Who borrowed a variable?

Asked 5 years, 2 months ago
Active 3 years, 3 months ago
Viewed 665 times

12

3

I'm fighting with the borrow checker. I have two similar pieces of code, one working as I expect, and the other not.

The one that works as I expect:

```
mod case1 {
  struct Foo {}

  struct Bar1 {
    x: Foo,
  }

  impl Bar1 {
    fn f<'a>(&'a mut self) -> &'a Foo {
      &self.x
    }
  }

  // only for example
  fn f1() {
    let mut bar = Bar1 { x: Foo {} };
    let y = bar.f(); // (1) 'bar' is borrowed by 'y'
    let z = bar.f();  // error (as expected) : cannot borrow `bar` as mutable more
                      // than once at a time [E0499]
  }

  fn f2() {
    let mut bar = Bar1 { x: Foo {} };
    bar.f(); // (2) 'bar' is not borrowed after the call
    let z = bar.f();  // ok (as expected)
  }
}
```

The one that doesn't:

```
mod case2 {
  struct Foo {}

  struct Bar2<'b> {
    x: &'b Foo,
  }

  impl<'b> Bar2<'b> {
    fn f(&'b mut self) -> &'b Foo {
      self.x
    }
  }

  fn f4() {
    let foo = Foo {};
    let mut bar2 = Bar2 { x: &foo };
    bar2.f(); // (3) 'bar2' is borrowed as mutable, but who borrowed it?
    let z = bar2.f(); // error: cannot borrow `bar2` as mutable more than once at a time [E0499]
  }
}
```

I hoped I could call `Bar2::f` twice without irritating the compiler, as in case 1.

The question is in the comment (3): who borrowed `bar2`, whereas there is no affectation?

Here's what I understand:

1. In case 1, `f2` call: the lifetime parameter `'a` is the one of the receiving `&Foo` value, so this lifetime is empty when there is no affectation, and `bar` is not borrowed after the `Bar1::f` call;
2. In case 2, `bar2` borrows `foo` (as immutable), so the lifetime parameter `'b` in `Bar2` struct is the `foo` reference lifetime, which ends at the end of `f4` body. Calling `Bar2::f` borrows `bar2` for that lifetime, namely to the end of `f4`.

But the question is still: who borrowed `bar2`? Could it be `Bar2::f`? How `Bar2::f` would hold the borrowed ownership after the call? What am I missing here?

I'm using Rust 1.14.0-nightly (86affcdf6 2016-09-28) on x86_64-pc-windows-msvc.

rust lifetime borrow-checker

edited Oct 3 '16 at 13:29

Shepmaster
305k • 59 • 824 • 1083

asked Oct 3 '16 at 8:20

jferard

6,932 ● 1 ● 14 ● 29

Add a comment

4 Answers

Active Oldest Votes

9

Ah... you basically self-borrowed yourself.

The issue hinges on the fact that you have the same lifetime (`'b`) used for both the lifetime of `Foo` and the lifetime of `Bar` . The compiler then dutifully unifies those lifetimes, and you end up in a strange situation where suddenly the lifetime of the borrow which should have ended at the end of the statement instead ends after the value should have gone out of scope.

As a rule of thumb: *always* use a fresh lifetime for `self` . Anything else is weird.

It's interesting to note that this pattern can actually be useful (though more likely with an immutable borrow): it allows *anchoring* a value to a stack frame, preventing any move after the call to the function, which is (sometimes) useful to represent a borrow that is not well-modeled by Rust (like passing a pointer to the value to FFI).

Share

Improve this answer

Follow

answered Oct 3 '16 at 9:38

Matthieu M.

261k ● 40 ● 396 ● 665

Accepted because of this valuable rule of thumb. Thanks.

– jferard

Oct 5 '16 at 18:34

Add a comment

9

In case #2, you have this:

```
impl<'b> Bar2<'b> {
    fn f(&'b mut self) -> &'b Foo {
        self.x
    }
}
```

To highlight: `&'b mut self` and `&'b Foo` have the same lifetime specified.

This is saying that the reference to `self` and the returned reference to an instance of a `Foo` both have the same lifetime. Looking at the call site, you have this:

```
let foo = Foo {};
let mut bar2 = Bar2 { x: &foo };
```

So the compiler is inferring that both `foo` and `bar2` have the same lifetime. The lifetime of `foo` is the scope of the `f4` function, and so the mutable reference to `bar2` shares this.

One way to fix this, is to remove the explicit lifetime on the `self` reference:

```
fn f(&mut self) -> &'b Foo
```

This compiles and the compiler correctly understands that the reference to `bar2` and the reference to `foo` have different lifetimes.

Playground: https://play.rust-lang.org/?gist=caf262dd628cf14cc2884a3af842276a&version=stable&backtrace=0

TLDR: Yes, having the same lifetime specifier on the self reference and the returned reference means that the entire scope of `f4` holds a mutable borrow of `bar2` .

