How do I comply with lifetime bounds when passing functions as argument?

Asked 1 year, 7 months ago
Active 1 year, 7 months ago
Viewed 130 times

0

Original Issue

This piece of code is relatively similar to the piece of code I am trying to fix. I've also asked this on the Rust user's forum.

playground

```
/// assume this function can't be modified.
fn foo<A>(
    f1: impl Fn(&str) -> Result<(&str, A), ()>,
    base: &str,
    f2: impl Fn(A) -> bool
) {
    let s: String = base.to_owned();
    let option = Some(s.as_ref());
    let mapped = option.map(f1);
    let r = mapped.unwrap();
    let (rem, prod) = r.unwrap();
    assert!(f2(prod));
    assert_eq!(rem.len(), 0);
}


fn main() {
    fn bar<'a>(s: &'a str) -> Result<(&'a str, &'a str), ()> {
        Ok((&s[..1], &s[..]))
    }


    fn baz(s: &str) -> Result<(&str, &str), ()> {
        Ok((&s[..1], &s[..]))
    }

    foo(bar, "string", |s| s.len() == 5); // fails to compile

    foo(baz, "string", |s| s.len() == 5); // fails to compile
}


error[E0271]: type mismatch resolving `for<'r> <for<'a> fn(&'a str) -> std::result::Result<(&'a str, &'a str), ()> {main::bar} as std::ops::FnOnce<(&'r str,)>>::Output == std::result::Result<(&'r str, _), ()>`
  --> src/main.rs:27:5
   |
2  | fn foo<A>(
   |    ---
3  |     f1: impl Fn(&str) -> Result<(&str, A), ()>,
   |                          ---------------------- required by this bound in `foo`
...
27 |     foo(bar, "string", |s| s.len() == 5); // fails to compile
   |     ^^^ expected bound lifetime parameter, found concrete lifetime
```

Edit:

Based on recommendations from a number of people here, on the internals thread I made, and on the rust user forum, I changed my code to simplify it by using a wrapper trait.

playground

```
trait Parser<'s> {
    type Output;

    fn call(&self, input: &'s str) -> (&'s str, Self::Output);
}

impl<'s, F, T> Parser<'s> for F
where F: Fn(&'s str) -> (&'s str, T) {
    type Output = T;
    fn call(&self, input: &'s str) -> (&'s str, T) {
        self(input)
    }
}

fn foo<F1, F2>(
    f1: F1,
    base: &'static str,
    f2: F2
)
where
    F1: for<'a> Parser<'a>,
    F2: FnOnce(&<F1 as Parser>::Output) -> bool
{
    // These two lines cannot be changed.
    let s: String = base.to_owned();
    let str_ref = s.as_ref();

    let (remaining, produced) = f1.call(str_ref);
    assert!(f2(&produced));
    assert_eq!(remaining.len(), 0);
}

struct Wrapper<'a>(&'a str);
```

this code generates an internal compiler error currently:

```
error: internal compiler error: src/librustc_infer/traits/codegen/mod.rs:61: Encountered error `OutputTypeParameterMismatch(Binder(<[closure@src/main.rs:45:24: 45:40] as std::ops::FnOnce<(&<fo

thread 'rustc' panicked at 'Box<Any>', src/librustc_errors/lib.rs:875:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

note: the compiler unexpectedly panicked. this is a bug.

note: we would appreciate a bug report: https://github.com/rust-lang/rust/blob/master/CONTRIBUTING.md#bug-reports

note: rustc 1.43.0 (4fb7144ed 2020-04-20) running on x86_64-unknown-linux-gnu

note: compiler flags: -C codegen-units=1 -C debuginfo=2 --crate-type bin

note: some of the compiler flags provided by cargo are hidden

error: aborting due to previous error

error: could not compile `playground`.

To learn more, run the command again with --verbose.
```

I have made a bug report here.

generics rust lifetime borrowing

Share
Improve this question
Follow

edited May 6 '20 at 20:06

asked May 6 '20 at 5:15

Antonia Calia-Bogan
19 ● 4

Can you describe in more detail what the expected behaviour and the problem is? It's great that you've included a playground link, but we shouldn't have to run the playground to guess what you're asking about.
– Raniz
May 6 '20 at 5:26

1

The `Fn` trait doesn't seem to be flexible enough to express this, so you should define your own trait instead.
– Sven Marnach
May 6 '20 at 9:47

I just saw the comment "assume this function can't be modified" on the playground. In that case, the function simply can't take the function you want to pass in. `A` needs to have static lifetime, so make it some kind of "owned" type (in this case a `String`).
– Sven Marnach
May 6 '20 at 9:51

@PeterHall Ok well then assuming that function could be modified, what modifications would you make?
– Antonia Calia-Bogan
May 6 '20 at 17:40

@AntoniaCalia-Bogan one suggestion is in my answer below. But we're talking about a generic function here, so it really depends on your anticipated bounds on the genericity.

