



Products

Will the non-lexical lifetime borrow checker release locks prematurely?

Log in Sign up

Ask Question

Asked 2 years, 4 months ago

Active 2 years, 4 months ago

Viewed 548 times



6



I've read [What are non-lexical lifetimes?](#). With the non-lexical borrow checker, the following code compiles:

```
fn main() {  
    let mut scores = vec![1, 2, 3];  
    let score = &scores[0]; // borrows 'scores', but never used  
    // its lifetime can end here  
  
    scores.push(4); // borrows 'scores' mutably, and succeeds  
}
```

It seems reasonable in the case above, but when it comes to a mutex lock, we don't want it to be released prematurely.

In the following code, I would like to lock a shared structure first and then execute a closure, mainly to avoid deadlock. However, I'm not sure if the lock will be released prematurely.

```
use lazy_static::lazy_static; // 1.3.0  
use std::sync::Mutex;  
  
struct Something;  
  
lazy_static! {  
    static ref SHARED: Mutex<Something> = Mutex::new(Something);  
}  
  
pub fn lock_and_execute(f: Box<Fn()>) {  
    let _locked = SHARED.lock(); // '_locked' is never used.  
    // does its lifetime end here?  
  
    f();  
}
```

Does Rust treat locks specially, so that their lifetimes are guaranteed to extend to the end of their scope? Must we use that variable explicitly to avoid premature dropping of the lock, like in the following code?

```
pub fn lock_and_execute(f: Box<Fn()>) {  
    let locked = SHARED.lock(); // - lifetime begins  
    f(); // |  
    drop(locked); // - lifetime ends  
}
```

[rust](#) [mutex](#) [raii](#) [borrow-checker](#)

Share

Improve this question

Follow

edited Aug 12 '19 at 20:14



Shepmaster

305k ● 59 ● 824 ● 1083

asked Aug 12 '19 at 19:59



Zhiyao

3,420 ● 2 ● 9 ● 20

Add a comment

2 Answers

[Active](#) [Oldest](#) [Votes](#)



9



Your privacy

There is a misunderstanding here: NLL (non-lexical lifetimes) affects the *borrow-checks*, not the actual *lifetime* of the objects. By clicking "Accept all cookies", you agree Stack Exchange can store cookies on your device and disclose information in accordance with our [Cookie Policy](#).

Rust uses RAII¹ extensively, and thus the Drop implementation of a number of objects, such as locks, has side-effects which *have* to occur at a well-determined and predictable point in the flow of execution.

NLL did NOT change the lifetime of such objects, and therefore their destructor is executed at exactly the same point that it was before: at the end of their lexical scope, in

reverse order of creation.

NLL did change the *understanding of the compiler* of the use of lifetimes for the purpose of borrow-checking. This does not, actually, cause any code change; this is purely analysis. This analysis was made more clever, to better recognize the actual scope in which a reference is used:

- Prior to NLL, a reference was considered "in use" from the moment it was created to the moment it was dropped, generally its lexical scope (hence the name).
- NLL, instead:
 - Tries to defer the start of the "in use" span, if possible.
 - Ends the "in use" span with the last use of the reference.

In the case of a `Ref<a>` (from `RefCell`), the `Ref<a>` will be dropped at the end of the lexical scope, at which point it will *use* the reference to `RefCell` to decrement the counter.

NLL does not peel away layers of abstractions, so *must* consider that any object containing a reference (such as `Ref<a>`) *may* access said reference in its `Drop` implementation. As a result, any object that contains a reference, such as a lock, will force NLL to consider that the "in use" span of the reference extends until they are dropped.

¹ *Resource Acquisition Is Initialization*, whose original meaning is that once a variable constructor has been executed it has acquired the resources it needed and is not in a half-baked state, and which is generally used to mean that the destruction of said variable will release any resources it owned.

[Share](#)

[Improve this answer](#)

[Follow](#)

answered Aug 13 '19 at 8:19



[Matthieu M.](#)

261k ● 40 ● 396 ● 665

Will variable shadowing end a variable's lifetime immediately? (i.e. bind the same name to another object using `let`)

– [Zhiyao](#)

Aug 13 '19 at 17:53

@ZhiyaoMa: Shadowing no. Assigning yes.

– [Matthieu M.](#)

Aug 14 '19 at 6:33

[Add a comment](#)



5



Does Rust treat locks specially, so that their lifetimes are guaranteed to extend to the end of their scope?

No. This is the default for *every* type, and has nothing to do with the borrow checker.

Must we use that variable explicitly to avoid premature dropping of the lock

No.

All you need to do is ensure that the lock guard is bound to a variable. Your example does this (`let _lock = ...`), so the lock will be dropped at the end of scope. If you had used the `_` pattern instead, the lock would have been dropped immediately:

You can prove this for yourself by testing if the lock has indeed been dropped:

```
pub fn lock_and_execute() {
    let shared = Mutex::new(Something);

    println!("A");
    let _locked = shared.lock().unwrap();

    // If `_locked` was dropped, then we can re-lock it:
    println!("B");
    shared.lock().unwrap();

    println!("C");
}

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

This code will deadlock, as the same thread attempts to acquire the lock twice.

You could also attempt to use a method that requires `&mut self` to see that the immutable borrow is still held by the guard, which has not been dropped:

```
pub fn lock_and_execute() {
    let mut shared = Mutex::new(Something);

    println!("A");
    let _locked = shared.lock().unwrap();

    // If `_locked` was dropped, then we can re-lock it:
    println!("B");
    shared.get_mut().unwrap();

    println!("C");
}
```

```
error[E0502]: cannot borrow `shared` as mutable because it is also borrowed as immutable
--> src/main.rs:13:5
|
9 |   let _locked = shared.lock().unwrap();
|   ----- immutable borrow occurs here
...
13 |   shared.get_mut().unwrap();
|   ~~~~~^~~~~~ mutable borrow occurs here
...
16 | }
|_- immutable borrow might be used here, when `_locked` is dropped and runs the `Drop` code for type `std::sync::MutexGuard`
```

See also:

- [Where is a MutexGuard if I never assign it to a variable?](#)
- [How to lock a Rust struct the way a struct is locked in Go?](#)
- [Why does `_` destroy at the end of statement?](#)

Share

Improve this answer

Follow

edited Aug 12 '19 at 20:21

answered Aug 12 '19 at 20:08



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](#) [mutex](#) [rail](#) [borrow-checker](#) 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
[Can the optimizer reorder a drop call to before the variable appears to go out of scope?](#)

117
[What are non-lexical lifetimes?](#)

[Where is a MutexGuard if I never assign it to a variable?](#)

8
[Why does `_` destroy at the end of statement?](#)

[How to lock a Rust struct the way a struct is locked in Go?](#)

Related

[Initialising an anonymous mutex-lock-holding class instance in the LHS of a comma operator](#)

[Fighting with the Rust borrow checker](#)







3
[Satisfying the Rust borrow checker with structs](#)

[Why does Rust borrow checker ignores the mutable pointer among the function parameters?](#)

[Perplexing borrow checker message: "lifetime mismatch"](#)

[When it is not possible to infer the lifetime in the Rust borrow checker?](#)

Hot Network Questions

-  [Are they already planning a successor to the JWST?](#)
-  [Split polyline to equal parts using QGIS](#)
-  [Naruto fighting game with Hulk and Homer Simpson?](#)
-  [Is there any other notation, like tab notation, for piano?](#)
-  [Bit Rot within LUKS Encryption](#)
- [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](#)

site design / logo © 2021 Stack Exchange Inc; user contributions licensed under cc by-sa. rev 2021.12.22.41046