LECTURE 19

Theory and Design of PL (CS 538)

April 01, 2020

MORE TRAITS

RUST SYNTAX USES TRAITS

- To use Rust syntax with your types, implement traits
  - Square brackets: Index/IndexMut
  - Dereferencing (star): Deref/DerefMut
  - Operators (+, -, ...): Add/Sub/...
  - For loops: iterators
- See more in std::ops

OUR RUNNING EXAMPLE

- A type of points in 3D: x, y, z

```rust
struct Point3D {
    x: f32,
    y: f32,
    z: f32,
}
```

SQUARE BRACKETS

- Index and IndexMut. Simplified trait definitions:

```
trait Index<IdxType> {
    type Output;
    fn index(&self, idx: IdxType) -> &Self::Output;
}

trait IndexMut<IdxType> {
    type Output;
    fn index(&mut self, idx: IdxType) -> &mut Self::Output;
}
```

- `Output` is an *associated type* of the trait
  - Need to pick a type when implementing trait

IMPLEMENTING IT

```rust
impl Index<char> for Point3D {
    type Output = f32;  // indexing produces floats
    fn index(&self, idx: char) -> &Self::Output {
        match idx {
            'x' => &self.x,
            'y' => &self.y,
            'z' => &self.z,
            _ => panic!("Unknown coordinate!"),
        }
    }
}

impl IndexMut<char> for Point3D { ... }
```

TRYING IT OUT

```
let mut my_point = Point3D { x: 1.0, y: 2.0, z: 3.0 };

println!("x is {}", my_point['x']);  // uses Index

my_point['x'] = 5.0;  // uses IndexMut

println!("x is {}", my_point['x']);  // uses Index
```

OVERLOADING OPERATORS

- Just about all operators have corresponding traits
  - “+”: std::ops::Add
  - “+=”: std::ops::AddAssign
  - “<<”: std::ops::Shl
- Here’s Add (slightly simplified):

```
// Default: RightSideType = Self (same type)
// Can change to add two things of different types together
trait Add<RightSideType = Self> {
    type Output;
    fn add(self, rhs: RightSideType) -> Self::Output;
}
```

IMPLEMENTING IT

```rust
// Add with no type params: RightSideType = Point3D
// In other words: add two Point3D together
impl Add for Point3D {
    type Output = Point3D; // result of adding is a Point3D
    fn add(self, rhs: Point3D) -> Self::Output {
        Point3D {
            x: self.x + rhs.x,
            y: self.y + rhs.y,
            z: self.z + rhs.z,
        }
    }
}
```

TRYING IT OUT

```
// Make two points
let my_point = Point3D { x: 1.0, y: 2.0, z: 3.0 };
let my_point2 = Point3D { x: 3.0, y: 2.0, z: 1.0 };

// Add them up
let my_final = my_point + my_point2;  // (4.0, 4.0, 4.0)
```

DEREFERENCING

- Use * operator to turn data into reference
- Mutable or immutable
  - Usually: get immutable reference (read)
  - Left-side of assignment: mutable ref (write)
- Note: usually don’t get owned values!
  - Refs usually don’t have ownership
  - Special exception: dereferencing Box

DEREF/DEREFMUT TRAITS

- Simplified defs look something like this:

```
trait Deref {
    type Target;  // returned ref will be to this type
    fn deref(&self) -> &Self::Target;
}

trait DerefMut {
    type Target;  // returned ref will be to this type
    fn deref(&mut self) -> &mut Self::Target;
}
```

- We’ll hear much more next time…

ITERATORS

EVERYWHERE IN RUST

- Reading command line args
- Stepping through a file system directory
- Looping through lines in a file
- Handling incoming network connections
- ...

IN A NUTSHELL

- Type that lets you step through a collection
- Many things in Rust can be treated as iterators

```
trait Iterator {
  // Type of item produced
  type Item;

  // Try to get the next item
  fn next(&mut self) -> Option<Self::Item>;
}
```

- `next` returns next item, or nothing if no more
- Hold iterator state in the type implementing Iterator

GETTING AN ITERATOR

- Three typical flavors
  - `into_iter()`: produced owned values
  - `iter()`: produce immutable references
  - `iter_mut()`: produce mutable references

```
impl Blah {
    fn into_iter(self) -> BlahIterOwn { ... }

    fn iter(&self) -> BlahIterRef { ... }

    fn iter_mut(&mut self) -> BlahIterMut { ... }
}
```

CONSUMING ITERATORS

- Consuming iterators yield owned values

```
let v = vec![String::from("Hello"), String::from("World")];
let mut v_iter = v.into_iter();  // must be mut!

// Can get owned Strings out
let hello_string: String = v_iter.next().unwrap();
let world_string: String = v_iter.next().unwrap();

// Can't use v anymore: moved into iterator
```

BORROWING ITERATORS

```
let v = vec![1, 2, 3];
let mut v_iter = v.iter();

assert_eq!(v_iter.next(), Some(&1));
assert_eq!(v_iter.next(), Some(&2));
assert_eq!(v_iter.next(), Some(&3));
assert_eq!(v_iter.next(), None);
```

MUTABLE ITERATORS

```
let mut v = vec![1, 2, 3];
let mut v_iter = v.iter_mut();
let v_ref = v_iter.next().unwrap();

*v_ref = 9;  // mutate underlying vec

assert_eq!(v[0], 9);
```

DANGER...

- How do we know the references are valid?
  1. Get an iterator from a mutable vector
  2. Get a reference from iterator
  3. Delete everything in vector
- What happens to the reference?