– Peter Hall
May 6 '20 at 17:47

Show 1 more comment

2 Answers

Active Oldest Votes

2

Look at the first function argument:

```
f1: impl Fn(&str) -> Result<(&str, A), ()>,
```

Where could a value of type `A` come from? It has to be either:

- derived from the `str` in the argument, or
- plucked from nowhere, which would mean it is `'static`

But `A` is declared for `foo`, not for the specific `f1` argument. This means that the lifetime of `A` cannot depend on the argument of `f1`. But that is exactly what `bar` and `baz` do.

So what can you do? Given your requirement of "assume this function can't be modified", you are stuck with changing `bar` and `baz` so that the type of `A` is static. This gives you the choice of a newly allocated `String` or a `&'static str`:

```
fn bar<'a>(s: &'a str) -> Result<(&'a str, String), ()> {
    Ok((&s[..1], s[..].to_owned()))
}
```

Or:

```
fn bar<'a>(s: &'a str) -> Result<(&'a str, &'static str), ()> {
    Ok((&s[..1], "hello"))
}
```

If you *were* able to change the type signature of `foo`, you could use references to `A` in the argument functions' signatures, which would let you describe their lifetimes in relation to their other arguments:

E.g.:

```
fn foo<A: ?Sized>(
    f1: impl Fn(&str) -> Result<(&str, &A), ()>,
    base: &str,
    f2: impl Fn(&A) -> bool
) {
    unimplemented!()
}
```

Which is equivalent to the following, without lifetime elision:

```
fn foo<A: ?Sized>(
    f1: impl for<'a> Fn(&'a str) -> Result<(&'a str, &'a A), ()>,
    base: &str,
    f2: impl for<'a> Fn(&'a A) -> bool
) {
    unimplemented!()
}
```

Note that the type signature of `f1` now expresses an association between the lifetime of the input `&str` and the `&A` in the result.

Share
Improve this answer
Follow
edited May 6 '20 at 11:37

answered May 6 '20 at 11:26

Peter Hall
41.4k ● 11 ● 90 ● 157

Add a comment

0

For anyone who reads through this later on and wonders if I ever found a workaround, I did!

This workaround involves placing the state referenced by the function calls into a `Context` structure. Because of the current issues with Higher Rank Trait Bounds (HRTBs), this workaround avoids them entirely. Instead, we refactor object initialization (which could be expensive) into the Context's constructor. The context completely owns the function state. This is fine though, since the function only needs a reference to that state, not ownership. Whenever we need to call the function, we pass it to the Context's call function, which ensures that the lifetime of the arguments to the function and it's output matches the lifetime of the context in/on which it runs.

playground

```
// Workaround code: the state of the function is placed
// into a struct so that all of the references are valid
// for the lifetime of of &self. This way concrete lifetimes
// can be used because all lifetimes start on the function
// call, rather than several statements into the function.
//
// This work around eliminates the need for Higher Ranked
// Trait Bounds (HRTBs), since all lifetimes are instantiated
// on function initialization.

struct Wrapper<'a>(&'a str);

struct ParserContext {
    inner: String
}

impl ParserContext {
    fn new(base: &str) -> Self {Self {inner: base.to_owned()}}

    fn call<'a, O>(
        &'a self,
        f1: fn(&'a str) -> (&'a str, O),
        f2: fn(O) -> bool,
    ) {
        let (remaining, produced) = f1(self.inner.as_str());
        assert_eq!(remaining.len(), 0);
        assert!(f2(produced));
    }
}


fn main() {
    fn bar(s: &str) -> (&str, &str) {
        (&s[..0], &s[..])
    }
```

Share
Improve this answer
Follow

answered May 7 '20 at 1:30

Antonia Calia-Bogan
19 ● 4

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 generics rust lifetime 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

Visit chat

Related

Lifetime error using associated type of trait with lifetime parameter

Lifetime issues with a closure argument in Rust

Problems with Tuple's lifetime in rust.

Store data that implements a trait in a vector

1

Rust: how to use await in function chain

Problems with lifetime/borrow on str type

2

Lifetimes implementing AsRef

Generic type parameter is not constrained when passing lifetime

Awaiting a Number of Futures Unknown at Compile Time

why rustc compile complain my simple code "the trait std::io::Read is not implemented for Result<File, anyhow::Error>"

Hot Network Questions

- How to install a package via `apt-get` without flagging it as manually installed
- If we can get people to the moon and back, why are we so adamant that it's impossible to service James Webb at 4x that with a one way robotic vehicle?
- Question on OEIS A000085
- Who (or what) created the atropal?
- How to salvage bitter homemade mustard?

more hot questions

Question feed