Cannot infer an appropriate lifetime for a closure that returns a reference

Asked 3 years, 8 months ago

Active 1 year, 8 months ago

Viewed 2k times

21

7

Considering the following code:

```
fn foo<'a, T: 'a>(t: T) -> Box<Fn() -> &'a T + 'a> {
    Box::new(move || &t)
}
```

What I expect:

- The type T has lifetime 'a .
- The value t live as long as T .
- t moves to the closure, so the closure live as long as t
- The closure returns a reference to t which was moved to the closure. So the reference is valid as long as the closure exists.
- There is no lifetime problem, the code compiles.

What actually happens:

- The code does not compile:

```
error[E0495]: cannot infer an appropriate lifetime for borrow expression due to conflicting requirements
 --> src/lib.rs:2:22
  |
2 |     Box::new(move || &t)
  |                      ^^
  |
note: first, the lifetime cannot outlive the lifetime  as defined on the body at 2:14...
 --> src/lib.rs:2:14
  |
2 |     Box::new(move || &t)
  |              ^^^^^^^^^^
note: ...so that closure can access `t`
 --> src/lib.rs:2:22
  |
2 |     Box::new(move || &t)
  |                      ^^
note: but, the lifetime must be valid for the lifetime 'a as defined on the function body at 1:8...
 --> src/lib.rs:1:8
  |
1 | fn foo<'a, T: 'a>(t: T) -> Box<Fn() -> &'a T + 'a> {
  |        ^^
  = note: ...so that the expression is assignable:
          expected std::boxed::Box<(dyn std::ops::Fn() -> &'a T + 'a)>
             found std::boxed::Box<dyn std::ops::Fn() -> &T>
```

I do not understand the conflict. How can I fix it?

rust closures ownership borrowing

Share
Improve this question
Follow

edited Jan 18 '19 at 4:10

Shepmaster
305k ● 59 ● 824 ● 1083

asked Apr 12 '18 at 21:45

xardas
345 ● 1 ● 6

Add a comment

4 Answers

Active Oldest Votes

20

Very interesting question! I *think* I understood the problem(s) at play here. Let me try to explain.

***tl;dr*: closures cannot return references to values captured by moving, because that would be a reference to `self`. Such a reference cannot be returned because the `Fn*` traits don't allow us to express that.** This is basically the same as the *streaming iterator problem* and could be fixed via GATs (generic associated types).

Implementing it manually

As you probably know, when you write a closure, the compiler will generate a struct and `impl` blocks for the appropriate `Fn` traits, so closures are basically syntax sugar. Let's try to avoid all that sugar and build your type manually.

What you want is a type which *owns* another type and can return references to that owned type. And you want to have a function which returns a boxed instance of said type.

```
struct Baz<T>(T);

impl<T> Baz<T> {
    fn call(&self) -> &T {
        &self.0
    }
}

fn make_baz<T>(t: T) -> Box<Baz<T>> {
    Box::new(Baz(t))
}
```

This is pretty equivalent to your boxed closure. Let's try to use it:

```
let outside = {
    let s = "hi".to_string();
    let baz = make_baz(s);
    println!("{}", baz.call()); // works

    baz
};

println!("{}", outside.call()); // works too
```

This works just fine. The string `s` is moved into the `Baz` type and that `Baz` instance is moved into the `Box`. `s` is now owned by `baz` and then by `outside`.

It gets more interesting when we add a single character:

```
let outside = {
    let s = "hi".to_string();
    let baz = make_baz(&s);  // <-- NOW BORROWED!
    println!("{}", baz.call()); // works

    baz
};

println!("{}", outside.call()); // doesn't work!
```

Now we cannot make the lifetime of `baz` bigger than the lifetime of `s`, since `baz` contains a reference to `s` which would be an dangling reference of `s` would go out of scope earlier than `baz`.

The point I wanted to make with this snippet: we didn't need to annotate any lifetimes on the type `Baz` to make this safe; Rust figured it out on its own and enforces that `baz` lives no longer than `s`. This will be important below.

Writing a trait for it

So far we only covered the basics. Let's try to write a trait like `Fn` to get closer to your original problem:

```
trait MyFn {
    type Output;
    fn call(&self) -> Self::Output;
}
```

In our trait, there are no function parameters, but otherwise it's fairly identical to the real `Fn` trait.

