Rust Closures Under the Hood: Comparing impl Fn and Box<dyn Fn>

Introduction

In this article, we will delve into the inner workings of Rust closures by comparing the assembly code generated for closures returned as `impl Fn` versus `Box<dyn Fn>` . We will examine how captured variables are stored on the stack and the heap, and how dynamic dispatch is implemented for `Box<dyn Fn>` closures. It is recommended to read the article "Rust to Assembly: Static vs Dynamic Dispatch" before diving into this one to get a better understanding of vtables.

Rust closures

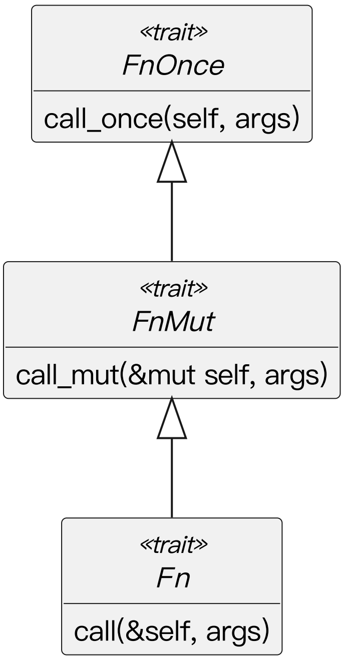
In the Rust programming language, closures are anonymous functions that can capture variables from the scope where they are defined. Closures can be passed as arguments to functions, can be returned as values from functions, and can be assigned to variables. Closures can be called multiple times or can be called only once. Closures can be borrowed immutably, borrowed mutably, or can take ownership of captured variables. Closures can be implemented using the `Fn` , `FnMut` , and `FnOnce` traits.

Rust closure traits

In the Rust programming language, `Fn` , `FnMut` , and `FnOnce` are traits that define different behaviors for closures and other types of function objects.

- The `Fn` trait represents closures that can be called multiple times and can be borrowed immutably. This trait has one associated method `call(&self, args)` that takes a borrowed reference to self, and it can be implemented by any closure that meets these requirements.
- The `FnMut` trait represents closures that can be called multiple times and can be borrowed mutably. This trait has one associated method `call_mut(&mut self, args)` that takes a mutable reference to self, and it can be implemented by any closure that meets these requirements.
- The `FnOnce` trait represents closures that can be called only once. This trait has one associated method `call_once(self, args)` that takes ownership of self, and it can be implemented by any closure that meets these requirements.

From the following diagram, we can see that `FnOnce` is a supertrait of `FnMut` and `FnMut` is a supertrait of `Fn` . If a closure implements `Fn` , it also implements `FnMut` and `FnOnce` as `call_mut` and `call_once` are implemented in terms of `call` .



Now let's look at the assembly code generated for closures returned as `impl Fn` versus `Box<dyn Fn>` .

Return an `impl Fn` from a function

The following example defines function `make_quadratic` that takes three `f64` parameters `a` , `b` , and `c` . It returns a closure that implements the trait `Fn(f64) -> f64` . The closure captures the variables `a` , `b` , and `c` from the surrounding scope and implements the logic of a quadratic equation with input `x` by calculating `a*x*x + b*x + c` and returning the result.

```
pub fn make_quadratic(a: f64, b: f64, c: f64) -> impl Fn(f64) -> f64 {
    move |x| a*x*x + b*x + c
}
```

> Note that `move` keyword before the returned closure signals to the compiler that `a` , `b` , and `c` should be moved into the closure environment. If we don't use the `move` keyword, the closure will borrow the variables `a` , `b` , and `c` immutably. Borrowing the variables `a` , `b` , and `c` immutably is not possible because the closure is returned from the function and they will be dropped when the function returns.

Assembly code for `make_quadratic`

The generated assembly for the function `make_quadratic` is shown below. This example demonstrates how the function is preparing the closure environment that captures the variables `a` , `b` , and `c` . The memory layout of the closure environment is shown in the diagram below (the byte offsets are shown on the left). The closure environment is stored on the stack and is returned in the `rax` register. Note that only the environment needs to be returned.

