COMP 737011 - Memory Safety and Programming Language Design

Lecture 9: More Features of Rust

Hui Xu

xuh@fudan.edu.cn

Outline

1. Functional Programming
2. Metaprogramming with Macros

1. Functional Programming

Functional Programming

- Function is the first-class citizen, which is similar as variables
- It can be used as right value:
  - Assignment
  - Return value
  - Function parameter: known as high-order function

Function Types

- Use the "fn" keyword to denote a function.
- unsafe fn is a super type of safe fn.

```
fn foo<T:Display> (x: T) {
    println!("{}", x);
}

type TyNew = fn(i32) -> ();
type TyUnsafe = unsafe fn(i32) -> ();
let f1 = foo::<i32>;
f1(1);
let f2:TyUnsafe = f1;
unsafe { f2(2) };
let f3:TyNew = f2;
f3(3);
```

Function as A Parameter

```
fn add(a:i32, b:i32) -> i32 {
    a + b
}

fn hofn(v1: i32, v2: i32, f: fn(i32, i32) -> i32) -> i32 {
    f(v1, v2)
}

hofn(1, 2, add);
```

add is a function parameter

```
fn hofn(v1: i32, v2: i32, f: unsafe fn(i32, i32) -> i32) -> i32 {
    unsafe {
        f(v1, v2)
    }
}
```

```
fn hofn<F>(v1:i32, v2:i32, f: F) -> i32
    where F: Fn(i32, i32) -> i32 {
    f(v1, v2)
}
```

Function as a Return Value

```
fn add(a:i32, b:i32) -> i32 {
    a + b
}

fn hofn<F>(f: F) -> F
    where F: Fn(i32, i32) -> i32 {
    f
}

println!("{}", hofn(add)(1, 2));
```

The example is trivial.

Closure (anonymous function) as a Return Value

```
fn hofn(x:u32) -> Box<dyn Fn(u32) -> u32> {
    Box::new(move |y| {
        x + y
    })
}

fn main() {
    hofn(10)(10);
}
```

Closure

- Anonymous functions
  - Parameters are wrapped with ||, e.g., |x|.
  - Body is wrapped with {}, which can be omitted for a single statement.
- It can automatically capture the enclosing environment

```
fn hofn<F>(v1:i32, v2:i32, f: F) -> i32
    where F: Fn(i32, i32) -> i32 {
    f(v1,v2)
}

let i = 10;
let cl = |a, b| {a+b+i};
let r = hofn(20, 10, cl);
```

- a and b are parameters
- i is a captured variable

Closure also has Ownership

- Borrow check: same as other types.
- Immutable closure cannot mutate the captured variables.

```
fn hofn<F>(v1:i32, v2:i32, f: F) -> i32
    where F: Fn(i32, i32) -> i32 {
    f(v1,v2)
}


let i = Box::new(10);
let cl1 = move |a, b| {a+b+*i};
let cl2 = &cl1;
let x = cl2(1,2);
let r1 = hofn(20, 10, cl1);
```

Borrow the ownership of cl1

Transfer the ownership of cl1

Ownership of Captured Variables

- Captured variables are immutable borrow by default
  - Move the ownership via the keyword move
  - Change the mutability via the mutability of the closure

```
let mut i = Box::new(10);
let mut cl1 = |a, b| {*i = *i+1; a+b+*i };
let x = cl1(1,2);
let mut cl2 = move |a, b| {*i = *i+1; a+b+*i };
let y = cl2(1,2);
println!("{}{}", x, y);
```

borrow the ownership of i

move the ownership of i

Ownership of Captured Variables

```
let mut i = Box::new(10);
let mut cl = move |a, b| {*i = *i+1; a+b+*i };
let y = cl(1,2);
let x = cl(1,2);
println!("{}{}", x, y);
```

move once

use multiple times

Characteristics of Closure

- Lazy Evaluation
  - Do not evaluate the function until needed
- No side effect
  - The captured variable is immutable by default

Trait Bound for Closures

- Fn: the closure cannot mutate its state
  - The most rigorous bound
- FnMut: the closure can mutate the state
  - Derived from FnOnce
- FnOnce: the closure cannot be called twice
  - All closures meet the bound

