Why can I not return a mutable reference to an outer variable from a closure?

Asked 3 years, 2 months ago

Active 1 year, 9 months ago

Viewed 2k times

11

3

I was playing around with Rust closures when I hit this interesting scenario:

```
fn main() {
    let mut y = 10;

    let f = || &mut y;

    f();
}
```

This gives an error:

```
error[E0495]: cannot infer an appropriate lifetime for borrow expression due to conflicting requirements
 --> src/main.rs:4:16
  |
4 |     let f = || &mut y;
  |                ^^^^^^
  |
note: first, the lifetime cannot outlive the lifetime  as defined on the body at 4:13...
 --> src/main.rs:4:13
  |
4 |     let f = || &mut y;
  |             ^^^^^^^^^
note: ...so that closure can access `y`
 --> src/main.rs:4:16
  |
4 |     let f = || &mut y;
  |                ^^^^^^
note: but, the lifetime must be valid for the call at 6:5...
 --> src/main.rs:6:5
  |
6 |     f();
  |     ^^^
note: ...so type `&mut i32` of expression is valid during the expression
 --> src/main.rs:6:5
  |
6 |     f();
  |     ^^^
```

Even though the compiler is trying to explain it line by line, I still haven't understood what exactly it is complaining about.

Is it trying to say that the mutable reference cannot outlive the enclosing closure?

The compiler does not complain if I remove the call `f()`.

reference rust closures lifetime mutable

Share
Improve this question
Follow

edited Oct 11 '18 at 12:45

Shepmaster
305k • 59 • 824 • 1083

asked Oct 11 '18 at 4:30

soupybionics
3,710 • 4 • 26 • 40

Could this be related to this in some way, as I understand that closures are basically a form of `struct` without a definite type?
– soupybionics
Oct 11 '18 at 5:40

As a workaround, you can take the reference in the outer function and return that from the closure (playground), but I don't know why your original code fails…
– Jmb
Oct 11 '18 at 6:37

Add a comment

3 Answers

Active Oldest Votes

12

Short version

The closure `f` stores a mutable reference to `y`. If it were allowed to return a copy of this reference, you would end up with two simultaneous mutable references to `y` (one in the closure, one returned), which is forbidden by Rust's memory safety rules.

Long version

The closure can be thought of as

```
struct __Closure<'a> {
    y: &'a mut i32,
}
```

Since it contains a mutable reference, the closure is called as `FnMut`, essentially with the definition

```
fn call_mut(&mut self, args: ()) -> &'a mut i32 { self.y }
```

Since we only have a mutable reference to the closure itself, we can't move the field `y` out of the borrowed context, neither are we able to copy it, since mutable references aren't `Copy`.

We can trick the compiler into accepting the code by forcing the closure to be called as `FnOnce` instead of `FnMut`. This code works fine:

```
fn main() {
    let x = String::new();
    let mut y: u32 = 10;
    let f = || {
        drop(x);
        &mut y
    };
    f();
}
```

Since we are consuming `x` inside the scope of the closure and `x` is not `Copy`, the compiler detects that the closure can only be `FnOnce`. Calling an `FnOnce` closure passes the closure itself by value, so we are allowed to move the mutable reference out.

Another more explicit way to force the closure to be `FnOnce` is to pass it to a generic function with a trait bound. This code works fine as well:

```
fn make_fn_once<'a, T, F: FnOnce() -> T>(f: F) -> F {
    f
}

fn main() {
    let mut y: u32 = 10;
    let f = make_fn_once(|| {
        &mut y
    });
    f();
}
```

Share
Improve this answer
Follow
edited Oct 12 '18 at 11:59

answered Oct 11 '18 at 12:01
Sven Marnach
511k ● 113 ● 891 ● 798

Moving-out-of-borrowed-context is more easy to understand than lifetimes-conflict. I hope both these are related to each other and that the lifetimes-conflict stems out from moving-out-of-borrowed-context issue, which I believe is the core issue.
– soupybionics
Oct 12 '18 at 11:20

