



Products

Linking the lifetimes of self and a reference in method

Log in Sign up

Ask Question

Asked 6 years, 7 months ago

Active 4 years, 3 months ago

Viewed 2k times



12



3



I have [this piece of code](#):

```
#[derive(Debug)]
struct Foo<'a> {
    x: &'a i32,
}

impl<'a> Foo<'a> {
    fn set(&mut self, r: &'a i32) {
        self.x = r;
    }
}

fn main() {
    let v = 5;
    let w = 7;
    let mut f = Foo { x: &v };

    println!("f is {:?}", f);

    f.set(&w);

    println!("now f is {:?}", f);
}
```

My understanding is that in the first borrow of the value of `v`, the generic lifetime parameter `'a` on the struct declaration is filled in with the lifetime of the value of `v`. This means that the resulting `Foo` object must not live longer than this `'a` lifetime or that the value of `v` must live at least as long as the `Foo` object.

In the call to the method `set`, the lifetime parameter on the `impl` block is used and the lifetime of the value of `w` is filled in for `'a` in the method signature. `&mut self` is assigned a different lifetime by the compiler, which is the lifetime of `f` (the `Foo` object). If I switched the order of the bindings of `w` and `f` in the `main` function, this would result in an error.

I wondered what would happen if I annotated the `&mut self` reference with the same lifetime parameter `'a` as `r` in the `set` method:

```
impl<'a> Foo<'a> {
    fn set(&'a mut self, r: &'a i32) {
        self.x = r;
    }
}
```

Which results in the following error:

```
error[E0502]: cannot borrow `f` as immutable because it is also borrowed as mutable
--> src/main.rs:21:31
|
| 19 |     f.set(&w);
|    - mutable borrow occurs here
| 20 |
| 21 |     println!("now f is {:?}", f);
|                                ^ immutable borrow occurs here
| 22 | }
|    - mutable borrow ends here
```

In contrast to the example above, `f` is still considered mutably borrowed by the time the second `println!` is called, so it cannot be borrowed simultaneously as immutable.

How did this come to be?

By not leaving off the lifetime annotation the compiler filled one in for me for `&mut self` in the first example. This happens by the rules of lifetime elision. However by explicitly setting it to `'a` in the second example I linked the lifetimes of the value of `f` and the value of `w`.

Is `f` considered borrowed by itself somehow?

And if so, what is the scope of the borrow? Is it `min(lifetime of f, lifetime of w) -> lifetime of f`?

I assume I haven't fully understood the `&mut self` reference in the function call yet. I mean, the function returns, but `f` is still considered to be borrowed.

I am trying to fully understand lifetimes. I am primarily looking for corrective feedback on my understanding of the concepts. I am grateful for every bit of advice and further clarification.

rust

Share

Improve this question

Follow

Your privacy

By clicking "Accept all cookies", you agree Stack Exchange can store cookies on your device and disclose information in accordance with our [Cookie Policy](#).



Accept all cookies Customize settings

305k ● 59 ● 824 ● 1083

asked May 16 '15 at 9:11



[jtepe](#)

3,002 ● 2 ● 18 ● 30

[Add a comment](#)

1 Answer

[Active](#) [Oldest](#) [Votes](#)



6



In the call to the method `set` the lifetime parameter on the impl block is used and the lifetime of the value of `w` is filled in for `'a` in the method signature.

No. The value of the lifetime parameter `'a` is fixed at the creation of the `Foo` struct, and will never change as it is part of its type.

In your case, the compiler actually choses for `'a` a value that is compatible with both the lifetimes of `v` and `w`. If that was not possible, it would fail, such as in this example:

```
fn main() {
    let v = 5;
    let mut f = Foo { x: &v };

    println!("f is {:?}", f);
    let w = 7;
    f.set(&w);

    println!("now f is {:?}", f);
}
```

which outputs:

```
error[E0597]: `w` does not live long enough
  --> src/main.rs:21:1
   |
18 |     f.set(&w);
   |         - borrow occurs here
...
21 | }
   |   ^ `w` dropped here while still borrowed
   |
   = note: values in a scope are dropped in the opposite order they are created
```

Exactly because the `'a` lifetime imposed by `v` is not compatible with the shorter lifetime of `w`.