00 | a	5.0
08 | b	4.0
16 | c	3.0

```
; Input parameters:
;   xmm0: a
;   xmm1: b
;   xmm2: c
;   rdi: Address where the closure should be stored
; Output parameters:
;   rax:  Address of the closure
example::make_quadratic:
        mov     rax, rdi                        ; rax = Address of the closure
        movsd   qword ptr [rdi], xmm0           ; closure.a = a
        movsd   qword ptr [rdi + 8], xmm1       ; closure.b = b
        movsd   qword ptr [rdi + 16], xmm2      ; closure.c = c
        ret                                     ; Return the address of the closure in rax
```

Calling the closure

The following example calls the closure returned by `make_quadratic` .

```
pub fn call_make_quadratic(x: f64) -> f64 {
    let quad_fn = make_quadratic(5.0, 4.0, 3.0);
    quad_fn(x)
}
```

Assembly code for `call_make_quadratic`

The generated assembly for the function `call_make_quadratic` is shown below. We don't see the closure being called as the compiler has inlined the closure.

```
; Input parameters:
;   xmm0: x
; Output parameter:
;   xmm0: Result of the quadratic function
```

```
example::call_make_quadratic:
        movsd   xmm1, qword ptr [rip + .LCPI1_0] ; xmm1 = 5.0
        mulsd   xmm1, xmm0 ; xmm1 = 5.0 * x
        mulsd   xmm1, xmm0 ; xmm1 = 5.0 * x * x
        mulsd   xmm0, qword ptr [rip + .LCPI1_1] ; xmm0 = 4.0 * x
        addsd   xmm0, xmm1 ; xmm0 = 5.0 * x * x + 4.0 * x
        addsd   xmm0, qword ptr [rip + .LCPI1_2] ; xmm0 = 5.0 * x * x + 4.0 * x + 3.0
        ret ; Return the result in xmm0

.LCPI1_0:
        .quad   0x4014000000000000 ; 5.0_f64
.LCPI1_1:
        .quad   0x4010000000000000 ; 4.0_f64
.LCPI1_2:
        .quad   0x4008000000000000 ; 3.0_f64
```

Return a `Box<dyn Fn>` from a function

Now let's look at an example where the closure is returned as a `Box<dyn Fn>` . In this case, the closure is allocated on the heap and a `Box` is returned.

The following example defines function `make_quadratic_box` that takes three `f64` parameters `a` , `b` , and `c` . The function returns a `Box<dyn Fn(f64) -> f64>` that implements the logic of a quadratic equation with input `x` by calculating `a*x*x + b*x + c` and returning the result.

```
pub fn make_quadratic_box(a: f64, b: f64, c: f64) -> Box<dyn Fn(f64) -> f64> {
    Box::new(make_quadratic(a, b, c))
}
```

Assembly code for `make_quadratic_box`

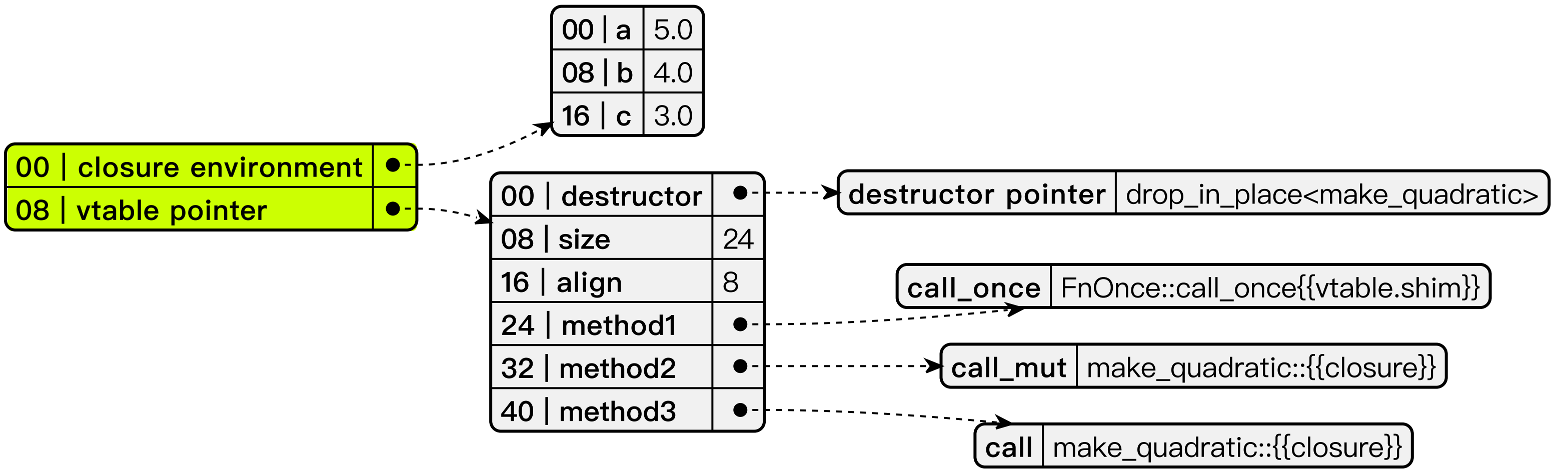
Now let's look at the assembly code generated for the function `make_quadratic_box` . Before we look at the assembly code, let's look at the memory layout of the closure environment.

