Traits

Jonathan Dönszelmann & Vivian Roest

Delft University of Technology

2024-11-18

TU Delft

Last lecture:

- Pattern Matching
- Error Handling

Today:

- Generic Types
- Traits

Generics

A small example:

```rust
pub fn square(x: u64) -> u64 {
    x * x
}
```

What if we also want a `square` for `u32`?

Generics

The c way:

```rust
pub fn square_u64(x: u64) -> u64 {
    x * x
}

pub fn square_u32(x: u64) -> u64 {
    x * x
}

// ...
```

Question:

How many more things can we square?

Generics

Solutions

1. Use a macro to generate all these…
2. Use generics:

```rust
//            v-- for any type T
pub fn square<T>(x: T) -> T {
  //               ^-----^-- This function takes that type, and returns the same type
  x * x
}

assert_eq!(square(10u64), 100);
assert_eq!(square(10u32), 100);
// works for floats too!
assert_eq!(square(10.0), 100.0);
```

In C++ this is often called a “template”

Generics

How does this work?

- Functions with a *generic type parameter* are polymorphic
- Not actually a function, but a template
- For every *concrete type*, a new concrete function is made

“Monomorphization” – https://godbolt.org/z/KePKnsafr

```rust
// This function is "polymorphic"
pub fn square<T>(x: T) -> T {
  x * x
}
// monomorphizes once for floats
square(10.0);
// and once for ints
square(10);
```

Generics

- Wait a second...
- This was mostly an explanation of how templates work in C++
- Because this example does not actually work

```rust
// This function is "polymorphic"
pub fn square<T>(x: T) -> T {
    x * x
}
```

Question:

Why is this example broken?

Generics

```rust
// This function is "polymorphic"
pub fn square<T>(x: T) -> T {
  x * x
}

// fine, we can multiply floats
square(10.0);
// sure, integers too
square(10);
// what??
square("example")
```

Generics

In C++:

- Template functions are barely checked
- Type checks happen after a (template) function is called and monomorphized

Rust

```
// In C++ this function on its own would compile
pub fn square<T>(x: T) -> T {
  x * x // imagine this is some library call?
}
// fine, we can multiply floats
square(10.0);
// sure, integers too
square(10);
// only here the error happens
square("example")
```

Templates in C++: https://godbolt.org/z/bojsTfMns

Generics

In Rust:

- Template functions are checked **standalone**
- Even if a generic function is *never* monomorphized it can error

```rust
// In rust this fails to compile
pub fn square<T>(x: T) -> T {
    x * x
}

// even if we never instantiate the template
```

Templates in Rust: https://godbolt.org/z/G41Ta3neb

Generic types seem pretty much useless? We're still missing something...

Generics

How to fix: constrain `T`

```rust
pub fn square<T: Mul<Output=T>>(x: T) -> T {
    x * x
}
```

`T: Mul<Output = T>` means:

- Any type `T`
- as long as we can multiply it
- and the output after multiplication is also `T`

We call `Mul` a trait.

Traits

Traits

- Traits are *properties of types*
- Certain groups of types have similar properties
- A piece of behavior that a group of types has

Remember talking about types that add behavior?

A trait is behavior that multiple types *share*

Traits: Clone

- Anything that implements `Clone` must have a method `clone` with this signature

```rust
trait Clone {
    /// Returns a copy of the value.
    fn clone(&self) -> Self;
}
```

https://doc.rust-lang.org/stable/std/clone/trait.Clone.html

Traits: Clone

- Anything that implements `Clone` must have a method `clone` with this signature

```rust
trait Clone {
  /// Returns a copy of the value.
  fn clone(&self) -> Self;
}

impl Clone for Option {
  fn clone(&self) -> Self {
    // ...
  }
}
```

https://doc.rust-lang.org/stable/std/clone/trait.Clone.html

Traits: Clone

- Anything that implements `Clone` must have a method `clone` with this signature

```rust
//   v-- for any type T
impl<T> Clone for Option<T> {
  //    ^^^^^^^^^^^^^^^^^^^-- implement Clone for Option<T>
    fn clone(&self) -> Self {
      match self {
        None => None,
        Some(x) => Some(x.clone()),
      }
    }
}
```

Question:

Wait, is this correct?

Traits: Clone

Generics in impl blocks must also be bounded!

```rust
//   vvvvvvvv-- for any type T, that we can also Clone
impl<T: Clone> Clone for Option<T> {
  //           ^^^^^^^^^^^^^^^^^^^-- implement Clone for Option<T>
  fn clone(&self) -> Self {
    match self {
      None => None,
      //              vvvvvvvvv-- otherwise this would fail!
      Some(x) => Some(x.clone()),
    }
  }
}
```

Traits: Clone

Clone is actually a little more complicated

- To clone something, its size must be known at compile time
  - Constrained by another trait, `Sized`
- There is also `clone_from`
  - By default implemented in terms of `clone`
  - You might be able to implement it more efficiently...

```rust
pub trait Clone: Sized {
    fn clone(&self) -> Self;

    #[inline]
    fn clone_from(&mut self, source: &Self) {
        *self = source.clone()
    }
}
```

Traits: Mul

`Mul` defines how the `*` operator works

```rust
pub trait Mul<Rhs = Self> {
    type Output;

    fn mul(self, rhs: Rhs) -> Self::Output;
}
```

Also, `Div`, `Add`, `Sub`, `Neg`, etc...