Let's implement it!

```
impl<T> MyFn for Baz<T> {
    type Output = ???;
    fn call(&self) -> Self::Output {
        &self.0
    }
}
```

Now we have a problem: what do we write instead of `???`? Naively one would write `&T`... but we need a lifetime parameter for that reference. Where do we get one? What lifetime does the return value even have?

Let's check the function we implemented before:

```
impl<T> Baz<T> {
    fn call(&self) -> &T {
        &self.0
    }
}
```

So here we use `&T` without lifetime parameter too. But this only works because of lifetime elision. Basically, the compiler fills in the blanks so that `fn call(&self) -> &T` is equivalent to:

```
fn call<'s>(&'s self) -> &'s T
```

Aha, so the lifetime of the returned reference is bound to the `self` lifetime! (more experienced Rust users might already have a feeling where this is going...).

(As a side note: why is the returned reference not dependent on the lifetime of `T` itself? If `T` references something non-`'static` then this has to be accounted for, right? Yes, but it is already accounted for! Remember that no instance of `Baz<T>` can ever live longer than the thing `T` might reference. So the `self` lifetime is already shorter than whatever lifetime `T` might have. Thus we only need to concentrate on the `self` lifetime)

But how do we express that in the trait impl? Turns out: **we can't** (yet). This problem is regularly mentioned in the context of *streaming iterators* -- that is, iterators that return an item with a lifetime bound to the `self` lifetime. In today's Rust, it is sadly impossible to implement this; the type system is not strong enough.

What about the future?

Luckily, there is an RFC "Generic Associated Types" which was merged some time ago. This RFC extends the Rust type system to allow associated types of traits to be generic (over other types and lifetimes).

Let's see how we can make your example (kinda) work with GATs (according to the RFC; this stuff doesn't work yet ☹). First we have to change the trait definition:

```
trait MyFn {
    type Output<'a>;   // <-- we added <'a> to make it generic
    fn call(&self) -> Self::Output;
}
```

The function signature hasn't changed in the code, but notice that lifetime elision kicks in! The above `fn call(&self) -> Self::Output` is equivalent to:

```
fn call<'s>(&'s self) -> Self::Output<'s>
```

So the lifetime of the associated type is bound to the `self` lifetime. Just as we wanted! The `impl` looks like this:

```
impl<T> MyFn for Baz<T> {
    type Output<'a> = &'a T;
    fn call(&self) -> Self::Output {
        &self.0
    }
}
```

To return a boxed `MyFn` we would need to write this (according to this section of the RFC:

```
fn make_baz<T>(t: T) -> Box<for<'a> MyFn<Output<'a> = &'a T>> {
    Box::new(Baz(t))
}
```

And what if we want to use the *real* `Fn` trait? As far as I understand, we can't, even with GATs. I think it's impossible to change the existing `Fn` trait to use GATs in a backwards compatible manner. So it's likely that the standard library will keep the less powerful trait as is. (side note: how to evolve the standard library in backwards incompatible ways to use new language features is something I wondered about a few times already; so far I haven't heard of any real plan in this regards; I hope the Rust team comes up with something...)

Summary

What you want is not technically impossible or unsafe (we implemented it as a simple struct and it works). However, unfortunately it is impossible to express what you want in the form of closures/ `Fn` traits in Rust's type system right now. This is the same problem *streaming iterators* are dealing with.

With the planned GAT feature, it is possible to express all of this in the type system. However, the standard library would need to catch up somehow to make your exact code possible.

Share
Improve this answer
Follow

edited Apr 14 '20 at 23:58

trent formerly cl
20.1k ● 7 ● 42 ● 72

answered Apr 13 '18 at 9:05

Lukas Kalbertodt
61.1k ● 18 ● 189 ● 248

Add a comment

▲
9
▼

What I expect:

- The type `T` has lifetime `'a`.
- The value `t` live as long as `T`.

This makes no sense. A value cannot "live as long" as a type, because a type doesn't live. "`T` has lifetime `'a`" is a very imprecise statement, easy to misunderstand. What `T: 'a` really means is "instances of `T` must stay valid at least as long as lifetime `'a`. For example, T must not be a reference with a lifetime shorter than `'a`, or a struct containing such a reference. Note that this has nothing to do with forming references *to* `T`, i.e. `&T`.

The value `t`, then, lives as long as its lexical scope (it's a function parameter) says it does, which has nothing to do with `'a` at all.

