Lecture 7: Closures

Barinov Denis

March 21, 2023

barinov.diu@gmail.com

Closures

Closures

You've already seen closures in homeworks and lectures:

```
let x = 4;
let equal_to_x = |z| z == x;
let y = 4;
assert!(equal_to_x(y));
```

Question: What's the difference between the closures and the functions?

Closures

You've already seen closures in homeworks and lectures:

```
let x = 4;
let equal_to_x = |z| z == x;
let y = 4;
assert!(equal_to_x(y));
```

Question: What's the difference between the closures and the functions?

A closure is an *anonymous function* that can directly *use variables from the scope* in which it is defined.

Closures

Unlike functions, closures infer input and output types since it's more convenient most of the time.

```rust
let option = Some(2);

let x = 3;
// explicit types:
let new: Option<i32> = option.map(|val: i32| -> i32 {
    val + x
});
println!("{:?}", new);  // Some(5)

let y = 10;
// inferred:
let new2 = option.map(|val| val * y);
println!("{:?}", new2);  // Some(20)
```

Closures and traits

Let's try to duplicate `Option::map` functionality with handcrafted function.

```
fn map<X, Y>(option: Option<X>, transform: ...) -> Option<Y> {
    match option {
        Some(x) => Some(transform(x)),
        None => None,
    }
}
```

We need to fill in the `...` with something that transforms an `X` into a `Y`. What it will be?

Closures and traits

We want `transform` to be the callable object. In Rust, when we want to abstract over some property, we use traits!

```
fn map<X, Y, T>(option: Option<X>, transform: T) -> Option<Y>
    where T: /* the trait */ { ... }
```

Let's design it.

Closures and traits

We want `transform` to be the callable object. In Rust, when we want to abstract over some property, we use traits!

```
fn map<X, Y, T>(option: Option<X>, transform: T) -> Option<Y>
    where T: /* the trait */ { ... }
```

Let's design it.

- Idea: compiler generated structure that implements some trait.

Closures and traits

We want `transform` to be the callable object. In Rust, when we want to abstract over some property, we use traits!

```
fn map<X, Y, T>(option: Option<X>, transform: T) -> Option<Y>
    where T: /* the trait */ { ... }
```

Let's design it.

- Idea: compiler generated structure that implements some trait.
- Our trait will have only one function.

Closures and traits

We want `transform` to be the callable object. In Rust, when we want to abstract over some property, we use traits!

```
fn map<X, Y, T>(option: Option<X>, transform: T) -> Option<Y>
    where T: /* the trait */ { ... }
```

Let's design it.

- Idea: compiler generated structure that implements some trait.
- Our trait will have only one function.
- We'll use tuple as input type since we don't have variadics in Rust (and we don't actually need them, at least in this case).

```rust
trait Transform<Input> {
    type Output;
    fn transform(/* self */, input: Input) -> Self::Output;
}
```

Question: Do we need `self`, `&mut self` or `&self` here?

Closures and traits

```
trait Transform<Input> {
    type Output;
    fn transform(/* self */, input: Input) -> Self::Output;
}
```

Question: Do we need `self`, `&mut self` or `&self` here?

Since the transformation should be able to incorporate arbitrary information beyond what is contained in `Input`. Without any `self` argument, the method would look like `fn transform(input: Input) -> Self::Output` and the operation could only depend on `Input` and global variables.

Closures and traits

Question: What do we need exactly: `self`, `&mut self` or `&self`?

Closures and traits

Question: What do we need exactly: `self`, `&mut self` or `&self`?

	User
`self`	Can only call method once
`&mut self`	Can call many times, only with unique access
`&self`	Can call many times, with no restrictions

Closures and traits

Question: What do we need exactly: `self`, `&mut self` or `&self`?

	User
`self`	Can only call method once
`&mut self`	Can call many times, only with unique access
`&self`	Can call many times, with no restrictions

We usually want to choose the highest row of the table that still allows the consumers to do what they need to do.

Closures and traits

Let's start with `self`. In summary, our `map` and its trait look like:

```
trait Transform<Input> {
    type Output;
    fn transform(self, input: Input) -> Self::Output;
}

fn map<X, Y, T>(option: Option<X>, transform: T) -> Option<Y>
    where T: Transform<X, Output = Y>
{
    match option {
        Some(x) => Some(transform.transform(x)),
        None => None,
    }
}
```

Rust uses Fn, FnMut, FnOnce traits to unify functions and closures, similar to what we've invented.

```
pub trait FnOnce<Args> {
    type Output;
    fn call_once(self, args: Args) -> Self::Output;
}

pub trait FnMut<Args>: FnOnce<Args> {
    fn call_mut(&mut self, args: Args) -> Self::Output;
}

pub trait Fn<Args>: FnMut<Args> {
    fn call(&self, args: Args) -> Self::Output;
}
```

Rust uses `Fn`, `FnMut`, `FnOnce` traits to unify functions and closures, similar to what we've invented.

```
pub trait FnOnce<Args> {
    type Output;
    fn call_once(self, args: Args) -> Self::Output;
}

pub trait FnMut<Args>: FnOnce<Args> {
    fn call_mut(&mut self, args: Args) -> Self::Output;
}

pub trait Fn<Args>: FnMut<Args> {
    fn call(&self, args: Args) -> Self::Output;
}
```

Look carefuly as `self`. Every `FnMut` closure can implement `FnOnce` exactly the same way! Same applies to `Fn` and `FnMut`.