Other Traits

Somewhere in the standard library:

```rust
impl Mul for u64 {
  type Output = u64;

  fn mul(self, rhs: u64) -> Self::Output {
    // some assembly maybe
  }
}
```

Traits

Remember our square function?

This might be a nicer implementation:

```rust
pub fn square<T: Mul>(x: T) -> <T as Mul>::Output {
    x * x
}
```

Question:

When does multiplying **not** produce the same output as input?

Assignment: 10 minutes

Build a Vector

- Use this template and implement multiplication for this Vector

Template:

```rust
struct Vector {
    x: f64,
    y: f64,
}
```

- Vector-Vector multiplication is a dot product producing a float
- Vector-float multiplication scales the vector, producing a vector
- Create an example that demonstrates this by printing the result

Traits

Question:

Have you already seen traits in other places maybe?

Important Traits: Conversion

```rust
pub trait Into<T>: Sized {
    fn into(self) -> T;
}

pub trait From<T>: Sized {
    fn from(value: T) -> Self;
}

impl<T, U> Into<U> for T where U: From<T> {
    fn into(self) -> U {
        U::from(self)
    }
}
```

https://doc.rust-lang.org/stable/std/convert/trait.From.html

Important Traits: Dereferencing

```rust
pub trait Deref {
  type Target: ?Sized;

  fn deref(&self) -> &Self::Target;
}
```

- `Box` is just a struct
- But because it implements `Deref` it can be used like a pointer!

https://doc.rust-lang.org/stable/std/ops/trait.Deref.html

Important Traits: Equality

- == operator

```rust
pub trait PartialEq<Rhs = Self>
where
    Rhs: ?Sized,
{
    fn eq(&self, other: &Rhs) -> bool;

    fn ne(&self, other: &Rhs) -> bool {
      !self.eq(other)
    }
}
```

https://doc.rust-lang.org/stable/std/cmp/trait.PartialEq.html

Important Traits: Formatting

```rust
pub trait Display {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

https://doc.rust-lang.org/stable/std/fmt/trait.Display.html

Assignment: 10 minutes

Build a Vector

- Go back to your Vector implementation
- Implement `Display` for vectors, to make them printable

Hint:

Use https://docs.rs if you need some help

- Time left over? Also implement `Clone`

Important Traits: Drop

- Determines what to do when a value goes out of scope.
- Recursively called
- This is where `Vec` frees its elements

```rust
pub trait Drop {
    fn drop(&mut self);
}
```

https://doc.rust-lang.org/stable/std/ops/trait.Drop.html

Vec:

- https://doc.rust-lang.org/1.81.0/src/alloc/vec/mod.rs.html#398
- https://doc.rust-lang.org/1.81.0/src/alloc/raw_vec.rs.html#596
- https://doc.rust-lang.org/1.81.0/src/alloc/vec/mod.rs.html#3302

Deriving

Standard pattern for Clone:

- To implement `Clone` for a `struct`
- `Clone` each element

Or for Equality

- A `struct` is equal to another
- If all fields are equal

Deriving

```Rust
#[derive(
    PartialEq, Eq,   // equality checking
    PartialOrd, Ord, // ordering (a > b)

    Copy, Clone, // cloning (and implicit cloning)

    Hash, // hashing for `HashMap`
    Debug, // debug printing
    Default // initialize to zero
)]
pub struct Vector {
  x: f64,
  y: f64,
}
```

Note: also works on `enums`

Assignment: 5 minutes

- Write a function called `add_all`
- It takes two vectors of different types (`Vec<A>` and `Vec<B>`)
- It returns the elementwise sum of the two vectors
- Use the right generic bound

Dynamic Dispatch

- Monomorphization can be expensive
  - Code size
  - Compile time

```Rust
fn print_thing<D: Display>(i: D) {
  println!("the thing is {i}");
}

print_thing(3);
print_thing(3u64);
print_thing(3.0);
print_thing("hi");
```

https://godbolt.org/z/61E665xc5

- But they're all doing pretty much the same thing (call `Display::fmt`)

Dynamic Dispatch

What if we could pass a function pointer that we just call at the right time?

```rust
fn print_thing(print_fn: fn()) {
  print!("the thing is");
  print_fn();
  println!()
}

fn print_3() {
  print!("3")
}
// ...
print_thing(print_3);
print_thing(print_3u64);
print_thing(print_3f64);
print_thing(print_hi);
```

Dynamic Dispatch

Uses a vtable (runtime type description) to create a “trait object”

```rust
fn print_thing(x: &dyn Display) {
  println!("the thing is {i}");
}

print_thing(&3);
print_thing(&3u64);
print_thing(&3.0);
print_thing(&"hi");
```

https://godbolt.org/z/hcqGdW8z8

Question:

Why do we need the reference?

Dynamic Dispatch

Uses a vtable

```Rust
fn print_thing(x: &dyn Display) {
  println!("the thing is {i}");
}

print_thing(&3);
print_thing(&3u64);
print_thing(&3.0);
print_thing(&"hi");
```

Question:

What are the disadvantages?

Summary

- Generics create patterns of items
- They can be instantiated with concrete types
- The types used can be constrained by traits
- Traits can also be used as standalone objects

Next week: the `Iterator` trait!