In the second example, by forcing the lifetime of `self` to be `'a` as well, you are tying the mutable borrow to the lifetime `'a` as well, and thus the borrow ends when all items of lifetime `'a` goes out of scope, namely `v` and `w`.

[Share](#)

[Improve this answer](#)

[Follow](#)

edited Sep 24 '17 at 19:56



[Shepmaster](#)

305k ● 59 ● 824 ● 1083

answered May 16 '15 at 11:13



[Levans](#)

12.3k ● 3 ● 43 ● 50

Interesting. So in the first example, if the value is fixed at creation the compiler also takes the assignment in the method call into account to determine `'a`. But the `'a` still has to be at least as long as the lifetime of `f`.

– [jtepe](#)

May 16 '15 at 12:12

@JonasTepe Yes, that's it.

– [Levans](#)

May 16 '15 at 12:17

And from that, the second example also makes perfect sense now. Thank you very much.

– [jtepe](#)

May 16 '15 at 12:24

1

@Levans the example now compiles

– [rethab](#)

Dec 17 '20 at 9:41

[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](#) or [ask your own question](#).

The Overflow Blog

-  Sequencing your DNA with a USB dongle and open source code
-  Don't push that button: Exploring the software that flies SpaceX rockets and...

Featured on Meta

-  Providing a JavaScript API for userscripts
-  Congratulations to the 59 sites that just left Beta

Linked

Mutable borrow in loop

Borrow checker won't let me call getter after calling setter

1
Lifetime of references passed to function stops subsequent access

What are the differences between specifying lifetime parameters on an impl or on a method?

How to ensure end of immutable borrow after function call in order to enable mutable borrow?
8
Mutable borrow in a loop

Is it possible to mutate a struct's field inside a loop?

Borrowing an object as mutable twice for unrelated, sequential uses

Related

6
cannot move out of borrowed content when unwrapping a member variable in a &mut self method

How to implement a trait for any mutability?






How do I return a reference to something inside a RefCell without breaking encapsulation?

Why does linking lifetimes matter only with mutable references?
1
Lifetime in recursive struct with mutable reference

1
mutably borrow fields from a mutably borrowed struct

why is rust 'pub fn func(&a mut self)' considered "mutably borrowed" after run?
0
immutable and mutable reference needed for object inside struct

Hot Network Questions

-  Why doesn't dkim sign the letter?
 -  What does this entry on the Rocinante's pilot quick-menu mean?
 -  Using a friend to move cash into my checking account
 -  I've got a material setup that blends two shaders decently, but how would I apply it to a circular target?
 -  Why does the prophecy imply Macbeth has to murder the king?
- [more hot questions](#)

STACKOVERFLOW

[Questions](#)
[Jobs](#)
[Developer Jobs Directory](#)
[Salary Calculator](#)
[Help](#)
[Mobile](#)

PRODUCTS

[Teams](#)
[Talent](#)
[Advertising](#)
[Enterprise](#)

COMPANY

[About](#)
[Press](#)
[Work Here](#)
[Legal](#)
[Privacy Policy](#)
[Terms of Service](#)
[Contact Us](#)
[Cookie Settings](#)
[Cookie Policy](#)

STACK EXCHANGE NETWORK

[Technology](#)
[Culture & recreation](#)
[Life & arts](#)
[Science](#)
[Professional](#)
[Business](#)
[API](#)
[Data](#)

[Blog](#)
[Facebook](#)
[Twitter](#)
[LinkedIn](#)
[Instagram](#)

site design / logo © 2021 Stack Exchange Inc; user contributions licensed under [cc by-sa](#). rev 2021.12.22.41046