Closures and Iterators

Rust, in Practice and in Theory
Lecture 8

CAS CS 392

Closures

High Level

```
fn main() {
    fn  square (x : i32) -> i32 { x * x }
    let square_cls = |x| { x * x };
    assert_eq!(square(2), 4);
    assert_eq!(square_cls(2), 4);
}
```

Closures are anonymous functions, like in OCaml or Python

The big difference: Closures can *capture* values, and this can affect ownership

Common Use-case

```
fn main() {
    let v: Vec<i32> = vec![1, 2, 3, 4, 5];
    for s in v.into_iter().map(|x| x * x) {
        print!("{s}")
    }
}
```

The most common use case of closures is in functional patterns like mapping and and iterating

Example: Counters

```
fn mk_counter() -> impl FnMut() -> i32 {
    let mut count = 0;
    return move || { count += 1; count }
}
fn main() {
    let mut f = mk_counter();
    assert_eq!(f(), 1);
    assert_eq!(f(), 2);
}
```

The types get wonky very fast for basic examples.

Type Inference

```
fn main() {
    let v = vec![1, 2, 3];
    let w = vec![1, 2, 3];
    let id = |x| x;
    assert_eq!(id(v), w);
    // let x = id(2);
}
```

Rust does some type inference for closures, we rarely need to include type annotations

That said, closures are *monomorphic*

Borrow Inference

```
fn main() {
    let mut v = vec![1, 2, 3, 4, 5];
    let mut f = || { v.push(6) };
    v.push(8);
    f();
}
```

Rust also determines to what extent captured values need to be borrowed or moved

Moving/Borrowing happens when the closure is defined

Closures and Traits

```
fn main() {
    let mut v = vec![1, 2, 3];
    let f = || v; // FnOnce only
    // let f = || v.push(4); // Not Fn
    // let f = || println!("{}", v[0]); // All three
}
```

Closures are structures which satisfy a trait. There are three kinds of closures:

- **FnOnce:** moves out captured values
- **FnMut:** does not move out captured values, but mutates them
- **Fn:** does not move out values, does not mutate them ("purely" functional)

let's take a look at these traits

example

(using closures, existential types)

Iterators

High Level

```
fn main() {
    (0..5).flat_map(|x| x * 100 .. x * 110)
        .enumerate()
        .filter(|&(i, x)| (i + x) % 3 == 0)
        .for_each(|(i, x)| println!("{i}:{x}"));
}
```

We can use closures and iterators to write "functional style" Rust

Creating Iterators

There are three common methods which can create iterators from a collection:

- **iter()** for immutable references to elements
- **iter_mut()** for mutable references to elements
- **into_iter()** for consuming and iterating over the elements

Common Design Pattern

There is a common pattern for defining iterators in Rust:

1. Define a separate struct to house the iterator (e.g., std::VecDeque::Iter)

2. Implement the Iterator trait for this struct

3. Implement an iter() method to construct an iterator from a value

4. (Implement the IntoIterator trait)

let's take a look at these traits

example

(using closures, existential types)