Products

How can this instance seemingly outlive its own parameter lifetime?

Log in Sign up

Ask Question

Asked 4 years, 9 months ago

Active 4 years, 9 months ago

Viewed 3k times

32

18

Before I stumbled upon the code below, I was convinced that a lifetime in a type's lifetime parameter would always outlive its own instances. In other words, given a `foo: Foo<'a>`, then `'a` would always outlive `foo`. Then I was introduced to this counter-argument code by @Luc Danton (Playground):

```
#[derive(Debug)]
struct Foo<'a>(std::marker::PhantomData<fn(&'a ())>);

fn hint<'a, Arg>(_: &'a Arg) -> Foo<'a> {
    Foo(std::marker::PhantomData)
}

fn check<'a>(_: &Foo<'a>, _: &'a ()) {}

fn main() {
    let outlived = ();
    let foo;

    {
        let shortlived = ();
        foo = hint(&shortlived);
        // error: `shortlived` does not live long enough
        //check(&foo, &shortlived);
    }

    check(&foo, &outlived);
}
```

Even though the `foo` created by `hint` appears to consider a lifetime that does not live for as long as itself, and a reference to it is passed to a function in a wider scope, the code compiles exactly as it is. Uncommenting the line stated in the code triggers a compilation error. Alternatively, changing `Foo` to the struct tuple `(PhantomData<&'a ()>)` also makes the code no longer compile with the same kind of error (Playground).

How is it valid Rust code? What is the reasoning of the compiler here?

rust lifetime

Share
Improve this question
Follow

asked Mar 7 '17 at 0:10

E_net4 the flagger
22.8k ● 11 ● 80 ● 117

1

Wow, that's strange ! Looking at the MIR for both hint functions, it looks like rust drop the 'a lifetime when using PhantomData<fn(&'a ())>. IDK if it's a feature or a bug :D
– Grégory OBANOS
Mar 7 '17 at 0:34

I suspect the answer has to do with variance, specifically the offhand comment that `fn(T)` is *contravariant* in `T` -- however, I'm not quite up to the task of explaining why.
– trent formerly cl
Mar 7 '17 at 2:19

Add a comment

2 Answers

Active Oldest Votes

70

+200

Despite your best intentions, your `hint` function may not have the effect you expect. But we have quite a bit of ground to cover before we can understand what's going on.

Let's begin with this:

```
fn ensure_equal<'z>(a: &'z (), b: &'z ()) {}

fn main() {
    let a = ();
    let b = ();
    ensure_equal(&a, &b);
}
```

OK, so in `main`, we defining two variables, `a` and `b`. They have distinct lifetimes, by virtue of being introduced by distinct `let` statements. `ensure_equal` requires two references with *the same lifetime*. And yet, this code compiles. Why?

That's because, given `'a: 'b` (read: `'a` outlives `'b`), `&'a T` is a [*subtype*]() of `&'b T`.

Let's say the lifetime of `a` is `'a` and the lifetime of `b` is `'b`. It's a fact that `'a: 'b`, because `a` is introduced first. On the call to `ensure_equal`, the arguments are typed `&'a ()` and `&'b ()`, respectively[1]. There's a type mismatch here, because `'a` and `'b` are not the same lifetime. But the compiler doesn't give up yet! It knows that `&'a ()` is a subtype of `&'b ()`. In other words, a `&'a ()` *is a* `&'b ()`. The compiler will therefore coerce the expression `&a` to type `&'b ()`, so that both arguments are typed `&'b ()`. This resolves the type mismatch.

If you're confused by the application of "subtypes" with lifetimes, then let me rephrase this example in Java terms. Let's replace `&'a ()` with `Programmer` and `&'b ()` with `Person`. Now let's say that `Programmer` is derived from `Person`: `Programmer` is therefore a subtype of `Person`. That means that we can take a variable of type `Programmer` and pass it as an argument to a function that expects a parameter of type `Person`. That's why the following code will successfully compile: the compiler will resolve `T` as `Person` for the call in `main`.

```
class Person {}
class Programmer extends Person {}

class Main {
    private static <T> void ensureSameType(T a, T b) {}

    public static void main(String[] args) {
        Programmer a = null;
        Person b = null;
        ensureSameType(a, b);
    }
}
```

Perhaps the non-intuitive aspect of this subtyping relation is that the longer lifetime is a subtype of the shorter lifetime. But think of it this way: in Java, it's safe to pretend that a `Programmer` is a `Person`, but you can't assume that a `Person` is a `Programmer`. Likewise, it's safe to pretend that a variable has a *shorter* lifetime, but you can't assume that a variable with some known lifetime actually has a *longer* lifetime. After all, the whole point of lifetimes in Rust is to ensure that you don't access objects beyond their actual lifetime.

