COMP 737011 - Memory Safety and Programming Language Design

Lecture 9: Rust Functional Programming

徐 辉
xuh@fudan.edu.cn

Outline

- 1. Functional Programming
- 2. Function Type in Rust
- 3. Typical Applications in Rust
- 4. More

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

- Closures are 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

- Functional programming evolves 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

2. Function Type in Rust

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

	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

3. Typical Applications in Rust

Iterator

```
fn main() {
    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();
    println!("sum = {:?}{:?}", sum1, sum2);     closure parameters

    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>{
    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> {
        ...
    }
}
```

a proxy struct for iterate over the List objects

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());
}
```

In-Class Practice

- Extend your program with an iterator
  - Demonstrate filter
  - Optional: try collect() with map()

4. More

History: Computable Problem

David Hilbert
(23 unsolved problems)
1900

Kurt Gödel
(incompleteness theorems)
1931

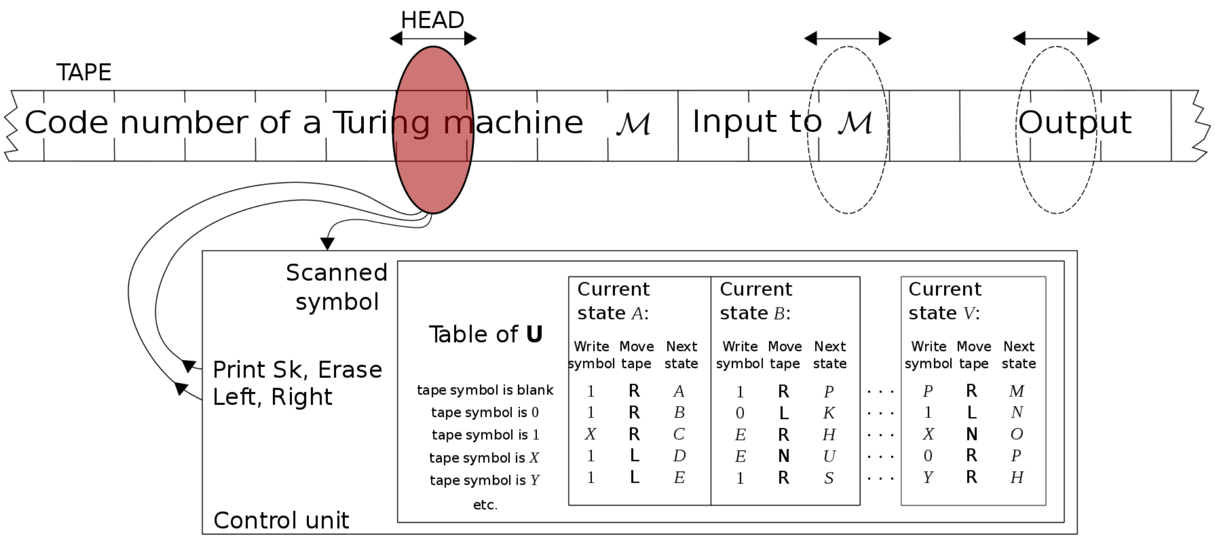

Alonzo Church, Alan Turing
(computable problem)
1936

- Hilbert's program
  - Completeness: all of mathematics follows from a correctly chosen finite system of axioms;
  - Consistency: such axiom system is provably consistent

- Counter example: "This statement is not provable"
  - If complete, then the statement should be provable (inconsistent);
  - If consistent, the system is incomplete.

- Which problems can be computed in limited time?
- Lambda Calculus
- Turing machine
  - On Computable Numbers, with an Application to the Entscheidungsproblem

Turing Machine => Imperative Language



- Universal Turing Machine:
  - A tape of cells, and each cell contains a symbol of a finite set;
  - A head that reads/writes the tape;
  - A state register;
  - A finite table of instructions: the next move based on the register state and the symbol.
- => Von Neumann-style computer

Lambda Calculus => Functional Language

Name	Syntax	Example	
Variable	x		
Abstraction	$(\lambda x.M)$	$\lambda x.x + 2$	Function: $f(x) = x + 2$
Application	(M,N)	$(\lambda x.x + 2)\ 1$	Evaluation: $f(1)$

Evaluation: β reduction

$$(\lambda f.f\ 1)(\lambda x.x+2)$$

$$\Rightarrow (\lambda x.x + 2)\ 1$$

$$\Rightarrow 1 + 2$$

Anonymous Recursion with Fixed Point

β-normal form: $(\lambda x.x\ x)(\lambda x.x\ x)$

$\Rightarrow (\lambda x.x\ x)(\lambda x.x\ x)$

$X = (\lambda x.f(x\ x))(\lambda x.f(x\ x))$

$\Rightarrow X = f(\lambda x.f(x\ x)\ \lambda x.f(x\ x))$

$\Rightarrow X = f(X)$

Y Combinator: $\lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x))$

$$Yf = f(Yf) = f(f(Yf)) = f(f(\ldots f(Yf)\ldots))$$

Returns a fixed point with any input function

Example: Factorial Function

F := λf.λx.(if x == 0 then 1 else x * f (x-1))

YF 3
F(YF) 3
λf.λx.(if x == 0 then 1 else x * f (x-1)) (YF) 3
if 3 == 0 then 1 else 3 * (YF)(3-1)
3 * (YF) 2
3 * F(YF) 2
3 * (λf. λx.(if x == 0 then 1 else x * f (x-1)) (YF) 2)
3 * (if 2 == 0 then 1 else 2 * (YF)(2-1))
3 * (2 * (YF) 1)
6 * (YF) 1
6 * F(YF) 1
6 * (λf. λx.(if x == 0 then 1 else x * f (x-1)) (YF) 1)
6 * (if 1 == 0 then 1 else 1 * (YF)(1-1))
6 * (YF) 0
6 * F(YF) 0
6 * (λf. λx.(if x == 0 then 1 else x * f (x-1)) (YF) 0)
6 * (if 0 == 0 then 1 else 0* (YF)(0-1))
6 * 1
6

Y Combinator in Rust

```rust
trait Apply<T, R> {
    fn apply(&self, f: &dyn Apply<T, R>, t: T) -> R;
}
impl<T, R, F> Apply<T, R> for F where F: Fn(&dyn Apply<T, R>, T) -> R {
    fn apply(&self, f: &dyn Apply<T, R>, t: T) -> R {
        self(f, t)
    }
}
fn y<T, R>(f: impl Fn(&dyn Fn(T) -> R, T) -> R) -> impl Fn(T) -> R {
    move |t| (&|x: &dyn Apply<T, R>, y| x.apply(x, y))
             (&|x: &dyn Apply<T, R>, y| f(&|z| x.apply(x, z), y), t)
}
fn fac(n: usize) -> usize {
    let almost_fac = |f: &dyn Fn(usize) -> usize, x|
                        if x == 0 { 1 }
                        else { x * f(x - 1) };
    y(almost_fac)(n)
}
fn main() {
    println!("fac(10) = {}", fac(10));
}
```

https://rosettacode.org/wiki/Y_combinator#Rust

More Reference

- https://doc.rust-lang.org/reference/types/function-item.html
- https://doc.rust-lang.org/reference/types/closure.html
- https://aloso.github.io/2021/03/09/creating-an-iterator.html
- https://en.wikipedia.org/wiki/Fixed-point_combinator#Y_combinator
- https://rosettacode.org/wiki/Y_combinator#Rust