CMSC 330: Organization of Programming Languages

Closures and Iterators In Rust

CMSC330 Spring 2018

Copyright © 2018 Michael Hicks, the University of Maryland. Some material based on https://doc.rust-lang.org/book/second-edition/index.html

Closures

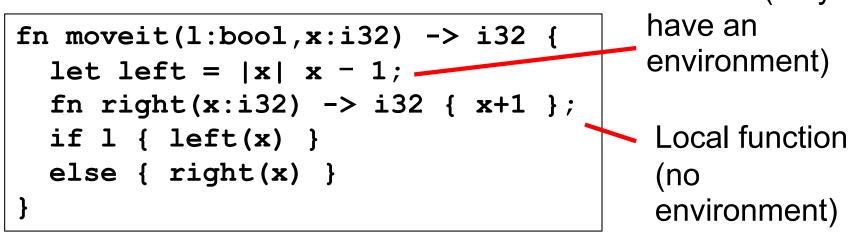
- Syntax
 - $|x1[:t1]^{?}, ..., xn[:tn]^{?}| [-> u]^{?} e$
 - Type annotations are optional will be inferred if absent
- Evaluation
 - A closure is a value
- Type checking
 - has type (t1, ..., tn) -> u

when **e** : **u** under assumptions **x1** : **t1**, ..., **xn** : **tn**

• Not curried

Using Closures/Functions Locally

• Rust has local functions, and closures Closure (may



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

OCaml infers polymorphic types

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

• We'll see polymorphically typed closures shortly

Passing Closures 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 Using the Fn Trait fn app_int<T>(f:T,x:i32) -> i3 where T:Fn(i32) -> i32 { f(x) } fn main() { println!("{}", app_int((|x| x-1),1)); }

But cannot write

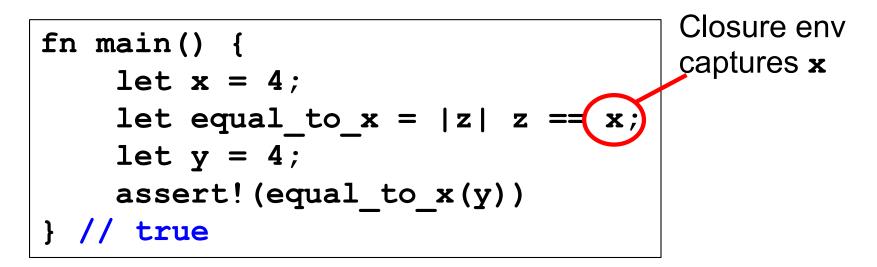
```
fn app_int(f:(i32) -> i32,x:i32) -> i32
{ f(x) }
```

 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



– 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
 - In general, try this first, and 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 = {}",x); //fails
let s = add_x(" there");//consumes closure
let t = add_x(" joe");//fails, add_x consumed
```

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)

Notes

- You can make your own iterators too
 - Implement the Iterator trait
 - Several examples in the Rust Book
- Iterators perform extremely well
 - Better that 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.