The real `map` looks like this:

```
impl<T> Option<T> {
    pub fn map<U, F>(self, f: F) -> Option<U>
    where
        F: FnOnce(T) -> U,
    {
        match self {
            Some(x) => Some(f(x)),
            None => None
        }
    }
}
```

`FnOnce(T) -> U` is another name for our `Transform<X, Output = Y>` bound, and `f(x)` for `transform.transform(x)`.

Returning and accepting closures

Since the closure is a compiler-generated type, it's **non-denotable**, i.e you cannot write its exact type.

```rust
fn return_closure() -> impl Fn() {
    || println!("hello world!")
}
```

`Fn`, `FnMut`, `FnOnce` are traits, and we can benefit from trait objects here too!

```rust
let c1 = || {
    println!("calculating...");
    42 * 2 - 22
};
let c2 = || 42;
let vec: Vec<&dyn Fn() -> i32> = vec![&c1, &c2];
for elem in vec { println!("{}", elem()); }
```

Basically any funtions also implement these traits!

```
fn cast(x: i32) -> i64 {
    (x + 1) as i64
}

fn func(f: impl FnOnce(i32) -> i64) {
    println!("f(42) = {}", f(42));
}

fn main() {
    func(cast)
}
```

So, like everything in Rust, operator () is defined by traits

There's also *function pointers* in Rust. It's not a trait, it's an actual *type* that refers to the code, not data. Unlike closures, they cannot capture the environment.

```rust
fn add_one(x: usize) -> usize { x + 1 }

let ptr: fn(usize) -> usize = add_one;
assert_eq!(ptr(5), 6);

let clos: fn(usize) -> usize = |x| x + 5;
assert_eq!(clos(5), 10);

// error: mismatched types
// let y = 2;
// let clos: fn(usize) -> usize = |x| y + x + 5;
// assert_eq!(clos(5), 10);
```

Closures: capturing

Let's find out how Rust closures decide how to capture the variables.

```
struct T { ... }

fn by_value(_: T) {}
fn by_mut(_: &mut T) {}
fn by_ref(_: &T) {}
```

Closures: capturing

```rust
let x: T = ...;
let mut y: T = ...;
let mut z: T = ...;

let closure = || {
    by_ref(&x);
    by_ref(&y);
    by_ref(&z);

    // Forces `y` and `z` to be at least
    // captured by `&mut` reference
    by_mut(&mut y);
    by_mut(&mut z);

    // Forces `z` to be captured by value
    by_value(z);
};
```

Closures: capturing

This is how closure environment will look like:

```
struct Environment<'x, 'y> {
    x: &'x T,
    y: &'y mut T,
    z: T
}

/* impl of FnOnce for Environment */

let closure = Environment {
    x: &x,
    y: &mut y,
    z: z,
};
```

Closures: capturing

Since this closure implements `FnOnce`, it cannot be called twice:

```
// Ok
closure();
// error: moved due to previous call
// closure();
```

Closures: capturing

What if you need to move out a closure from the scope? In this case, you need to move all the variables even if it's enough to have a shared reference.

```rust
// Returns a function that adds a fixed number
// to the argument. Reminds of Higher Order Functions
// from functional programming!
fn make_adder(x: i32) -> impl Fn(i32) -> i32 {
    |y| x + y
}
fn main() {
    let f = make_adder(3);
    println!("{}", f(1));
    println!("{}", f(10));
}
```

Closures: capturing

```
error[E0597]: `x` does not live long enough
 --> src/main.rs:2:9
  |
2 |     |y| x + y
  |     --- ^ borrowed value does not live long enough
  |     |
  |     value captured here
3 | }
  |  -
  |  |
  |  `x` dropped here while still borrowed
  |  borrow later used here
```

Closures: capturing

Let's use `move` keyword to tell Rust we need to capture by value:

```
fn make_adder(x: i32) -> impl Fn(i32) -> i32 {
    // Compiles just fine!
    move |y| x + y
}

fn main() {
    let f = make_adder(3);
    println!("{}", f(1));  // 4
    println!("{}", f(10));  // 13
}
```

Closures: capturing

Going back to previous example, the closure with `move` keyword will capture all variables by value:

```
let closure = move || {
    by_ref(&x);
    by_ref(&y);
    by_ref(&z);

    // Forces `y` and `z` to be at least
    // captured by `&mut` reference
    by_mut(&mut y);
    by_mut(&mut z);

    // Forces `z` to be captured by value
    by_value(z);
};
```

Closures: capturing

```rust
struct Environment {
    x: T,
    y: T,
    z: T,
}
```

In Rust, there are no fine-grained capture lists like in C++11. *But do we need it*? In practice, we don't (at least the lecturer doesn't know good examples).

Closure type

Every closure have **distinct type**. This implies that in this example `id0`, `id1`, `id2` and `id3` have **different types**.

```
fn id0(x: u64) -> u64 { x }
fn id1(x: u64) -> u64 { x }
fn main() {
    let id2 = || 1;
    let id3 = || 1;
}
```

And this code won't compile:

```
fn make_closure(n: u64) -> impl Fn() -> u64 {
    move || n
}

vec![make_closure(1), make_closure(2)];
vec![(|| 1), (|| 1)];  // Error: mismatched types
```

Closures and optimizations

- We create a structure for the closure, do some moves... It must be expensive!

Closures and optimizations

- We create a structure for the closure, do some moves... It must be expensive!
- Actually, the compiler knows a lot about our code and optimizes it with ease. Most closure calls are inlined and in binary is the same as code without closure.

Closures and optimizations

- We create a structure for the closure, do some moves... It must be expensive!
- Actually, the compiler knows a lot about our code and optimizes it with ease. Most closure calls are inlined and in binary is the same as code without closure.
- Zero cost abstraction!

Questions?