<T: Display + Clone> MyStruct<T> {
    fn print_value(&self, value: T) {
        println!("{}", value.clone());
    }
}
```

Where Clauses

- Alternative syntax for trait bounds
- More readable for complex constraints
- Can refer to anything, not just Self or params

```rust
fn process<T>(value: T)
where
    T: Display + Clone,
    T: PartialOrd
{
    // Method body
}
```

Dyn compatibility in Rust

What is Dyn compatibility?

- Formerly known as object safety
- Defines which traits can be used with trait objects
- Enables dynamic dispatch
- Requires specific rules to be followed

Dyn compatibility rules

- Method must not have any type parameters
- Method must not use `Self` except as return value
- Method must not have static method requirements
- We can use the `Self: Sized` bound to include methods that don’t match above requirements

```rust
// dyn-incompatible
trait NotSafe<T> {
    fn generic_method<U>(x: T, y: U);
    fn uses_self(self: Self) -> Self;
}

// Dyn compatible
trait Safe {
    fn concrete_method(&self, x: i32);
    fn returns_self(&self) -> Box<dyn Safe>;
}
```

Example of Dyn compatibility

```rust
// this works
trait Animal {
    fn make_sound(&self);
}

// this doesn't
trait Comparable<T> {
    fn compare(&self, other: &T) -> bool;
}

// can create trait objects for Animal
let animals: Vec<Box<dyn Animal>> = vec![];
```

Common traits

- `Iterator`:
  - implement the `next()`[1] method and get 75 other methods for free
- Often seen in the wild: `#[derive(Debug, Clone)]`
- `Display`
  - allows the value to be formatted (also automatically implements `ToString`)
- Operators like `Add` are traits
- `From` and `Into`
  - allow conversion between types, prefer implementing `From`
  - `From<T> for U` implies `Into<U> for T`
- `Drop`, the destructor trait in Rust

[1]https://doc.rust-lang.org/std/iter/index.html#implementing-iterator

Marker traits

- Interesting detail of the type system not seen outside Rust
- `Send`, `Sync`, `Copy`, `Sized`, `Unpin`
- Don’t include any methods
- Instead tell the compiler something about how the type behaves

Orphan rules

- prevent implementing a trait for a type unless either trait or the type is defined in your current crate
- They exist to enforce trait coherende[1]
  - there must be at most one implementation of a trait for any given type
- They can sometimes get in the way, easiest resolved by using wrappers

```
struct MyVec(Vec);

impl ForeignTrait for MyVec {}
```

[1]https://github.com/Ixrec/rust-orphan-rules?tab=readme-ov-file#what-is-coherence