Share

Improve this answer

Follow

answered Oct 3 '16 at 9:35

Simon Whitehead

59.3k ● 7 ● 101 ● 130

Add a comment

1

I put the body of `f4()` in a `main()` and implemented `Drop` for `Bar2` to find out when it is dropped (i.e. goes out of scope):

```
impl<'b> Drop for Bar2<'b> {
    fn drop(&mut self) { println!("dropping Bar2!"); }
}
```

And the result was:

```
error: `bar2` does not live long enough
  --> <anon>:24:5
   |
24 |     bar2.f();
   |     ^^^^ does not live long enough
25 | }
   | - borrowed value dropped before borrower
   |
   = note: values in a scope are dropped in the opposite order they are created
```

Something's fishy; let's examine it in detail, with helper scopes:

```
fn main() {
    {
        let foo = Foo {}; // foo scope begins
        {
            let mut bar2 = Bar2 { x: &foo }; // bar2 scope begins; bar2 borrows foo
            bar2.f();
        } // bar2 should be dropped here, but it has the same lifetime as foo, which is still live
    } // foo is dropped (its scope ends)
}
```

It looks to me that there is a leak here and `bar2` is never dropped (and thus `Drop` cannot be implemented for it). That's why you cannot re-borrow it.

Share
Improve this answer
Follow
edited Oct 7 '16 at 3:01

answered Oct 3 '16 at 9:23

ljedrz
17.1k • 3 • 60 • 81

Add a comment

1

I would like to add about the roles that subtyping/variance play here.

`&mut T` is invariant over `T`. Given two types `T` and `U`, where `T < U` (`T` is a subtype of `U`), then `&mut T` has no subtyping relation with `&mut U` (i.e they are invariant with each other), whereas `&T` is a subtype of `&U` (`&T < &U`). But `&'lifetime` and `&'lifetime mut`, both are covariant over `'lifetime`. So given two lifetimes `'a` and `'b` for a type `T`, where `'a` outlives `'b`, then as per subtyping relation `&'a T < &'b T`, similarly `&'a mut T < &'b mut T`

Coming to the question, in the call to function `f`, `self` is a reference to `Bar2<'a>`. The compiler will see if it can "temporarily shorten" the life of `bar2` to fit around the scope of the function `f`'s invocation say `'x`, as if `bar2` and `foo` were created just before `f` is called and go away immediately after `f` (i.e temporary shortening: assuming variable `bar2` created within `'x` and hence `Bar2<'a>` to `Bar2<'x>`, `'a` being the original (real) lifetime). But here, "shortening" is not possible; One, because of mutable reference to `self` and two, same lifetime on references to `Foo` as well as `Bar2` (`self`), in the function `f`'s definition. Firstly, since it is a mutable reference, it can't convert `Bar2<'a>` to `Bar2<'x>`, because `&mut Bar2<'a>` and `&mut Bar2<'x>` are invariant with each other. (remember even if `T<U` or `T>U`, then `&mut T` is invariant with `&mut U`). So the compiler has to go with `Bar2<'a>` and secondly, since the function `f` is having the same lifetimes for references to `Bar2` and `Foo`, can't convert `&'a Bar2<'a>` to `&'x Bar2<'a>`. So it means the references aren't "shortened" when calling the function `f` and they will remain valid till the end of the block.

If `self`'s lifetime is elided, then the compiler will give a fresh lifetime to the `self` (disjoint with `'b`), which means it is free to "temporarily shorten" the life of `Bar2` and then pass it's `mut` reference to `f`. i.e It will do `&'a mut Bar2<'a>` to `&'x mut Bar2<'a>` and then pass it to `f`. (remember `&'lifetime mut` is covariant over `'lifetime`) and hence it will work.

Share
Improve this answer
Follow
edited Sep 27 '18 at 6:28

answered Sep 8 '18 at 5:52

vikram2784
549 • 5 • 13

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 borrow-checker 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

Declaring lifetime on a test function

1
Accessing mutable reference with getter not working as expected

Related

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

Can you control borrowing a struct vs borrowing a field?

"the immutable borrow prevents mutable borrows" when pumping events with rust-sdl2

Borrowed value does not live long enough compiler error for struct

7
Cannot pass self as callback parameter due to double borrowing

Rust Borrow checker only complains about borrowing as mutable multiple times when a function that returns a reference with the same lifetime assigned

Two mutable borrows happen on the same line?

1
Prevent cannot borrow `*self` as immutable because it is also borrowed as mutable when accessing disjoint fields in struct?

Hot Network Questions

Is Elon Musk really exploiting a loophole to avoid taxes?

`apt-mark showmanual` shows almost all packages. messed up?

What edges are not in a Gabriel graph, yet in a Delauney graph?

Alternatives to replace tandem single pole MWBC breakers?

Does anyone know what this fan blower thingy is?

more hot questions

Question feed