Now, let's talk about [variance](). What's that?

> Variance is a property that type constructors have with respect to their arguments. A type constructor in Rust is a generic type with unbound arguments. For instance `Vec` is a type constructor that takes a `T` and returns a `Vec<T>`. `&` and `&mut` are type constructors that take two inputs: a lifetime, and a type to point to.

Normally, you would expect all elements of a `Vec<T>` to have the same type (and we're not talking about trait objects here). But variance lets us cheat with that.

`&'a T` is *covariant* over `'a` and `T`. That means that wherever we see `&'a T` in a type argument, we can substitute it with a subtype of `&'a T`. Let's see how it works out:

```
fn main() {
    let a = ();
    let b = ();
    let v = vec![&a, &b];
}
```

We've already established that `a` and `b` have different lifetimes, and that the expressions `&a` and `&b` don't have the same type[1]. So why can we make a `Vec` out of these? The reasoning is the same as above, so I'll summarize: `&a` is coerced to `&'b ()`, so that the type of `v` is `Vec<&'b ()>`.

`fn(T)` is a special case in Rust when it comes to variance. `fn(T)` is *contravariant* over `T`. Let's build a `Vec` of functions!

```
fn foo(_: &'static ()) {}
fn bar<'a>(_: &'a ()) {}

fn quux<'a>() {
    let v = vec![
        foo as fn(&'static ()),
        bar as fn(&'a ()),
    ];
}

fn main() {
    quux();
}
```

This compiles. But what's the type of `v` in `quux`? Is it `Vec<fn(&'static ())>` or `Vec<fn(&'a ())>`?

I'll give you a hint:

```
fn foo(_: &'static ()) {}
fn bar<'a>(_: &'a ()) {}

fn quux<'a>(a: &'a ()) {
    let v = vec![
        foo as fn(&'static ()),
        bar as fn(&'a ()),
    ];
    v[0](a);
}

fn main() {
    quux(&());
}
```

This *doesn't* compile. Here are the compiler messages:

```
error[E0495]: cannot infer an appropriate lifetime due to conflicting requirements
 --> <anon>:5:13
  |
5 |     let v = vec![
  |  _____________^ starting here...
6 | |       foo as fn(&'static ()),
7 | |       bar as fn(&'a ()),
8 | |   ];
  | |_____^ ...ending here
  |
note: first, the lifetime cannot outlive the lifetime 'a as defined on the body at 4:23...
 --> <anon>:4:24
  |
4 |  fn quux<'a>(a: &'a ()) {
  |  ________________________^ starting here...
5 | |   let v = vec![
6 | |       foo as fn(&'static ()),
7 | |       bar as fn(&'a ()),
8 | |   ];
9 | |   v[0](a);
10| | }
  | |_^ ...ending here
note: ...so that reference does not outlive borrowed content
 --> <anon>:9:10
  |
9 |   v[0](a);
  |        ^
  = note: but, the lifetime must be valid for the static lifetime...
note: ...so that types are compatible (expected fn(&()), found fn(&'static ()))
 --> <anon>:5:13
  |
5 |     let v = vec![
  |  _____________^ starting here...
6 | |       foo as fn(&'static ()),
7 | |       bar as fn(&'a ()),
```

We're trying to call one of the functions in the vector with a `&'a ()` argument. But `v[0]` expects a `&'static ()`, and there's no guarantee that `'a` is `'static`, so this is invalid. We can therefore conclude that the type of `v` is `Vec<fn(&'static ())>`. As you can see, contravariance is the opposite of covariance: we can replace a short lifetime with a *longer* one.

Whew, now back to your question. First, let's see what the compiler makes out of the call to `hint`. `hint` has the following signature:

```
fn hint<'a, Arg>(_: &'a Arg) -> Foo<'a>
```

`Foo` is *contravariant* over `'a` because `Foo` wraps a `fn` (or rather, *pretends to*, thanks to the `PhantomData`, but that doesn't make a difference when we talk about variance; both have the same effect), `fn(T)` is contravariant over `T` and that `T` here is `&'a ()`.

When the compiler tries to resolve the call to `hint`, it only considers `shortlived`'s lifetime. Therefore, `hint` returns a `Foo` with `shortlived`'s lifetime. But when we try to assign that to the variable `foo`, we have a problem: a lifetime parameter on a type always outlives the type itself, and `shortlived`'s lifetime doesn't outlive `foo`'s lifetime, so clearly, we can't use that type for `foo`. If `Foo` was covariant over `'a`, that would be the end of it and you'd get an error. But `Foo` is *contravariant* over `'a`, so we can replace `shortlived`'s lifetime with a *larger* lifetime. That lifetime can be any lifetime that outlives `foo`'s lifetime. Note that "outlives" is not the same as "strictly outlives": the difference is that `'a: 'a` (`'a` outlives `'a`) is true, but `'a` strictly outlives `'a` is false (i.e. a lifetime is said to outlive itself, but it doesn't *strictly outlive* itself). Therefore, we might end up with `foo` having type `Foo<'a>` where `'a` is exactly the lifetime of `foo` itself.