Fn

- The closure satisfies the trait bound cannot mutate its state.

```
fn callfn<F>(f: F)
    where F: Fn() -> i32 {
        println!("{}", f());
        println!("{}", f());
}

let mut y = 1;
let f3 = move || y;
callfn(f3);
```

f3 does not mutate the state

FnMut

- The closure can mutate the state.
- FnMut is a super trait of Fn.
- Any instance of Fn can be used where FnMut is expected.

```
fn callmut<F>(mut f: F)
    where F: FnMut() -> i32 {
        println!("{}", f());
        println!("{}", f());
}

let mut y = 1;
let f2 = move || { y = y*2; return y};
callmut(f2);
```

f2 mutates y

FnOnce

- The closure cannot be called twice
- FnOnce is a super trait of FnMut.
- Any instance of FnMut/Fn meets FnOnce.

```
fn callonce<F>(f: F)
    where F: FnOnce() -> String {
    println!("{}", f());
    println!("{}", f());
}

let x = String::from("x");
let f = move || x;
callonce(f);
```

all closures implement FnOnce

error, f cannot be called twice

f meets the bound of FnOnce

Closure vs Functions

	Closure	Functions
Named?	Anonymous	Yes
Capture Environment	Yes	No
FnOnce	Yes	Yes
FnMut	depends on its captured variables	Yes
Fn		Yes
Copy		Yes
Clone		Yes
Send		Yes
Sync		Yes

Typical Application: Iterator

```
let mut v:Vec<u32> = (1..100).collect();
let iter1 = v.iter();
let sum1:u32 = iter1().sum();     sum() consumes the iterator

let iter2 = v.iter().filter(|x| *x % 2 as u32 == 0);
let sum2:u32 = iter2().sum();
                                        closure parameters

println!("sum = {:?}{:?}", sum1, sum2);

let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();
println!("v2 = {:?}", v2);        collect() consumes the iterator
```

Implement Iterator

- We generally use a proxy struct for the iterator
  - Why? Because it consumes the ownership
- filter()/map() are available by default, but not collect()

```
struct List {
    val: i32,
    next: Option<Box<List>>,
}

struct ListIter<'a> {
    current: Option<&'a List>,
}

impl List {
    fn iter(&self) -> ListIter {
        ListIter {
            current: Some(self),
        }
    }
}
```

a proxy struct for iterating over the List

Sample Iterator Function

```
impl<'a> Iterator for ListIter<'a> {
    type Item = i32; // required by the Iterator trait

    fn next(&mut self) -> Option<Self::Item> {
        match self.current {
            Some(node) => {
                let val = node.val;
                self.current = node.next.as_deref();
                Some(val)
            }
            None => None,
        }
    }
}

let list = List { val: 1, next: None };
for val in list.iter() {
    println!("{}", val); // prints: 1 2 3
}
```

Iterator is Efficient

- Iterator does boundary check only once.
- For loop requires boundary check twice.
  - Loop condition + Boundary check

```
let len = 1000000;
let mut vec:Vec<usize> = (1..len).collect();
let start = Instant::now();
for i in vec.iter_mut(){
    *i += 1;
}
println!("{:?}", start.elapsed().as_nanos());

let start = Instant::now();
for i in 0..len-1 {
    vec[i] = vec[i]-1;
}
println!("{:?}", start.elapsed().as_nanos());
```

```
32392039
230681881
```

2. Metaprogramming with Macros

Metaprogramming

- Read other code (data) and produce code to execute.
- Minimize the lines of code to express a solution.
- Macro is a typical compile-time approach.

Declarative Macros with macro_rules!

- Code rewrite during compile preprocess.

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

$x: expression named x
,*: multiple expressions separated by comma

```
let v: Vec<u32> = vec![1, 2, 3];
```

⇨

```
let mut temp_vec = Vec::new();
temp_vec.push(1);
temp_vec.push(2);
temp_vec.push(3);
temp_vec
```

Create a Macro for List

```
#[macro_export]
macro_rules! list {
    ( $( $x:expr ),+ ) => {
        {
            let mut head = None;
            let mut tmp = &mut head;
            $(
                let node = Box::new(List{val:$x, next:None});
                *tmp = Some(node);
                tmp = &mut tmp.as_mut().unwrap().next;
            )+
           head
        }
    };
}

let l1 = List![1,2,3];
let l2 = List![1,2,3,4,5];
```

!!!Be careful of unsafe code when defining macros!

In-Class Practice

- Extend your binary search tree with:
  - An iterator, *e.g.,* in-order traversal
  - A macro constructor