Memory layout of the `Box<dyn Fn>` closure

`Box<dyn Fn>` is a fat pointer that really is a tuple of the closure environment and the vtable pointer for the type. The closure environment contains the captured variables `a` , `b` , and `c` . The vtable pointer is a pointer to the vtable for the trait `Fn(f64) -> f64` . The vtable contains the function pointers for the trait methods. In this case, the vtable contains the function pointer for the method `call` defined by the `Fn` trait. The vtable also contains the `call_mut` and `call_once` function pointers. In this case, the `call_mut` and `call_once` function pointers are essentially the same as the `call` function pointer.

The vtable also contains the `destructor` function pointer. The `destructor` function is called when the `Box` is dropped. The vtable also saves the size and alignment of the closure environment. The size and alignment are used when the `Box` is dropped.

The diagram below shows the memory layout of the closure environment and the vtable pointer. The fat pointer that will be returned by the function `make_quadratic_box` is highlighted in green. The figure also includes the byte offsets for the memory layout.



Assembly code

The closure environment is stored on the stack and is copied to the heap by calling the function `__rust_alloc`. The address of the closure environment on the heap is returned in the `rax` register. The register `rdx` contains the vtable pointer for the type. The tuple `(rax, rdx)` is being used to return the `Box<dyn Fn>`.

```
; Input parameters:
;    xmm0: a
;    xmm1: b
;    xmm2: c
; Output parameter:
;    rax:  Address of the closure environment.
;    rdx:  Vtable pointer for the trait Fn(f64) -> f64
example::make_quadratic_box:
        sub     rsp, 24                         ; Allocate space for the closure
        movsd   qword ptr [rsp + 16], xmm2      ; closure.c = c
        movsd   qword ptr [rsp + 8], xmm1       ; closure.b = b
        movsd   qword ptr [rsp], xmm0           ; closure.a = a
        mov     edi, 24                         ; Size of the closure
        mov     esi, 8                          ; Alignment of the closure
        ; 📦 Allocate space for the closure (the result is returned in rax).
        call    qword ptr [rip + __rust_alloc@GOTPCREL]
        test    rax, rax                        ; Check if allocation was successful
        je      .LBB4_1                         ; If not, call the error handler.
        movsd   xmm0, qword ptr [rsp]           ; xmm0 = closure.a
        movsd   qword ptr [rax], xmm0           ; closure.a = a
```

```
        movsd   xmm0, qword ptr [rsp + 8]         ; xmm0 = closure.b
        movsd   qword ptr [rax + 8], xmm0         ; closure.b = b
        movsd   xmm0, qword ptr [rsp + 16]        ; xmm0 = closure.c
        movsd   qword ptr [rax + 16], xmm0        ; closure.c = c
        lea     rdx, [rip + .L__unnamed_1]        ; Address of the vtable
        add     rsp, 24                           ; Deallocate space for the closure.
        ; Return the address of the closure in rax and the vtable in rdx.
        ret
.LBB4_1:
        ; ❌ Memory allocation has failed, call the error handler
        mov     edi, 24                          ; Size of the closure.
        mov     esi, 8                           ; Alignment of the closure.
        ; 💀 Call the error handler.
        call    qword ptr [rip + alloc::alloc::handle_alloc_error@GOTPCREL]
        ; 💀 Throw an invalid instruction exception.
        ud2
```

