Now I feel like the borrow checker

help

grom March 29, 2023, 3:42pm 1

Why does this compile

```
    #[derive(Debug)]
   struct A; // non-copy non-clone
   fn foo(mut xx: [A; 2]) -> impl FnOnce() -> () {
       move || { dbg!(&mut xx); } // xx borrowed here
   } // xx dropped here
   ^____ xx dropped while borrowed
```

Cerber-Ursi March 29, 2023, 3:48pm 2

Since it's a `move` closure, `xx` is moved inside it and borrowed only during its call. Note that this code doesn't compile:

```
#[derive(Debug)]
struct A;
fn foo(mut xx: [A; 2]) -> impl FnOnce() -> () {
   let closure = move || {
       dbg!(&mut xx);
   };
   dbg!(&mut xx); // use of moved value
   closure
}
```

Neither does this:

```
#[derive(Debug)]
struct A;
fn foo(mut xx: [A; 2]) -> impl FnOnce() -> () {
   || {
       dbg!(&mut xx); // closure may outlive borrowed value
   }
}
```

2 Likes

grom March 29, 2023, 3:54pm 3

So in the context of `move ||` closure, the `&xx` in the body is two operations. Copy/move `xx` then apply the `&` and take a reference? This doesn't happen outside a closure? Just so I'm aware in the future, taking references with `&` of a variable is not going to copy the variable in no other context?

steffahn March 29, 2023, 4:32pm 4

grom:

> So in the context of `move ||` closure, the `&xx` in the body is two operations. Copy/move `xx` then apply the `&` and take a reference? This doesn't happen outside a closure?

No, the moving happens when the closure is *constructed*, the meaning of `&xx` itself does not change at all.

Put simply, a closure is a struct with fields for the captured data.

A closure

```
let closure = || {
   dbg!(&mut xx);
};

// and here's a call demo
closure();
```

will

- implicitly define a struct, somewhat like

  ```
  struct TheClosure<'a> { xx_ref: &'a mut [A; 2] }
  ```

- use the closure body to define a call method, replacing the places where captured variables appear with appropriate field accesses

```
impl TheClosure<'_> {
   fn call(self) -> () {
       dbg!(&mut *self.xx_ref);
   }
}
```

- the closure expression itself will then be replaced to a constructor call for this struct

```
let closure = TheClosure { xx_ref: &mut xx };
```

and calls to the closure desugar to call the `call` method

```
// and here's a call demo
closure.call();
```

Now, the `move` keyword influences the type of the field in this struct. It's not longer a (mutable or immutable) reference, but instead always an owned value of the same type as the variable being captured.

(By the way, I'm being deliberately a bit vague on the whole closure desugaring, because the whole story is quite complex with quite a few corner cases, especially if the finer-grained closure captures we got in edition 2021 are also taken into account. But it's mostly a bunch of little convenience rules to make your life easier, nothing you *actually* need to learn in detail to understand the general principles.)

(I have been and will be also ignoring anything beyond `FnOnce` for simplicity of the desugaring. Otherwise, we'd have a second kind of code analysis to discuss that's being done to the closure body, and up to 3 different `call` methods would be generated for the same closure, and the desugaring of calling the closure would need to choose the right call method to call.)

So with this keyword the thing changes as follows.

The `move` closure

```
let closure = move || {
   dbg!(&mut xx);
};

// and here's a call demo
closure();
```

will

define the struct with an owned value of the type of `xx`

```
struct TheMoveClosure { xx_owned: [A; 2] }
```

- use the closure body to define a call method, now adapted to the new field type, so the dereferencing can go away

```
impl TheMoveClosure {
   fn call(self) -> () {
       dbg!(&mut self.xx_ref);
   }
}
```

the closure expression itself will then be replaced to a constructor call for this struct, and moving the ownership happens *immediately on construction*

```
let closure = TheMoveClosure { xx_owned: xx };
```

and calls to the closure still desugar to call the `call` method

```
// and here's a call demo
closure.call();
```

For determining these desugaring, the general idea is that a closure `move || … code mentioning variable `foo` …` will always capture `foo` by-value, whereas a closure `|| … code mentioning variable `foo` …` will inspect how `foo` is actually used and downgrade the capture to by-mutable-reference or by-shared-reference if that's all that's needed. *Well, and… with finer-grained capturing of struct fields etc. in edition 2021, the full story is a bit more complex, because it's no longer always the whole variable being captured.*

So it's not `&xx` or `&mut xx` that changes meaning, it's just that `move` will make the closure analysis simpler and just see something like `&xx` as "some code that uses `xx`", and no longer as "some code that uses `xx` only in ways where access by shared reference would be sufficient".

5 Likes

This topic was automatically closed 90 days after the last reply. We invite you to open a new topic if you have further questions or comments.

Related topics

Topic		Replies	Views	Activity
Move inside closure code review	11	827	July 26, 2022	
Capturing outer reference in closure help	5	446	May 4, 2021	
Selectively move values into a closure help	17	6482	March 20, 2023	
Why does `move` not appear to move ownership here? help	16	360	March 12, 2025	
Confused about moving references and closures help	9	1153	September 19, 2023	