RUST REJECTS PROGRAM

- This problem is called *iterator invalidation*

```
let mut my_vec = vec![1, 2, 3];

// Borrow my_vec immutably: fn iter(&self) -> ...
let mut my_iter = my_vec.iter();
let my_next = my_iter.next();

// Borrow my_vec mutably: fn clear(&mut self) -> ...
my_vec.clear();

// Fails: can't take immutable borrow, then mutable borrow
```

FOR LOOPS USE ITERATORS

- Can loop over anything convertible into Iterator

```
let v = vec![1, 2, 3];

for val in v {
    println!("Got: {}", val);
}
```

INTOITERATOR TRAIT

- “This type can be converted into an iterator”
- Trait definition looks something like the following:

```
trait IntoIterator {
    type Item;  // type of Item produced
    type IntoIter;  // type of iterator, needs Iterator trait

    // Turn self into an Iterator
    fn into_iter(self) -> Self::IntoIter;
}
```

FOR LOOPS, DESUGARED

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

for val in IntoIterator::into_iter(v) {
    println!("Got: {}", val);
}

for val in v.into_iter() {
    println!("Got: {}", val);
}

// same idea for &v or &mut v
```

IMPLEMENTING ITERATORS

- Make a struct to hold state of iterator

```
struct Point3DIter {
    it_x: Option<f32>, it_y: Option<f32>, it_z: Option<f32>,
    cur_coord: char,
}

impl Point3DIter {
    fn new(p: Point3D) -> Self {
        Point3DIter {
            it_x: Some(p.x),
            it_y: Some(p.y),
            it_z: Some(p.z),
            cur_coord: 'x' // Initialize cur_coord to 'x'
        }
    }
}
```

ITERATOR TRAIT

```
impl Iterator for Point3DIter {
    type Item = f32;  // iterator produces floats (f32)
    fn next(&mut self) -> Option<Self::Item> {
        match self.cur_coord {
            'x' => { self.cur_coord = 'y'; self.it_x.take() }
            'y' => { self.cur_coord = 'z'; self.it_y.take() }
            'z' => { self.cur_coord = 'a'; self.it_z.take() }
            _ => None
        }
    }
}
```

TESTING IT OUT

- Implement IntoIterator for Point3D

```
impl IntoIterator for Point3D {
    fn into_iter(self) -> Point3DIter {
        Point3DIter::new(self)
    }
}

let my_point = Point3D { x: 1.0, y: 2.0, z: 3.0 };

for val in my_point {
    println!("Coordinate: {}", val);
}
```

CLOSURES IN RUST

REVIEW: HASKELL CLOSURES

- Functions mentioning external variables
- May be anonymous, or named

```
fooEnv x y = let closure = (\a b -> x + y + a + b) in
             ...
```

CLOSURES IN RUST

- Syntax similar, arguments between pipes
  - Don’t need to annotate types (unlike functions)
  - Braces are optional

```
let my_closure       = |arg| arg + 1;

// with type annotations and braces
let my_closure_annot = |arg:i32| -> i32 { arg + 1 };

// two arguments
let my_closure_two   = |foo, bar| foo + bar;

// no arguments, always returns 42
let my_closure_unit  = || 42;
```

UNEXPECTED INTERACTIONS

- Normal Rust functions don't capture context

```
{
  let ext = String::from("foo");
  fn bar(mut arg: String) -> String { arg.push_str(ext) }; // Bad
}
```

- Who owns captured variables (ext) in this closure?

```
{
  let ext = String::from("foo");
  let bar = |mut arg| { arg.push_str(ext) }; // Who owns ext?
}
```

CLOSURE TRAITS

- Rust uses traits to describe capture ownership
  - Only affects variables *mentioned* in body
- Compiler infers which trait to assign a closure
  - Tries to assign the most permissive trait
  - Compiler may need help sometimes

OPTION 1: MOVE

FNONCE TRAIT

- Trait for functions that can be called at most once
- Closures take ownership of captured variables
  - As soon as closure is defined
  - Never returns ownership
- FnOnce closures can be called at most once
  - Can't take ownership multiple times!

EXAMPLE: FNONCE

- Use `move` syntax to make closure FnOnce
  - Only required variables are moved
- Useful when spawning new threads (later)
  - Move everything thread needs into closure

```
let v = vec![1, 2, 3];
let copy_v = vec![1, 2, 3];
let is_equal = move |z| z == v; // Takes ownership of vector

println!("Can't print v: {}", v); // Not OK: v doesn't own

println!("Run closure: {}", is_equal(copy_v)); // OK

println!("Again? {}", is_equal(copy_v));       // Not OK
```

OPTION 2: IMMUTABLE BORROW

FN TRAIT

- Trait for fns that can be called any number of times
- Closures *immutably borrow* captured variables
  - Can’t modify captured variables

```
let env   = 0;
let a_str = "my string";

let simple_closure = |arg| arg + env;
let printf_closure = |arg| println!("strs: {}, {}", arg, a_str);
```

OPTION 3: MUTABLE BORROW

FNMUT TRAIT

- Trait for fns that can be called any number of times
- Closures *mutably borrow* captured variables
  - Can modify captured variables

EXAMPLE: FNMUT

- FnMut is automatically inferred by compiler

```
let mut s = String::new();
println!("Before: {}", s); // Before:

{
    let mut app_s = |arg| s.push_str(arg);
    app_s(" foo");
    app_s(" bar");
}

println!("After: {}", s); // After: foo bar
```