

Advanced Rust Features

Luca Abeni

`luca.abeni@santannapisa.it`

March 23, 2022

Rust Functions and Closures

- Rust makes a difference between *functions* and *closures*
 - Rust functions: blocks of code associated to names, formal parameters and return value
 - Associated to names: denotable entities
 - Can be stored in variables, or returned by functions
 - Cannot capture non-local variables → equivalent to C function pointers
 - Rust closures: functions associated to an environment for non-local variables
 - Again, denotable, can be stored in variables, and can be returned

Functions as Denotable Entities

- Functions are denotable: can be bound to a name
- Functions can be stored in a variable

```
fn main()
{
    fn inc(x: i64) -> i64 {
        x + 1
    }
    let f = inc;
    let v = 5;

    println! ("Inc_{}_=_{}", v, f(v))
}
```

- However, they cannot capture non-local variables!

Functions and Non-Local Variables

- Something like this will not compile:

```
fn main()
{
    let n = 1;
    fn add(x: i64) -> i64 {
        x + n
    }
    let f = inc;
    let v = 5;

    println!("Inc_{}_{}_=", v, f(v))
}
```

- The error says “can't capture dynamic environment in a fn item”...
 - ...And “use the ``|| ... `` closure form instead”
- What does this mean?

Rust Functions are Function Pointers

- A function has a type implementing the “`fn`” trait
- It really is just a function pointer, without additional data
- No associated environment for non-local symbols!
 - This is why the “`n`” variable cannot be used in “`add`”...
- What we need is a real closure (function pointer + associated environment)...
- ...And the compiler seems to suggest some kind of “`|| ...`” syntax!

Closures

- Closure: parameters between “| |”, followed by the body (between “{ }”)

```
fn main()
{
    let n = 1;
    let f = |x| {
        x + n
    };
    let v = 5;

    println! ("Inc_{}_=_{}", v, f(v))
}
```

- Here, “n” is borrowed
- This is not an issue because “f” and “n” have the same lifetime...
- ...But what happens if “f” survives to “n”?

Closures and Non-Local Variables

- This cannot compile, because the closure borrows “n” but is returned (and “n” does not exist outside of the function)

```
fn sum(n: i64) -> impl Fn(i64) -> i64
{
    |x| {
        x + n
    }
}
```

- The relevant error is “borrowed value does not live long enough”
- Side note: “Fn” is the trait implemented by closures, and “impl Fn...” means that the function returns a type implementing the “Fn” trait
- Anyway, how to fix the issue? By moving the value!

Closures Moving Non-Local Variables

- This compiles and works:

```
fn sum(n: i64) -> impl Fn(i64) -> i64
{
    move |x| {
        x + n
    }
}

fn main()
{
    let n = 1;
    let f = sum(n);
    let v = 5;

    println! ("Add_{}_{}_=_{}", v, n, f(v))
}
```

- Other traits for closures: “FnOnce” (move the environment when the closure is invoked) and “FnMut” (borrow mutably the environment)

Rust Threads

- Create a thread with “`std::thread::spawn`”
 - Thread body: closure (**warning**: can capture non-local variables)
 - The thread can survive to captured variables... They must be moved!
 - How to share variables, if we need to move them???
 - Trick similar to “`RefCell`”...
- `spawn()` returns a “`JoinHandle`”
 - Used to wait for the thread termination (invoke its “`join()`” method)

Smart Pointers for Threads

- How to share variables between threads?
- We need to move *cloned* values... Similar to `Rc`!!!
- `Rc` does not work with threads (it is not atomic): use `Arc`!
- But this is not mutable...
- Sharing mutable references: we need something similar to `RefCell`
- `Mutex`: allows to get mutable references (`lock()` method)
- So, we need an “`Arc<Mutex<...>>`” (use `new()` to create both the `Mutex` and the `Arc`)