COMP 737011 - Memory Safety and Programming Language Design

# Lecture 11: More Features of Rust

徐 辉
xuh@fudan.edu.cn

# Outline

1. Functional Programming
2. Metaprogramming with Macros

# 1. Functional Programming

# Functional Programming

- Function is the first class citizen (similar as variables) and can be used as
  - Rvalue (Assignment)
  - Parameter (High-Order Function)
  - Return value (Lazy Evaluation).

# Function as A Parameter

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

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

fn main() {
    hofn(1, 2, add);
}
```

add is a function parameter

# Closure

- Anonymous functions that can capture the enclosing environment
  - Parameters are wrapped with ||, e.g., |x|.
  - Function body is wrapped with {}
    - Can be omitted for a single expression

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

fn main() {
    let i = 10;
    let cl = move |a, b| {a+b+i};
    let result = hofn(20, 10, cl);
}
```

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

# Boxed Function as A Return Value

```
fn hofn(len:u32) -> Box<dyn Fn(u32) -> u32> {
    let vec:Vec<u32> = (1..len).collect();
    let sum:u32 = vec.iter().sum();
    Box::new(move |x| {
        sum + x
    })
}

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

# Other Functional Programming Languages

- Originates from Lambda Calculus by Alonzo Church
- Many functional programming languages
  - Common Lisp, Scheme, Clojure, Haskell, Ocaml, etc
- Languages with functional programming features
  - Lambda expression in C++
  - …

```
auto plus_one = [](const int value) {
    return value + 1;
};
assert(plus_one(2) == 3);
```

# Benefit

- Lazy Evaluation
  - Do not evaluate the function until needed
- No side effect
  - Recursion is preferred over iteration

# Lazy Evaluation

```
use std::collections::HashMap;
use std::{thread,time};

fn main() {
    let mut hmap = HashMap::new();
    let mut insert = |x: i32| {
        println!("enter closure...");
        match hmap.get(&x) {
            Some(&val) => (),
            _ => {
                thread::sleep(time::Duration::new(5,0));
                hmap.insert(x, "123");
            }
        };
    };

    println!("Before insertion...");
    insert(1);
    println!("After the first insertion...");
    insert(1);
    println!("After the second insertion...");
}
```

result can be cached for reuse

# Function Type

- Use the "fn” keyword to denote a function.

```
use std::fmt::Display;

fn main() {
    fn foo<T:Display>(x:T) { println!("{}",x); }
    let fn1 = &mut foo::<i32>;
    //*x = foo::<u32>; //~ ERROR mismatched types
    fn1(1);
    type Binop = fn(i32) -> ();
    let fn2:Binop = foo::<i32>;
    fn2(2);
}
```

# Traits Implemented by Closure Types

- The traits auto implemented by a closure
  - FnOnce: all closures
  - FnMut: a closure does not move out of any captured variables
  - Fn: a closure does not mutate or move out of any captured variables
  - Copy/Clone/Send/Sync: depends on all captured variables

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

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

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

# Trait Bound for Closures

- Use trait to bound closures/functions
  - Fn: the most rigorous bound which means the closure should not mutate its state
  - FnMut: either the function mutate the state or not is acceptable
  - FnOnce: all closures meet the bound, but the code cannot if the closure is called twice
- All function items implement
  - Fn/FnMut/FnOnce

# FnOnce

- FnOnce is a supertrait of FnMut
- Any instance of FnMut can be used where a FnOnce is expected

```
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

# FnMut

- FnMut is a supertrait of Fn
- Any instance of Fn can be used where FnMut is expected

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

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

f2 mutates y

# Fn

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

fn main(){
    let mut y = 1;
    let f3 = move || y;
    callimmut(f3);
}
```

f3 does not mutate the state

# Closure vs Functions

<table>
<tr><th></th><th>Closure</th><th>Functions</th></tr>
<tr><td>Named?</td><td>Anonymous</td><td>Yes</td></tr>
<tr><td>Capture Environment</td><td>Yes</td><td>No</td></tr>
<tr><td>FnOnce</td><td>Yes</td><td>Yes</td></tr>
<tr><td>FnMut</td><td rowspan="6">depends on its captured variables</td><td>Yes</td></tr>
<tr><td>Fn</td><td>Yes</td></tr>
<tr><td>Copy</td><td>Yes</td></tr>
<tr><td>Clone</td><td>Yes</td></tr>
<tr><td>Send</td><td>Yes</td></tr>
<tr><td>Sync</td><td>Yes</td></tr>
</table>

# Typical Applications: Iterator

```
fn main() {
    let mut v:Vec<u32> = (1..100).collect();
    let iter1 = v.iter();
    let sum1:u32 = iter1().sum();

    let iter2 = v.iter().filter(|x| *x % 2 as u32 == 0);
    let sum2:u32 = iter2().sum();
    println!("sum = {:?}{:?}", sum1, sum2);

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

sum() consumes the iterator

closure parameters

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>{    a proxy struct for iterate over the List objects
    val: i32,
    next: &'a Option<Box<List>>,
}
impl List {
   fn iter(&self) -> ListIter {
        ListIter{val: self.val, next: &self.next, }
    }
}
impl <'a> Iterator for ListIter<'a> {
    type Item = i32;
    fn next(&mut self) -> Option<Self::Item> {
        ...
    }
}
```

# Sample Iterator Function

```
impl <'a> Iterator for ListIter<'a> {
    type Item = i32;
    fn next(&mut self) -> Option<Self::Item> {
        let ret = self.val;
        static mut flag:bool = true;
        match self.next {
            Some(ref node) => {
                self.next = &(node).next;
                self.val = node.val;
                return Some(ret);
            }
            None => (),
        }
        unsafe {
            if flag == true {
                flag = false;
                return Some(ret);
            }
        }
        return None;
    }
}
```

# Iterator is Efficient

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

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

# 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

# Recall The Linked List

- How to create a constructor for MyList?

```
struct List{ val: u64, next: Option<Box<List>>, }
Impl MyList {
    fn from(a:&[i32]) -> MyList { ... }
    fn from(a:Box<i32>) -> MyList { ... }
    fn from(a:i32, b:i32,...) -> MyList { ... }
}
```

need an extra slice

need an extra box

variadic function is unsupported

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

# 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 The Linked 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
        }
    };
}
```

Be careful of Interior Unsafe within Macros!

# Procedural Macros

- Custom #[derive] macros that specify code added with the derive attribute used on structs and enums
- Attribute-like macros that define custom attributes usable on any item
- Function-like macros that look like function calls but operate on the tokens specified as their argument

# In-Class Practice

- Extending your binary search tree or double-linked list to support generic parameters.
- Implement the PartialEq and PartialOrd traits for your struct.