Unsized Coercion & Scoped Threads

Colm Oneacre

Sized vs Unsized Types

Sized – known at compile time (i32, arrays [T; n])

Unsized – unknown at compile time (str, [T], dyn Trait)

Rust normally requires sizes at compile time, but some types don't have a fixed size

Why Unsized Types Matter

Unsized types do a lot for us:

- Handle variable-length data
- Enable polymorphism
- Make API's flexible

Unsized Coercion

Automatic conversion: Sized -> Unsized

- Only happens behind pointers

Unsized Values must always be used via references or smart pointers

- Rust converts types when needed behind the scenes in safe context

Slicing an array

Most common example of unsized coercion

Let arr = [1, 2, 3];

Let slice: &[i32] = &arr;

[i32; 3] -> [i32]

String -> str

String is sized, but *str* js dynamically sized (Rust handles conversion)

Let s = String::from("hello");

Let slice: &str = &s;

Trait Object Coercion

This allows different types to be treated uniformly using dynamic dispatch

```
Trait Animal {
        fn speak(&self);
}
```

Let animal: &dyn Animal = &dog

Rules of Coercion

Happens automatically, only via pointers (&, Box, etc,)

You will rearely write coercions manually – they're built into the language

Why does it matter?

- Enables slices (&[T])
- Enables trait objects (dyn Trait)
- Supports flexible, reusable code

Rules of Coercion

```
Use std::thread;
Let v = vec![1, 2, 3];
Thread::spawn ( || {
        println!("{:?}", v);
});
```

Rust prevents this because the thread could run after v is dropped

What Are Scoped Threads?

Threads that cannot outlive their scope

- Introduced via std::thread::scope

Scoped threads fix the lifetime problem by guarnateeing threads finish before The scope ends

Basic Example

```
Use std::thread;
Let v = vec![1, 2, 3];

Thread::scope(|s| {
  spawn(|| {
    println!("{:?}", v);
  });
});
```

The thread is guaranteed to finish before leaving the scope, so borrowing is safe.

Key Advantages

No 'static' lifetime required

Can borrow local variables

Safer and more felxible

This makes much more sense than forcing everything to be 'static

Multiple Scoped Threads

```
Thread::scope(|s| {
    s.spawn(|| println!("Thread 1"));
    s.spawn(|| println!("Thread2"));
});
```

You can spawn multiple threads, and all will complete before the scope exits

Safety

Compiler tracks lifetimes

Scope guarantees thread completion

Prevents dangling references

This is rust's ownership model applied to concurrency

When to Use Scoped Threads

When borrowing local data

When avoiding Arc/Mutex overhead

When threads are short-lived

Scoped threads are ideal for temporary parallel work.

Recap

- Unsized coercion:
  - Enables slices and trait objects
- Scoped threads:
  - Allow safe borrowing across threads
- Both improve flexibilty without sacrificing safety

- Great ways that Rust balances power and safety