They are not related. Lifetime relates to memory region validity (a pointer must point to a valid memory slot) whereas moving-out-of-borrowed context relates to memory aliasing (two ipothetical "writable" pointers pointing to the same memory slot: you cannot have two owners in rust). I don't know which of them are more easy to understand, both of them are core concepts to understand and both came into play here.
– attdona
Oct 12 '18 at 12:35

Add a comment

10

There are two main things at play here:

1. Closures cannot return references to their environment
2. A mutable reference to a mutable reference can only use the lifetime of the outer reference (unlike with immutable references)

Closures returning references to environment

Closures cannot return any references with the lifetime of `self` (the closure object). Why is that? Every closure can be called as `FnOnce`, since that's the super-trait of `FnMut` which in turn is the super-trait of `Fn`. `FnOnce` has this method:

```
fn call_once(self, args: Args) -> Self::Output;
```

Note that `self` is passed by value. So since `self` is consumed (and now lives within the `call_once` function`) we cannot return references to it -- that would be equivalent to returning references to a local function variable.

In theory, the `call_mut` would allow to return references to `self` (since it receives `&mut self`). But since `call_once`, `call_mut` and `call` are all implemented with the same body, closures in general cannot return references to `self` (that is: to their captured environment).

Just to be sure: closures can capture references and return those! And they can capture *by* reference and return that reference. Those things are something different. It's just about what is stored in the closure type. If there is a reference stored within the type, it can be returned. But we can't return references to anything stored within the closure type.

Nested mutable references

Consider this function (note that the argument type implies `'inner: 'outer`; `'outer` being shorter than `'inner`):

```
fn foo<'outer, 'inner>(x: &'outer mut &'inner mut i32) -> &'inner mut i32 {
    *x
}
```

This won't compile. On the first glance, it seems like it should compile, since we're just peeling one layer of references. And it does work for immutable references! But mutable references are different here to preserve soundness.

It's OK to return `&'outer mut i32`, though. But it's impossible to get a direct reference with the longer (inner) lifetime.

Manually writing the closure

Let's try to hand code the closure you were trying to write:

```
let mut y = 10;

struct Foo<'a>(&'a mut i32);
impl<'a> Foo<'a> {
    fn call<'s>(&'s mut self) -> &'??? mut i32 { self.0 }
}

let mut f = Foo(&mut y);
f.call();
```

What lifetime should the returned reference have?

- It can't be `'a`, because we basically have a `&'s mut &'a mut i32`. And as discussed above, in such a nested mutable reference situation, we can't extract the longer lifetime!
- But it also can't be `'s` since that would mean the closure returns something with the lifetime of `'self` ("borrowed from `self`"). And as discussed above, closures can't do that.

So the compiler can't generate the closure impls for us.

Share
Improve this answer
Follow
edited Dec 27 '19 at 11:34

answered Oct 11 '18 at 10:04

Lukas Kalbertodt
61.1k ● 18 ● 189 ● 248

1

`Fn` closures *can* return a reference to a captured variable. According to your argument, this should be impossible, yet it works.
– Sven Marnach
Oct 11 '18 at 10:16

@SvenMarnach In your example, `y` is captured by reference. So in the closure's `self` a `&i32` is stored. So the closure isn't returning a reference to self, but captures the environment by reference and returns that. That's what I meant by the last paragraph in the closure part. But I know, it's complicated :/ Try ` move || { &y }` instead and you will get an error. I might try to improve my explanation.
– Lukas Kalbertodt
Oct 11 '18 at 10:19

In the original example, `y` is also captured by (mutable) reference, and the closure isn't returning a reference to `self`. Just remove all `mut` s from your argument, and you get an argument that the case I mentioned should be impossible.
– Sven Marnach
Oct 11 '18 at 10:24

2

I think the actual reason is that a mutable reference isn't `Copy`, while an immutable reference is, and we can't move the mutable reference out of the borrowed context. That's not what the error message says, though.
– Sven Marnach
Oct 11 '18 at 10:26

1

@SvenMarnach Correct, if we use immutable references, it works. The problem only arises due to the interplay of the both things I mention. The nested reference thing (which only applies to mutable references) basically forced the closure to return something with the lifetime of `self`, but since that's not possible, the error occurs.
– Lukas Kalbertodt
Oct 11 '18 at 10:28

Show 7 more comments

9

Consider this code:

```
fn main() {
    let mut y: u32 = 10;

    let ry = &mut y;
    let f = || ry;

    f();
}
```

It works because the compiler is able to infer `ry`'s lifetime: the reference `ry` lives in the same scope of `y`.