Now let's look at `check(&foo, &outlived);` (that's the second one). This one compiles because `&outlived` is coerced so that the lifetime is shortened to match `foo`'s lifetime. That's valid because `outlived` has a longer lifetime than `foo`, and `check`'s second argument is covariant over `'a` because it's a reference.

Why doesn't `check(&foo, &shortlived);` compile? `foo` has a longer lifetime than `&shortlived`. `check`'s second argument is covariant over `'a`, but its first argument is *contravariant* over `'a`, because `Foo<'a>` is contravariant. That is, both arguments are trying to pull `'a` in opposite directions for this call: `&foo` is trying to enlarge `&shortlived`'s lifetime (which is illegal), while `&shortlived` is trying to shorten `&foo`'s lifetime (which is also illegal). There is no lifetime that will unify these two variables, therefore the call is invalid.

[1] *That might actually be a simplification. I believe that the lifetime parameter of a reference actually represents the region in which the borrow is active, rather than the lifetime of the reference. In this example, both borrows would be active for the statement that contains the call to `ensure_equal`, so they would have the same type. But if you split the borrows to separate `let` statements, the code still works, so the explanation is still valid. That said, for a borrow to be valid, the referent must outlive the borrow's region, so when I'm thinking of lifetime parameters, I only care about the referent's lifetime and I consider borrows separately.*

Share
Improve this answer
Follow
edited Mar 7 '17 at 20:16

answered Mar 7 '17 at 3:45

Francis Gagné
50.1k ● 3 ● 138 ● 127

13

This is quite possibly the single best answer in the Rust tag. There is so much information here it is amazing. Thank you for taking the time to write it. (I know "thanks" comments are frowned upon ... but come on... look at this answer)
– Simon Whitehead
Mar 7 '17 at 4:40

Add a comment

5

Another way of explaining this is to notice that `Foo` doesn't actually hold a reference to anything with a lifetime of `'a`. Rather, it holds a function that ***accepts*** a reference with lifetime `'a`.

You can construct this same behaviour with an actual function instead of `PhantomData`. And you can even call that function:

```
struct Foo<'a>(fn(&'a ()));

fn hint<'a, Arg>(_: &'a Arg) -> Foo<'a> {
    fn bar<'a, T: Debug>(value: &'a T) {
        println!("The value is {:?}", value);
    }
    Foo(bar)
}

fn main() {
    let outlived = ();
    let foo;
    {
        let shortlived = ();
        // &shortlived is borrowed by hint() but NOT stored in foo
        foo = hint(&shortlived);
    }
    foo.0(&outlived);
}
```

As Francis explained in his excellent answer, the type of `outlived` is a subtype of the type of `shortlived` because its lifetime is longer. Therefore, the function inside `foo` can accept it because it can be coerced to `shortlived`'s (shorter) lifetime.

Share
Improve this answer
Follow

answered Mar 11 '17 at 2:51

Peter Hall
41.4k ● 11 ● 90 ● 157

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 lifetime or ask your own question.

Linked

How do I implement the Chain of Responsibility pattern using a chain of trait objects?

Clarify the meaning of binding two references to differently scoped referents to the same lifetime in a function signature

Do lifetime annotations in Rust change the lifetime of the variables?

Can I force a trait to be covariant?

How to declare generic parameters of the same type except for lifetime in rust?

How do lifetimes in Rust impact mutability?

Does a generic lifetime materialize as the reference's lifetime or the referenced value's lifetime?

2
Making a struct outlive a parameter given to a method of that struct

2
Why does this code with string literals compile when one of them appears to be out of scope?

Return reference wrapped in struct that requires lifetime parameters

See more linked questions

Related

Using a `let` binding to increase a values lifetime

Why is the bound `T: 'a` required in order to store a reference `&'a T`?

1
Why can this lifetime not outlive the closure?

How do I specify lifetimes such that a local referenced value is distinct from a passed in reference?

How can I return an impl Iterator that has multiple lifetimes?

Statically declared struct with method taking a non-static reference

Hot Network Questions

How am I able to simulate gears on my single speed?

Powering 2 5v strips using a 12v power supply

Do all observations arise from probability distributions?

Circuit analysis homework - LTspice results are different from calculations

Efficent way to let objects appear/disappear

more hot questions

Question feed