

CMSC 330: Organization of Programming Languages

Closures and Iterators In Rust

CMSC 330 - Spring 2021

Closures

- Syntax

- $| \mathbf{x1}[:t1]^? , \dots , \mathbf{xn}[:tn]^? | [-> u]^? e$

- Type annotations are optional – will be inferred if absent

- Evaluation

- A closure is a **value**

- Type checking

- has type $(t1, \dots, tn) \rightarrow u$

- when $e : u$ under assumptions $\mathbf{x1} : t1, \dots, \mathbf{xn} : tn$

- Not curried

Using Closures/Functions Locally

- Rust has local functions, and closures

```
fn moveit(l:bool,x:i32) -> i32 {
  let left = |x| x - 1;
  fn right(x:i32) -> i32 { x+1 };
  if l { left(x) }
  else { right(x) }
}
```

Closure (may
have an
environment)

Local function
(no
environment)

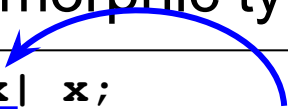
- OCaml local functions/closures

```
let moveit l x =
  let left = fun x -> x - 1 in
  let right = fun x -> x + 1 in
  if l then left x
  else right x
```

Limits of Type Inference

- Rust infers non-polymorphic types

```
let id = |x| x;  
let x = id(1); //infers x:i32  
let y = id("hi"); //fails: &str ≠ i32
```



- OCaml infers polymorphic types

```
let f = fun x -> x in (* 'a -> 'a *)  
let x = id 1 in  
let y = id "hi" in (* OK *) ...
```

- More details on closures at the end, including polymorphism
 - Now for something (not so completely) different

Iteration using the `Iterator` Trait

- Recall an earlier example:

```
let a = vec![10, 20, 30, 40, 50];
for e in a.iter() {
    println!("the value is: {}", e);
}
```

- The `iter()` method returns an *iterator*, i.e., a value with the `Iterator` trait

```
trait Iterator {
    type Item; //this is an associated type
    fn next(&mut self) -> Option<Self::Item>;
    ... //default method impls
}
```

Unpacking the `for` syntax

- Each call to `next` advances the iterator
 - So it has to be `mut`

```
let a = vec![10, 20];  
let mut iter = a.iter();  
assert_eq!(iter.next(), Some(&10));  
assert_eq!(iter.next(), Some(&20));  
assert_eq!(iter.next(), None);
```

- calls to `next` produce **immutable references** to the values in `a`
 - else may call `into_iter` or `iter_mut` on `a` to get different sorts of references

Iterator Adaptors

- We can make one iterator from another
 - An iterator is **consumed as it used**; it is *lazy*
- This is a pattern for higher order programming
 - `i.map(f)` produces an iterator returning `f(e)` for each of `i`'s elements `e`
 - `i.filter(f)` produces iterator for `i`'s elements `e` such that `f(e) == true`
 - `i.collect()` converts an iterator into a vector
 - `i.fold(a, f)` is like OCaml's `fold_right`
 - `fold_right f a v` where `v` is the list corresponding to `i`
 - `zip, sum, ...`

Examples

```
let a = vec![10,20];
let i = a.iter();
let j = i.map(|x| x+1).collect(); // [11,21]
let k = a.iter().fold(0, |a,x| x-a); // 10
for e in a.iter().filter(|&&x| x == 10) {
    println!("{}", e);
} // prints 10
```


Quiz 1: Output of the following code

```
fn main() {  
    let a = [0, 1, 2, 3, 4, 5];  
    let mut iter2 = a.iter().map(|x| 2 * x);  
    iter2.next();  
    let t2 = iter2.next();  
    println!("{:?}", t2)  
}
```

- A. Some(0)
- B. Some(1)
- C. Some(2)
- D. Some(4)

Quiz 1: Output of the following code

```
fn main() {  
    let a = [0, 1, 2, 3, 4, 5];  
    let mut iter2 = a.iter().map(|x| 2 * x);  
    iter2.next();  
    let t2 = iter2.next();  
    println!("{:?}", t2)  
}
```

- A. Some(0)
- B. Some(1)
- C. Some(2)
- D. Some(4)

Iterator Notes

- You can make your own iterators too
 - Implement the Iterator trait
 - Several examples in the Rust Book
- Iterators perform extremely well
 - Better than for loops with explicit indexes!
 - This is because Rust aggressively optimizes the code it generates, e.g., by unrolling the iteration loop
 - So feel free to program using map, fold, zip, etc.

Back to Closures: Passing as Arguments

- Each closure has a distinct type
 - Even if two closures have the **same signature**, their **types are considered different**
 - Such types are called *generative* types
- To specify the **type of a closure** (for a function parameter, say), use **generics with trait bounds**
 - **Fn** *t* (*will describe later*)
 - **FnMut** *t*
 - **FnOnce** *t*
- Functions (defined with **fn** *f*...) implement the above trait bounds too

Using the Fn Trait

Trait bound on T to
specify type of f

```
fn app_int<T>(f:T,x:i32) -> i32
    where T:Fn(i32) -> i32
{
    f(x)
}
fn main() {
    println!("{}",app_int((|x| x-1),1));
}
```

– But cannot write

```
fn app_int(f:(i32) -> i32,x:i32) -> i32
{ f(x) }
```

- Can also use function trait bounds in struct, enum, etc. definitions

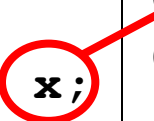
Using the Fn Trait Polymorphically

```
fn app<T,U,W>(f:T,x:U) -> W
    where T:Fn(U) -> W
{
    f(x)
}
fn main() {
    println!("{}",app(|x| x-1),1);//i32
    let s = String::from("hi ");
    println!("{}",app(|x| x+"there",s));//String
}
```

Capturing Free Variables

```
fn main() {  
    let x = 4;  
    let equal_to_x = |z| z == x;  
    let y = 4;  
    assert!(equal_to_x(y))  
} // true
```

Closure
env
captures **x**



- Note: fails if `equal_to_x` defined as a local function
 - Local functions do not have an environment
- Complication: What if **x** is owned?
 - Capturing it could move it or borrow (mut or immut)
 - Use various **FnX** traits to specify what to do

Distinguishing Fn Trait Bounds

- **FnOnce** *t* (where *t* is a func type)
 - Consumes the variables it captures from its enclosing scope (i.e., moves or copies them)
 - Thus can only be called once
 - The call consumes ownership
- **FnMut** *t*
 - Borrows captured variables mutably
- **Fn** *t*
 - Borrows captured variables immutably, or copies
 - `equal_to_x` copied `x` due to its `Copy` trait
 - Try this bound first; follow the compiler's advice if it doesn't work

Example use of FnOnce

```
let x = String::from("hi");  
let add_x = |z| x+z; //captures x; is FnOnce  
println!("{}",x); //fails  
let s = add_x(" there"); //consumes closure  
let t = add_x(" joe"); //fails, add_x consumed
```