Vtable for returned `Box<dyn Fn>`

We have already seen the memory layout of a vtable. The vtable for the returned `Box<dyn Fn>` is shown below.

```
.L__unnamed_1:
        .quad   core::ptr::drop_in_place<example::make_quadratic::{{closure}}> ; Call when d
        .asciz  "\030\000\000\000\000\000\000\000\b\000\000\000\000\000\000\000" ; Size of the c
        .quad   core::ops::function::FnOnce::call_once{{vtable.shim}} ; Call function for Fn
        .quad   example::make_quadratic::{{closure}} ; Call function for FnMut.
        .quad   example::make_quadratic::{{closure}} ; Call function for Fn.
```

Destructor for the closure

The destructor for the closure is called when the `Box` is dropped. In this case, the destructor is a no-op.

```
core::ptr::drop_in_place<example::make_quadratic::{{closure}}>:
        ret
```

`call` method for the closure

The `call` method for the closure is called when the closure is invoked. The vtable for the closure maps the `call` and `call_mut` methods to the function `make_quadratic::{{closure}}` .

```
; Input parameters:
;   rdi: Pointer to the closure
;   xmm0: x
; Output parameter:
;   xmm0: Result of the quadratic function
```

```
example::make_quadratic::{{closure}}:
        movupd  xmm1, xmmword ptr [rdi]         ; xmm1 = closure.a, closure.b
        movapd  xmm2, xmm0                      ; xmm2 = x
        unpcklpd        xmm2, xmm0              ; xmm2 = x, x
        mulpd   xmm2, xmm1                      ; xmm2 = closure.a * x, closure.b * x
        mulsd   xmm0, xmm2                      ; xmm0 = closure.a * x * x
        unpckhpd        xmm2, xmm2              ; xmm2 = closure.b * x, closure.b * x
        addsd   xmm0, xmm2                      ; xmm0 = closure.a * x * x + closure.b * x
        addsd   xmm0, qword ptr [rdi + 16]      ; xmm0 = closure.a * x * x + closure.b * x +
        ret                                     ; Return the result in xmm0
```

`call_once` shim for the closure

The compiler also generates a shim for the `call_once` method. The compiler generates the shim because the `call_once` method is part of the `FnOnce` trait. The generated code is identical to the call method for the closure but the compiler generates another copy.

Key Takeaways

- The Rust compiler captures the environment and stores it in a closure environment struct.
- If a closure is returned as `impl Fn`, the closure environment is stored on the stack and a thin pointer is returned to the caller.
- In many cases the compiler completely inlines the closure and the closure environment is not stored on the stack.
- If a closure is returned as a `Box<dyn Fn>`, the closure environment is stored on the heap and a fat pointer is returned to the caller. The fat pointer contains the address of the closure environment and the address of the vtable.
- The vtable contains the destructor for the closure environment, the size and alignment of the closure environment, and the `call` method for the closure.

Experiment with the Compiler Explorer

Use the Compiler Explorer to experiment with the code in this article.

Add the following code in the editor window and examine the generated assembly code. You will see that the compiler just returns from `make_quadratic_no_capture`.

```
pub fn make_quadratic_no_capture() -> impl Fn(f64) -> f64 {
    |x| 42.0*x*x + 84.0*x + 0.0
}
```

The `make_quadratic_box_no_capture` function returns the fat pointer in `eax` and `edx`. The `eax` register contains the address of the closure environment and the `edx` register contains the address of the vtable. In this case no closure environment needs to be saved to `eax` just contains `1` .

```
pub fn make_quadratic_box_no_capture() -> Box<dyn Fn(f64) -> f64> {
    Box::new(make_quadratic_no_capture())
}
```