Help understanding error: `'a` must outlive `'static`

help

vhborges January 15, 2025, 9:45pm 1

The following simplified code:

```
trait Tr {
   fn foo(self) -> Box<dyn Tr>;
}

struct Foo<'a> {
   member: &'a str
}

struct Bar<'a> {
   member: &'a str
}

impl<'a> Tr for Foo<'a> {
   fn foo(self) -> Box<dyn Tr> {
       Box::new(Bar {
           member: self.member
       })
   }
}

impl<'a> Tr for Bar<'a> {
   fn foo(self) -> Box<dyn Tr> {
       Box::new(Foo {
           member: self.member
       })
   }
}
```

Produces the following errors:

```
error: lifetime may not live long enough
 --> src/lib.rs:15:9
  |
13 |   impl<'a> Tr for Foo<'a> {
  |        -- lifetime `'a` defined here
14 |       fn foo(self) -> Box<dyn Tr> {
15 | /         Box::new(Bar {
16 | |             member: self.member
17 | |         })
  | |__________^ returning this value requires that `'a` must outlive `'static`

error: lifetime may not live long enough
 --> src/lib.rs:23:9
  |
21 |   impl<'a> Tr for Bar<'a> {
  |        -- lifetime `'a` defined here
22 |       fn foo(self) -> Box<dyn Tr> {
23 | /         Box::new(Foo {
24 | |             member: self.member
25 | |         })
  | |__________^ returning this value requires that `'a` must outlive `'static`
```

Why would `'a` need to outlive `'static`?
What could I do to fix it?

quinedot January 15, 2025, 9:51pm 2

vhborges:

> Why would `'a` need to outlive `'static`?

Outside of function bodies, `Box<dyn Trait>` is sugar for `Box<dyn Trait + 'static>`. The lifetime is needed so that the validity of the erased type, `Foo<'a>` for example, can be checked. If you could coerce `Foo<'a>` to `dyn Tr + 'static`, you could easily end up with a dangling reference -- there would be no lifetime information to make sure that doesn't happen.

More about dyn lifetimes.

vhborges:

> What could I do to fix it?

Here's one possibility:

```
trait Tr {
   fn foo<'a>(self) -> Box<dyn Tr + 'a> where Self: 'a;
}
```

play.rust-lang.org

Rust Playground

A browser interface to the Rust compiler to experiment with the language

2 Likes

undefined.behavior January 16, 2025, 1:33am 3

I was somewhat surprised that this also works

```
-    fn foo<'a>(self) -> Box<dyn Tr + 'a> where Self: 'a {
+    fn foo<'a>(self) -> Box<dyn Tr + 'a> where 'foo: 'a {
```

I understand that his is implied by `Self: 'a`, but was expecting the exact spelling as in the trait.

quinedot January 16, 2025, 1:56am 4

It will accept anything "at least as strong" as the trait definition, yep.[1] This has been true forever[2] to support things like...

```
// AKA PartialEq<Foo<'_>> for Foo<'_>, but the below syntax is accepted
// even on edition 1.0.0
impl<'a, 'b> PartialEq<Foo<'b>> for Foo<'a> {
   fn eq(&self, other: &Foo) -> bool {
       self.member.eq(other.member)
   }
}
```

where `other` is not exactly `Foo<'b>`. However, for those historical cases, downstream cannot take advantage of the "stronger" implementation details.

We now also have refinement, which does allow downstream to use the stronger implementation details. But I haven't played around to know if there's a way to opt in to refinements that have been historically allowed but don't trigger the refinement lint, like the example above.[3] (Related.[4])

(Your example is not more refined than the trait definition.)

> There are some nuances, things which aren't accepted for various reasons. ■■
>
> or close enough, more things have become accepted over time ■■
>
> A very simple attempt implied you can't so far. ■■
>
> Edit: Eh, not that related, I just noticed that issue is about `-> impl Trait` only. ■■

1 Like

vhborges January 16, 2025, 11:43am 5

quinedot:

> Outside of function bodies, `Box<dyn Trait>` is sugar for `Box<dyn Trait + 'static>`. The lifetime is needed so that the validity of the erased type, `Foo<'a>` for example, can be checked. If you could coerce `Foo<'a>` to `dyn Tr + 'static`, you could easily end up with a dangling reference -- there would be no lifetime information to make sure that doesn't happen.

