Why does a program compile despite an apparent lifetime mismatch?

Asked 1 year, 6 months ago
Active 1 year, 5 months ago
Viewed 229 times

11

2

Given the following Rust program:

```
struct Value<'v>(&'v ());
struct Container {}

impl Container {
    fn get<'v>(&'v self) -> Value<'v> {
        todo!()
    }

    fn set<'v>(&'v self, x: Value<'v>) {
        todo!()
    }
}

fn convert<'v1, 'v2>(x: &'v1 Container, env: &'v2 Container) {
    let root: Value<'v2> = env.get();
    x.set(root);
}
```

I would expect `convert` to be a compile time error as `Value<'v2>` gets passed to `x.set()` which requires a value of type `Value<'v1>` - but it successfully compiles. There is no subtyping relationship between `'v1` and `'v2`. How has Rust inferred satisfying lifetimes?

rust lifetime

edited Jun 30 '20 at 14:17

Shepmaster
305k • 59 • 824 • 1083

asked Jun 30 '20 at 14:01

Neil Mitchell
8,808 • 1 • 25 • 78

There's no way you can implement `get` and `set` to make `convert` unsound. Try it.
– trent formerly cl
Jun 30 '20 at 15:30

@trentcl not without using unsafe. But in the case I'm writing, I am using unsafe, but "safely", and it is the case that convert is unsound (but I appreciate that's just reflecting that my unsafe isn't quite safe enough).
– Neil Mitchell
Jun 30 '20 at 15:56

1 Answer

11

The compiler is always allowed to re-borrow with a shorter lifetime.

In this case, what happens in:

```
fn convert<'v1, 'v2>(x: &'v1 Container, env: &'v2 Container) {
    let root: Value<'v2> = env.get();
    x.set(root);
}
```

Is that the compiler reborrows `x` (aka `(&*x)`) with a lifetime `'v3`, shorter than `'v2`, which is allowed due to the (inferred) variance of `Value<'v>` (which matches `&'v T`).

It is possible to change the inferred variance of `Value<'v>` by changing the inner value:

- `&'v ()` (the current) is **covariant**.
- `Cell<&'v ()>` is **invariant**.
- `fn (&'v ()) -> ()` is **contravariant**, the inverse of *covariant*.

Using an *invariant* `Value<'v>` prevents unifying the lifetime with a fresh one, while using a *contravariant* `Value<'v>` only allows unifying the lifetime with a *greater* one.

Share
Improve this answer
Follow
edited Jul 1 '20 at 14:51

answered Jun 30 '20 at 15:36

Matthieu M.
261k ● 40 ● 396 ● 665

1

Thanks! And to stop the variance, I defined Value as `Cell<&'v ()>` which then causes a compile-time mismatch.
– Neil Mitchell
Jun 30 '20 at 15:56

2

I meant contra, so I'll sneakily remove my comment and integrate it in the answer... with the fix.
– Matthieu M.
Jul 1 '20 at 14:48

Add a comment

Your Answer

Post Your Answer

By clicking "Post Your Answer", you agree to our terms of service, privacy policy and cookie policy

Not the answer you're looking for? Browse other questions tagged rust lifetime or ask your own question.

Related

Why does the lifetime name appear as part of the function type?

Does <'a, 'b: 'a> mean that the lifetime 'b must outlive the lifetime 'a?

Why does my trait need a lifetime parameter?

Lifetime constraints to model scoped garbage collection

Is it possible to store a Rust struct containing a closure in a different struct?

Why does the Rust compiler not optimize code assuming that two mutable references cannot alias?

Why does this lifetime not "expire"?

Understanding lifetimes: max lifetime and 'static

Lifetime issue with guard pattern on struct with RefCell

Hot Network Questions

- How to salvage bitter homemade mustard?
- Who (or what) created the atropal?
- How to plot molecules with angles and bond lengths
- How to convince clan leaders and Party Cadres to give up their power?
- Why is 20m band waterfall showing signals every 50 kHz?

more hot questions

Question feed