Now, the equivalent version of your code:

```
fn main() {
    let mut y: u32 = 10;

    let f = || {
        let ry = &mut y;
        ry
    };

    f();
}
```

Now the compiler assigns to `ry` a lifetime associated to the scope of the closure body, not to the lifetime associated with the main body.

Also note that the immutable reference case works:

```
fn main() {
    let mut y: u32 = 10;

    let f = || {
        let ry = &y;
        ry
    };

    f();
}
```

This is because `&T` has copy semantics and `&mut T` has move semantics, see [Copy/move semantics documentation of &T/&mut T types itself] for more details.

The missing piece

The compiler throws an error related to a lifetime:

```
cannot infer an appropriate lifetime for borrow expression due to conflicting requirements
```

but as pointed out by Sven Marnach there is also a problem related to the error

```
cannot move out of borrowed content
```

But why doesn't the compiler throw this error?

The short answer is that the compiler first executes type checking and then borrow checking.

the long answer

A closure is made up of two pieces:

- the **state** of the closure: a struct containing all the variables captured by the closure
- the **logic** of the closure: an implementation of the `FnOnce`, `FnMut` or `Fn` trait

In this case the state of the closure is the mutable reference `y` and the logic is the body of the closure `{ &mut y }` that simply returns a mutable reference.

When a reference is encountered, Rust controls two aspects:

1. the **state**: if the reference points to a valid memory slice, (i.e. the read-only part of lifetime validity);
2. the **logic**: if the memory slice is aliased, in other words if it is pointed from more than one reference simultaneously;

Note the move out from borrowed content is forbidden for avoiding memory aliasing.

The Rust compiler executes its job through [several stages], here's a simplified workflow:

```
.rs input -> AST -> HIR -> HIR postprocessing -> MIR -> HIR postprocessing -> LLVM IR -> binary
```

The compiler reports a lifetime problem because it first executes the type checking phase in `HIR postprocessing` (which comprises lifetime analysis) and after that, if successful, executes borrow checking in the `MIR postprocessing` phase.

Share
Improve this answer
Follow
edited Mar 17 '20 at 17:43

Catherine Gasnier
93 ● 1 ● 1 ● 5

answered Oct 11 '18 at 9:43

attdona
13.7k ● 6 ● 38 ● 52
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 reference rust closures lifetime mutable 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

4
Why can't I create a closure that produces mutable references to what it closes on?

0
Closure compilation error "borrowed data escapes outside of closure" on referencing environment variable

Copy/move semantics documentation of &T/&mut T types itself

Returning a mutable reference that is behind an immutable reference, passed to the function

0
Is it possible to flatten multiple draining iterators from the same structure?

1
Why this rust FnMut closure code has lifetime errors?

Related

How do I create an array of unboxed functions / closures?

Allowing reference lifetime to outlive a closure

Why is function argument lifetime different to the lifetime of a binding inside a function?

7
Why can Rust not infer the proper lifetime in simple closures, or infers they are conflicting?

What's the real meaning of the error "closure may outlive the current function"?

0
Return and consume an iterator of mutable references from a closure

How to add lifetime argument to closure not returning a reference

Can I coerce a lifetime parameter to a shorter lifetime (soundly) even in the presence of `&mut T`?

Hot Network Questions

Stabilizers in a permutation group

How does a river freeze when the water keeps moving?

Why are nerves blocked even though potassium chanels are not blocked?

'Cloth shop' and 'Clothes shop'