Confused by Rust closure lifetime

Asked 2 years, 1 month ago
Active 2 years, 1 month ago
Viewed 267 times

1

1

I ran into a confusing situation where what the compiler outputs doesn't logically make sense. Here is the minimal example to reproduce the same issue I'm having with my project code.

```
use std::sync::Arc;

struct A<'a, T> {
    f: Box<dyn Fn(&u32) -> T + 'a>
}

struct B<'a> {
    inner: A<'a, Z<'a>>
}

impl<'a, T> A<'a, T> {
    fn new<F>(f: F) -> Self where F: Fn(&u32) -> T + 'a {
        A { f: Box::new(f) }
    }
}

struct X<'a> {
    _d: &'a std::marker::PhantomData<()>
}

struct Z<'a> {
    _d: &'a std::marker::PhantomData<()>
}

impl<'a> X<'a> {
    fn g(&self, y: u32) -> Z {
        Z { _d: &std::marker::PhantomData }
    }
}

impl<'a> B<'a> {
    fn new(x: Arc<X<'a>>) -> Self {
        B {
            inner: A::new(move |y: &u32| -> Z {
                x.g(*y)
```

And the confusing compilation error:

```
error[E0495]: cannot infer an appropriate lifetime for lifetime parameter in function call due to conflicting requirements
  --> t.rs:35:19
   |
35 |                 x.g(*y)
   |                 ^
   |
note: first, the lifetime cannot outlive the lifetime '_ as defined on the body at 34:27...
  --> t.rs:34:27
   |
34 |             inner: A::new(move |y: &u32| -> Z {
   |                            ^^^^^^^^^^^^^^^^^^^^
note: ...so that closure can access `x`
  --> t.rs:35:17
   |
35 |                 x.g(*y)
   |                 ^
note: but, the lifetime must be valid for the lifetime 'a as defined on the impl at 31:6...
  --> t.rs:31:6
   |
31 | impl<'a> B<'a> {
   |      ^^
   = note: ...so that the expression is assignable:
           expected B<'a>
             found B<'_>

error: aborting due to previous error
```

What I don't quite get is what "the lifetime" refers to as mentioned in the log, and what exactly that anonymous lifetime `'_` stands for.

rust closures lifetime

Share
Improve this question
Follow

asked Nov 17 '19 at 5:18

Determinant

3,476 5 29 45

Add a comment

1 Answer

Active Oldest Votes

2

There was a small oversight in your code. It should be:

```
impl<'a> X<'a> {
  fn g(&self, y: u32) -> Z<'a> {
    Z { _d: &std::marker::PhantomData }
  }
}
```

The whole thing then compiles. This is an example of Rust's lifetime elision rule coming into play.

According to the relevant elision rules, namely:

- Each elided lifetime in input position becomes a distinct lifetime parameter.
- If there are multiple input lifetime positions, but one of them is `&self` or `&mut self`, the lifetime of `self` is assigned to all elided output lifetimes.

Then to `rustc` your original code will actually be like following:

```
impl<'a> X<'a> {
  fn g<'b>(&'b self, y: u32) -> Z<'b> {
    Z { _d: &std::marker::PhantomData }
  }
}
```

The elided lifetime parameter `'b` will come from the calling site, which is exactly what you saw in the error message. The `rustc` couldn't reconcile the two lifetimes, hence the error.

Share
Improve this answer
Follow
answered Nov 17 '19 at 6:31

edwardw
10.3k 3 29 43

Thanks for asking so promptly! So I do know about the elision rule... and actually I wrote 'b explicitly previously. But what I didn't quite get, intuitively, is why rustc couldn't reconcile the two lifetimes. Could you point that out?
– Determinant
Nov 17 '19 at 6:45

1

@Determinant a flaw in your reasoning, *'b should be bounded by the lifetime of x, which is bounded by Arc<X<'a>>*. Not true, remember a closure is a suspended computing, which doesn't happen yet. `'b` is only materialized whenever `B::inner::f` is called. `rustc` can't possibly know what that is. So unless it can statically determines all lifetimes involved, it will complain. `fn g(&self, y: u32) -> Z<'a>` does just that, which was the only place in your original code that depended on elided lifetime.
– edwardw
Nov 17 '19 at 8:27

1

Minor correction: relationships between lifetimes are a different thing from variance, which is a relationship between a type constructor and its parameter. I don't know if there's a good name for the relationships between lifetimes but you could say that `'b` and `'a` are *independent* unless explicitly constrained. (Actually, because `'b` is already constrained not to strictly outlive `'a`, they are only partially independent; adding `'b: 'a` would constrain them to be the same.)
– trent formerly cl
Nov 17 '19 at 13:52

1

`'b: 'a` is a mistake because it says that `'b` must outlive `'a`. `'a` must already outlive `'b`, so adding the annotation just says that `'a` and `'b` must be the exact *same* lifetime -- in other words, it's equivalent to `fn g(&'a self, y: u32) -> Z<'a>`. Writing `&'a self` is nearly always a mistake because it forces `self` to be borrowed for `'a`, and when `'a` is the lifetime of something inside `self`, the lifetimes can become overconstrained. This recent question is about a bug in a library caused by just such a misplaced `'a`.
– trent formerly cl
Nov 17 '19 at 22:59

1

Applied to your problem, you can't make `new` compile by *adding* a constraint to `g` (forcing `'b` to outlive / be the same as `'a`), because the problem is that the constraints are contradictory. You need to *remove* a constraint (allow `'b` to be independent of `'a`).
– trent formerly cl
Nov 17 '19 at 23:01

Show **5** more comments

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 lifetime 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

1
cannot borrow `xxx` as mutable more than once at a time when using mpd's Query::and

Related

What is the difference between a 'closure' and a 'lambda'?

JavaScript closure inside loops – simple practical example

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

Allowing reference lifetime to outlive a closure

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

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

"the type does not fulfill the required lifetime" when using a method in a thread

Storing a Vector of structs containing closures in Rust

Awaiting a Number of Futures Unknown at Compile Time

Hot Network Questions

Is there any other notation, like tab notation, for piano?

Schrödinger's cat program

Is Elon Musk really exploiting a loophole to avoid taxes?

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

What's the largest REG_SZ value that Regedit can edit?

more hot questions

Question feed