- `t` moves to the closure, so the closure live as long as `t`

This is also incorrect. The closure lives as long as the closure does lexically. It is a temporary in the result expression, and therefore lives until the end of the result expression. `t`'s lifetime concerns the closure not at all, since it has its own `T` variable inside, the capture of `t`. Since the capture is a copy/move of `t`, it is not in any way affected by `t`'s lifetime.

The temporary closure is then moved into the box's storage, but that's a new object with its own lifetime. The lifetime of *that* closure is bound to the lifetime of the box, i.e. it is the return value of the function, and later (if you store the box outside the function) the lifetime of whatever variable you store the box in.

All of that means that a closure that returns a reference to its own capture state must bind the lifetime of that reference to its own reference. Unfortunately, **this is not possible**.

Here's why:

The `Fn` trait implies the `FnMut` trait, which in turn implies the `FnOnce` trait. That is, *every* function object in Rust can be called with a by-value `self` argument. This means that every function object must be still valid being called with a by-value `self` argument and returning the same thing as always.

In other words, trying to write a closure that returns a reference to its own captures expands to roughly this code:

```
struct Closure<T> {
    captured: T,
}
impl<T> FnOnce<()> for Closure<T> {
    type Output = &'??? T; // what do I put as lifetime here?
    fn call_once(self, _: ()) -> Self::Output {
        &self.captured // returning reference to local variable
                       // no matter what, the reference would be invalid once we return
    }
}
```

And this is why what you're trying to do is fundamentally impossible. Take a step back, think of what you're actually trying to accomplish with this closure, and find some other way to accomplish it.

Share
Improve this answer
Follow

edited Apr 13 '18 at 18:31

Shepmaster
305k ● 59 ● 824 ● 1083

answered Apr 13 '18 at 8:45

Sebastian Redl
63.4k ● 8 ● 110 ● 144

"*every function object in Rust can be called with a by-value `self` argument*" -- very nice observation! I totally missed that.
– Lukas Kalbertodt
Apr 13 '18 at 9:17

Add a comment

1

You expect the type `T` to have lifetime `'a`, but `t` is not a reference to a value of type `T`. The function takes ownership of the variable `t` by argument passing:

```
// t is moved here, t lifetime is the scope of the function
fn foo<'a, T: 'a>(t: T)
```

You should do:

```
fn foo<'a, T: 'a>(t: &'a T) -> Box<Fn() -> &'a T + 'a> {
    Box::new(move || t)
}
```

Share
Improve this answer
Follow

edited Apr 13 '18 at 18:28

Shepmaster
305k ● 59 ● 824 ● 1083

answered Apr 13 '18 at 7:40

SK.Chen
83 ● 1 ● 6

I think OP wanted the closure to own `T`. In your code the closure doesn't own `T`, but merely a reference to `T`. So the semantics of handling the returned `Box` are vastly different.
– Lukas Kalbertodt
Apr 13 '18 at 9:07

Add a comment

1

The other answers are top-notch, but I wanted to chime in with another reason your original code couldn't work. A big problem lies in the signature:

```
fn foo<'a, T: 'a>(t: T) -> Box<Fn() -> &'a T + 'a>
```

This says that the *caller* may specify *any* lifetime when calling `foo` and the code will be valid and memory-safe. That cannot possibly be true for this code. It wouldn't make sense to call this with `'a` set to `'static`, but nothing about this signature would prevent that.

Share
Improve this answer
Follow

answered Apr 13 '18 at 18:32

Shepmaster
305k ● 59 ● 824 ● 1083

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 closures ownership borrowing 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

2
Can a closure return a reference to data it owns?

How do I write an iterator that returns references to itself?

How to express lifetime restrictions of a closure to match a trait bounded lifetime?

Related

Storing a closure in a structure — cannot infer an appropriate lifetime

Unable to infer lifetime for borrow expression when using a trait with an explicit lifetime

Allowing reference lifetime to outlive a closure

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

How to set lifetime for boxed closure capturing `self`?

Cannot infer an appropriate lifetime for lifetime parameter cloning trait object

Lifetime of reference to boxed value does not live long enough

Why is it legal to borrow a temporary?

How to add lifetime argument to closure not returning a reference

Does move on a closure copy the reference "pointer" or the actual object referenced?

Hot Network Questions

Changing a color of a link

Remove the Times x from display in Inactivate expressions in V13

Sample without replacement from 1 to N and stop when the value is less than the previous one

Is it common practice to apply identical processing effects to a batch of photos?

Would I be able to avoid the wash sale rule if I buy back the security on January 1st after selling it on December 31st?

more hot questions

Question feed

STACK OVERFLOW

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