Functional Rust

Performant Software Systems with Rust — Lecture 10

Baochun Li, Professor

Department of Electrical and Computer Engineering

University of Toronto

What Is Functional Programming?

- A programming paradigm where programs are constructed by **applying** and **composing** functions
- Functions are first-class citizens
  - bound to names
  - passed as arguments
  - returned from other functions
- Used in a wide variety of languages such has Haskell and Clozure

Closures

- Closures are **anonymous functions** that can be
  - saved in a variable
  - passed as arguments to other functions
- Closures **capture** values from the environment (or scope) in which they are defined

First Example

- Every so often, our t-shirt company gives away an exclusive, limited-edition shirt as a promotion
- People can optionally add their favorite color to their profile
- If the person chosen for a free shirt has their favorite color set, they get that color shirt
- Otherwise, they get whatever color the company currently has the most of

```rust
#[derive(Debug, Copy, Clone)]
enum ShirtColor {
    Red,
    Blue,
}

struct Inventory {
    shirts: Vec<ShirtColor>,
}
```

```rust
impl Inventory {
    fn most_stocked(&self) -> ShirtColor {
        let mut num_red = 0;
        let mut num_blue = 0;

        for color in &self.shirts {
            match color {
                ShirtColor::Red => num_red += 1,
                ShirtColor::Blue => num_blue += 1,
            }
        }

        if num_red > num_blue {
            ShirtColor::Red
        } else {
            ShirtColor::Blue
        }
    }
```

```rust
impl Inventory {
    fn giveaway(&self, user_preference: Option<ShirtColor>)
        -> ShirtColor {
         user_preference.unwrap_or_else(|| self.most_stocked())
    }
}
```

Closure Type Inference and Annotation

- Closures don’t usually require you to annotate the types of the parameters or the return value
  - because they are not used in an exposed interface like `fn` functions do
  - rather, they are stored in variables and passed as arguments
- But we can add type annotations too, if needed

Closures with Type Annotation

```rust
let expensive_closure = |num: u32| -> u32 {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
};
```

Closures with Type Annotation and Inference

```rust
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
// brackets optional: closure body has only one expression
```

What happens if we compile this code?

```rust
let example_closure = |x| x;

let s = example_closure(String::from("hello"));
let n = example_closure(5);
```

Three Ways of Capturing Values From the Environment

- Borrowing immutably
- Borrowing mutably
- Taking ownership

Let’s look at an example on each

Borrowing Immutably

```rust
fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {list:?}");

    let only_borrows = || println!("From closure: {list:?}");

    println!("Before calling closure: {list:?}");
    only_borrows();
    println!("After calling closure: {list:?}");
}
```

Borrowing Mutably

```rust
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Before defining closure: {list:?}");

    let mut borrows_mutably = || list.push(7);

    borrows_mutably();
    println!("After calling closure: {list:?}");
}
```

Taking Ownership

```rust
use std::thread;

fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {list:?}");

    thread::spawn(move || println!("From thread: {list:?}"))
        .join()
        .unwrap();
}
```

How is unwrap_or_else() in Option<T> defined?

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

`Fn` Traits

- `FnOnce`: closures that can be called once
  - moves captured values out of its body
  - All closures implement at least this trait
- `FnMut`: closures that don’t move captured values out of their body, but may mutate the captured values
- `Fn`: closures that don’t move captured values out of their body, and don’t mutate captured values, or capture nothing

Using the `sort_by_key` method with closures

```rust
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    // closure may be called multiple times
    // so it takes an `FnMut` closure
    list.sort_by_key(|r| r.width);
    println!("{list:#?}");
}
```

Will this work?

```rust
let mut sort_operations = vec![];
let value = String::from("closure called");

list.sort_by_key(|r| {
    sort_operations.push(value);
    r.width
});
```

No, since it moves `value` out of the closure by transferring ownership of `value` to `sort_operations`, and can only be called once — an `FnOnce` closure.

Will this work?

```rust
let mut num_sort_operations = 0;

list.sort_by_key(|r| {
    num_sort_operations += 1;
    r.width
});
```

Yes, since it only captures a mutable reference to `num_sort_operations` — an `FnMut` closure.

Iterators

- Allows you to perform a task on a sequence of items
- Iterators are **lazy** — no effect until you call methods that consume the iterator to use it up

Using an iterator in a `for` loop

```rust
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();

for val in v1_iter {
    println!("Got: {val}");
}
```

The `Iterator` Trait and the `next` Method

```rust
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // methods with default implementations elided
}
```

```rust
let v1 = vec![1, 2, 3];
let mut v1_iter = v1.iter();

assert_eq!(v1_iter.next(), Some(&1));
assert_eq!(v1_iter.next(), Some(&2));
assert_eq!(v1_iter.next(), Some(&3));
assert_eq!(v1_iter.next(), None);
```

`iter()` vs. `into_iter()` vs. `iter_mut()`

- With `iter()`, `next()` returns immutable references to values in the vector
- To create an iterator that takes ownership, call `into_iter()`
- To iterate over mutable references, call `iter_mut()`

Methods that Consume the Iterator

Methods, such as `sum()`, that call `next()` are called **consuming adaptors**, because calling them uses up the iterator

```rust
fn iterator_sum() {
    let v1 = vec![1, 2, 3];
    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();
    assert_eq!(total, 6);
}
```

Methods that Produce Other Iterators

- **Iterator adaptors** don’t consume the iterator
- Instead, they produce different iterators by changing some aspect of the original iterator

Calling the iterator adaptor `map()` to create a new iterator

```rust
let v1: Vec<i32> = vec![1, 2, 3];

v1.iter().map(|x| x + 1);
```

```
note: iterators are lazy and do nothing unless consumed
```

We need to consume the iterator — they are **lazy** — and the closure never gets called!

```rust
let v1: Vec<i32> = vec![1, 2, 3];

// collect() consumes the iterator
let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

assert_eq!(v2, vec![2, 3, 4]);
```

Using Closures that Capture Their Environment

```rust
#[derive(PartialEq, Debug)]
struct Shoe {
    size: u32,
    style: String,
}

fn shoes_in_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
    // filter() takes a closure that captures `shoe_size`
    shoes.into_iter().filter(|s| s.size == shoe_size).collect()
}
```

Required Additional Reading

The Rust Programming Language, Chapter 13.1, 13.2