Closures and Iterators

CAS CS 392: Rust, in Theory and in Practice

February 13, 2025 (Lecture 8)

Outline

Closures

Iterators

Workshop

High Level

Closures are anonymous functions, like lambdas in Python:

```
fn  square   (x : i32) -> i32 { x * x }
let square = |x|              { x * x }
```

The big difference: Closures can *capture* values, like closures in other languages, but his becomes interesting with respect to ownership

Common Example: Higher-Order Functions

We can pass closures into higher-order functions like map and filter:

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

// prints: 1 4 9 16 25
```

Note that map returns a `Map` struct which implements the `Iterator` trait

Common Example: Counter Maker

We can also return closures, but we have to be careful about types:

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

Note the use of existential type in return position, this is one case where this pattern is useful.

Type Inference

For most closures, we don't need to write type annotations:

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

That said, closures must be *monomorphic*:

```
// DOES NOT COMPILE
let example_closure = |x| x;
let s = example_closure(String::from("hello"));
let n = example_closure(5);
```

"Borrow" Inference

The compiler determines if a closure only need to immutably borrow:

```
let mut v = vec![1, 2, 3, 4, 5];
let immutable_borrow = &v;
|| println!("{}", v[0]); // unused closure
println!("{}", immutable_borrow[0]);
v.clear();
```

Same with mutable borrows:

```
// DOES NOT COMPILE
let mut v = vec![1, 2, 3, 4, 5];
let immutable_borrow = &v;
|| v.push(6);
println!("{}", immutable_borrow[0]);
v.clear();
```

Moving/borrowing happens when the closures is defined, not called

Moving Captured Values

We can force ownership of captured values with the `move` keyword:

```
fn mk_counter() -> impl FnMut() -> i32 {
    let mut count = 0;
    return move || { count += 1; count }
}
```

Note that `count` has copy semantics so the value isn't moved out of the closure, but it needs to take ownership in order for count to live longer than the scope of `mk_counter`

We can't selectively move/borrow captured values

Closures and Traits

```
let mut v = vec![1, 2, 3];
let f1 = || v;                         // FnOnce only
let f2 = || v.push(4);                 // Not Fn
let f3 = || println!("{}", v[0]);      // All three
```

Closures are just structures which implement the following traits:

- `FnOnce`: moves out captured values
- `FnMut`: does not move values out captured values, mutably borrows captured values
- `Fn`: does not move values out, immutable borrows captured values ("functional" closures)

`FnOnce` is a supertrait of `FnMut` is a supertrait of `Fn`

Outline

Closures

Iterators

Workshop

High Level

Iterators are a common programming pattern for lazily walking through structured data:

```
let v1 = vec![1, 2, 3];
let v1_iter = v1.iter();
for val in v1_iter {
    println!("Got: {val}");
}
```

Anytime you use a for-loop, you're working with an iterator (More on that in a moment)

The Iterator Trait

Iterators implement a particular trait:

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // methods with default implementations elided
}
```

This trait has a single required method, with a *ton* of derived methods

Associated Types

The iterator trait has an *associated type* `Item`:

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // methods with default implementations elided
}
```

There is a subtle different between associated types and generic traits: *since it's only possible to define a trait once for a type*, there can only be one iterator for a given type (with a fixed `Item`)

Laziness

Laziness here means that any computation associated with the `next` element is delayed until the `next` is called:

```
// Does not print anything
let v = vec![1, 2, 3, 4, 5];
v.iter().map(|x| println!("{x}"));
```

Defining Iterators

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. Derive the `Iterator` trait for this struct
3. Implement an `iter()` method to construct an iterator from a value of the given type

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()`, which is consuming, for iterating over the elements themselves

The IntoIterator Trait

It's also possible to automatically convert types into iterators:

```
pub trait IntoIterator {
    type Item;
    type IntoIter: Iterator<Item = Self::Item>;

    // Required method
    fn into_iter(self) -> Self::IntoIter;
}
```

We can add a last step to the previous slide:

1. Define a separate struct to house the iterator (e.g., `std::VecDeque::Iter`)
2. Derive the `Iterator` trait for this struct
3. Implement an `iter()` method to construct an iterator from a value of the given type
4. **Derive the `IntoIterator` trait for your given type**

Iterators and For Loops

For-loops implicitely call one of the three functions for creating iterators.

```
let values = vec![1, 2, 3, 4, 5];

for x in values { // same as `values.into_iter()`
      println!("{x}");
}

let mut values = vec![41];

for x in &mut values { // same as `values.iter_mut()`
    *x += 1;
}

for x in &values { // same as `values.iter()`
    assert_eq!(*x, 42);
}
```

Adapters

We can use closures and higher order functions to build more complex iterators:

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

This allows for mor functionally-styled code

Outline

Workshop

- Finish Assignment 3 (maybe try to implement colon definitions)
- Define a *Gap Buffer* structure and implement the `IntoIterator` trait for it (this will be a question on Assignment 4)

A Gap Buffer is a buffer-like data structure that allows for fast local updates at a particular postion. They're useful for things like text editors, in which the particular position is the cursor.

The easiest way to build a gap buffer is to store two vectors, where "moving the cursor" means popping from one vector and pushing to another (note that this means one vector may be stored backwards)