It's clear now, thanks!

quinedot:

> Here's one possibility:

```
trait Tr {
   fn foo<'a>(self) -> Box<dyn Tr + 'a> where Self: 'a;
}
```

Rust Playground

Just to be clearer, considering the following snippet from your code:

```
impl<'foo> Tr for Foo<'foo> {
   fn foo<'a>(self) -> Box<dyn Tr + 'a> where Self: 'a {
       Box::new(Bar {
           member: self.member
       })
   }
}
```

The reason you gave another name for the lifetime in `impl<'foo> Tr for Foo<'foo>` was to be able to constraint `Foo`'s lifetime to outlive `Box<dyn Tr>`'s lifetime?
In other words, would `Self: 'a` be equivalent to `'foo: 'a`? (Of course we couldn't write in that way because `'foo` isn't defined for `Tr`)

ekuber January 16, 2025, 5:40pm 6

The other option here is to give the trait a lifetime param, but it is never as nice as one would hope:

```
trait Tr<'a> {
   fn foo(self) -> Box<dyn Tr<'a> + 'a>;
}
impl<'foo> Tr<'foo> for Foo<'foo> {
   fn foo(self) -> Box<dyn Tr<'foo> + 'foo> {
       Box::new(Bar {
           member: self.member
       })
   }
}
```

philomathic_life January 16, 2025, 6:06pm 7

I was going to recommend the same thing. Admittedly, I still find myself struggling when I should make a `trait` generic over a lifetime vs. making the method(s) generic over a lifetime. I think this may at least be in part due to all the lifetime "magic" that Rust does (e.g., subtyping, reborrows, etc.).

Conceptually, I understand that `trait Foo<'a>` means the *implementor* is in control of the lifetime in contrast to when the method is generic where the *caller* is in control; yet I find myself questioning which way I should define the `trait`. Sometimes it takes several implementations of a `trait` for me to *finally* realize why I should prefer one over the other (or even use a GAT). For "obvious" cases like defining a method that takes a reference that the implementing type should not care about the lifetime, then it's obvious to make the method generic; but there are less obvious cases.

Here since `Tr::foo` takes `self` and the desire is to have ability for implementing types to be based on references that can return said reference, I *feel* like `Tr<'a>` is more appropriate, but I could very well be wrong. It seems similar to why something like `serde::de::Deserializer` is generic over a lifetime instead of the methods being generic.

quinedot January 16, 2025, 7:47pm 8

vhborges:

> The reason you gave another name for the lifetime in `impl<'foo> Tr for Foo<'foo>` was to be able to constraint `Foo`'s lifetime to outlive `Box<dyn Tr>`'s lifetime?

I wrote it that way because I hammered everything out with `'a` mechanically,[1] got chastised about `'a` already being in use, and replaced the parameters on `Foo` with `'foo` and on `Bar` with `'bar` in the implementation (but was too lazy to go make the `structs` match). More consistent and meaningful lifetime naming would be better discipline.

vhborges:

> In other words, would `Self: 'a` be equivalent to `'foo: 'a`? (Of course we couldn't write in that way because `'foo` isn't defined for `Tr`)

The bound would be the same; the breakdown is something like

```
Self: 'a

Foo<'foo>: 'a // Self is just an alias we replaced

'foo: 'a      // Outlives bounds are syntactical, and in the case of structs
             // this means recursively applying to all parameters
```

And you can write it that way (like @undefined.behavior showed), or you can even write...

```
impl<'foo> Tr for Foo<'foo> {
   fn foo<'a>(self) -> Box<dyn Tr + 'foo> where 'foo: 'a {
```

...in which case the implementation is "stronger" than the trait actually requires, but not usable as such, as I referred to above. (So I don't really recommend it.)

There's no way to omit the second lifetime on `foo` completely with this approach.

ekuber:

> but it is never as nice as one would hope

What would be really nice IMO if is there was a `'limit<..>` lifetime or such that equated to the intersection of the lifetimes meetable within, like what happens with `dyn` lifetimes automatically inside a function body.

```
trait Tr {
   fn foo(self) -> Box<dyn Tr + 'limit<Self>>;
}
```

Then `'limit<dyn Trait<'all, The, Params> + 'static>` could be a better default lifetime for `Box<dyn Trait<'all, The, Params>>` than `'static` as well.

I've wanted that for other scenarios that just `dyn`, too.

> copying the signature from the trait ■■

philomathic_life January 16, 2025, 8:11pm 9

You could also more succinctly/idiomatically elide the lifetime too, right? For example, `impl Tr for Foo<'_>…` is the same thing.

1 Like

quinedot January 16, 2025, 8:21pm 10

Yes, that's true.

system Closed April 16, 2025, 8:22pm 11

This topic was automatically closed 90 days after the last reply. We invite you to open a new topic if you have further questions or comments.

Related topics

Topic		Replies	Views	Activity
Why 'a must outlive 'static here? help	8	4741	August 11, 2023	
Why does `Box<dyn Trait>` introduce a `static lifetime? help	8	1418	October 16, 2024	
Lifetime 'static confusion	7	619	May 22, 2023	
Understanding lifetimes of Box<dyn SomeTrait> help	3	3751	January 21, 2021	
[SOLVED] Trait with lifetime parameter and compiler message I don't understand help	3	2191	January 12, 2023	