value "does not live long enough", but only when using a function pointer

Asked 3 years, 11 months ago
Active 3 years, 11 months ago
Viewed 204 times

1

I'm trying to get the following simplified code to compile:

```
type FunctionType<'input> = fn(input_string: &'input str) -> Result<(), &'input str>;

fn okay<'input>(_input_string: &'input str) -> Result<(), &'input str> {
    Ok(())
}

fn do_stuff_with_function(function: FunctionType) {
    let string = String::new();
    match function(string.as_str()) {
        Ok(_) => {},
        Err(_) => {},
    }

}

fn main() {
    do_stuff_with_function(okay);
}
```

The playground complains:

```
error[E0597]: `string` does not live long enough
  --> src/main.rs:13:20
   |
13 |     match function(string.as_str()) {
   |                    ^^^^^^ does not live long enough
...
18 | }
   | - borrowed value only lives until here
   |
note: borrowed value must be valid for the anonymous lifetime #1 defined on the function body at 11:1...
  --> src/main.rs:11:1
   |
11 | / fn do_stuff_with_function(function: FunctionType) {
12 | |     let string = String::new();
13 | |     match function(string.as_str()) {
14 | |         Ok(_) => {},
... |
17 | |     }
18 | | }
   | |_^
```

I understand why the error would be firing under other circumstances: `string` only lives as long as the execution of `do_stuff_with_function`, but `do_stuff_with_function` returns the value of `function`'s invocation which includes a same-lifetime reference to its input value, i.e., `string`.

However, I'm confused on three points:

1. I `match` the result of the function call, then return `()` for both branches. Why does the lifetime of value returned by `function` matter if it's unconditionally thrown out?
2. If I replace the call to the parameter `function` with a direct reference to `okay` (which has an identical signature), it compiles without complaint.
3. The error message suggests (though does not state outright?) that the necessary lifetime is already the same as the actual lifetime.

rust

Share
Improve this question
Follow

asked Dec 30 '17 at 23:52

stuffy
313 ● 4 ● 13

Add a comment

1 Answer

Active Oldest Votes

2

TL;DR: Use the `for<'a>` syntax to have a specific lifetime for the function, rather than take one from the context where it's used.

Lifetime elision on args of `do_stuff_with_function` hides what's going on here. Your actual lifetime is:

```
fn do_stuff_with_function<'a>(function: FunctionType<'a>) {
```

This lifetime declared in function argument doesn't mean "some random short lifetime I'll come up with later", but rather "a lifetime that already exists at the point of calling this function".

These lifetime annotations are not wildcards, but more like identifiers to track where values came from and where they are going to. They can be used, for example, to clarify a situation like this:

```
fn do_stuff_with_function<'a>(function: FunctionType<'a>, t: &'a str) {
    match function(t) {
        Ok(_) => {},
        Err(_) => {},
    }
}

fn main() {
    let string = String::new();
    do_stuff_with_function(okay, string.as_str());
}
```

However, the problem is solvable in Rust, but just needs a more advanced syntax. For the purpose of the explanation, first, let's change it to:

```
fn do_stuff_with_function<'a, F>(function: F) where F: Fn(&'a str) -> Result<(), &'a str> {
```

That means "make a copy of `do_stuff_with_function` for every unique `F` which can be anything that looks like a function taking `&'a str` (etc.). This is essentially the same (+ allows closures) as your code. However, I still had to name the lifetime as tied to the call of `do_stuff_with_function<'a>`.

So here's a freaky type magic:

```
fn do_stuff_with_function<F>(function: F) where F: for<'a> Fn(&'a str) -> Result<(), &'a str> {
```

Which allows moving definition of the lifetime from `do_stuff_with_function` to definition of the `Fn` using `for<'a>` syntax. This way it's specific to the `F`, rather than `do_stuff_with_function` arguments.

Share
Improve this answer
Follow
edited Dec 31 '17 at 15:57

answered Dec 31 '17 at 0:30

Kornel
93.1k ● 32 ● 207 ● 290

This works for my case! I'm still unsure of why point (1) in my original question is not sufficient to please the compiler and am sad that you have to inline the alias (the real one I have is rather huge), but glad this is possible, even with freaky type magic.
– stuffy
Dec 31 '17 at 1:28

2

@stuffy, you don't need to inline the alias. `for<'_>` can be added to it. Playground
– red75prime
Dec 31 '17 at 3:30

2

You can also just remove `<'input>` from type alias. `type FunctionType = fn(input_string: &str) -> Result<(), &str>;` Lifetime elision will do the right thing.
– red75prime
Dec 31 '17 at 4:15

@red75prime I may have simplified too much: I thought I had to inline it because my function type alias is generic, as it would not compile following the pattern of this answer. Instead, it looks like I can't combine generic `FunctionType<T>` with generic `do_stuff_with_function<T, F>(...) where F: FunctionType<T>` and have to `do_stuff_with_function<T>(function: FunctionType<T>)`. Also, elision doesn't work here for me because I actually have multiple arguments. :) Playground. `for<'_>` is indeed the key.
– stuffy
Dec 31 '17 at 17:35

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

Related

"borrowed value does not live long enough" when using the builder pattern

Factory method: instance does not live long enough

Borrow data out of a mutex "borrowed value does not live long enough"

borrowed value does not live long enough in this case (Vec<&Fn(i32) -> i32>)

Borrowed value does not live long enough - string slice into HashMap

Values does not live long enough in constructor and setter in OOP Rust

2
borrowed value does not live long enough when use generic lifecycle。

`*arg0` does not live long enough - wasm_bindgen

nom & borrowed value does not live long enough error

Trying to save reference of closure, but not living long enough

Hot Network Questions

How can I let my opponent know that I'm touching a piece to adjust it?

What's the social meaning of "He was a student of..."?

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

Why is my reasoning incorrect - probability?

Why does the prophecy imply Macbeth has to murder the king?

more hot questions

Question feed