6 Systems Programming

Thomas Prokosch
November 20, 2018

Outline

Function pointers

What are functions?

What happens here?

```
fn hello() { }
let f = hello;
```

1. Compile error, parentheses missing after `hello` in line `2`.
2. Everything okay, `f` contains the unit value `()`.
3. Everything okay, `f` contains a pointer.
4. Everything okay, function `hello` is moved into `f`.

What are functions?

What happens here?

```
fn hello() { }
let f = hello;
```

3 Everything okay, `f` contains a pointer.

Functions

- are represented by their function pointers
- can be **assigned** to variables
- are **callable addresses**:
  - indicate start of a new stack frame
  - put function arguments on the stack
  - jump to the address of the function and continue execution there

Returning function pointers

```
fn make_plusone() -> fn (usize) -> usize {

    |n| n+1
}
let two = make_plusone()(1);
println!("Result: {}", two);
```

1. Rust is confused about two arrows in line 1.
2. Rust is confused about two sets of parentheses in line 5.
3. The result is 2.
4. The result is something else.

Returning function pointers

```
fn make_plusone() -> fn (usize) -> usize {

    |n| n+1
}
let two = make_plusone()(1);
println!("Result: {}", two);
```

3 The result is 2.

What is the result?

```
fn make_plusone() -> fn (usize) -> usize {
    let a = 1;
    |n| n+a
}
let two = make_plusone()(1);
println!("Result: {}", two);
```

1. Success: The result is 2.
2. Compile-time error: `|n| n+a` does not return a function pointer.
3. Compile-time error because `a` gets dropped at the end of its scope.
4. Run-time error: `a` is dropped, its value undefined, garbage output.

What is the result?

```
fn make_plusone() -> fn (usize) -> usize {
    let a = 1;
    |n| n+a
}
let two = make_plusone()(1);
println!("Result: {}", two);
```

2 Compile-time error: `|n| n+a` does not return a function pointer.

`|n| n+a` uses a value from outer scope → Closure

Closures as input parameters

Closure thought model

```
let a = 1;
let plusa = |n| n+a;
```

can be **mentally expanded** to:

```
struct AnonymousType<'a> {
    a: &'a i32
}

impl<'a> AnonymousType<'a> {
    fn call(&self, n: i32) -> i32 {
        n + self.a
    }
}

let a = 1;
let plusa = AnonymousType{a: &a}; // borrow
```

Properties of closures

Closures contribute heavily to the

- **functional paradigm** of Rust.
  Functions are first-class citizens (Strachey 1966):
  - functions as **arguments** to other functions
  - **returning** functions as values
  - assigning functions to **variables**
  - support for **function literals** (anonymous functions)
  - names of functions treated like variables with function type
- **high-performance goal** of Rust.
  Seemingly complex, closures can be inlined with no overhead — zero cost abstraction

`Fn` Trait

What is the type of `square`? Of `plus_a`?

- ```
  let square = |x| x*x;
  let a = square(2);
  ```

- ```
  let a = 1;
  let plus_a = |x| x+a;
  ```

`Fn` Trait

What is the type of `square`? Of `plus_a`?

- ```
  let square = |x| x*x;
  let a = square(2);
  ```
  → `fn (i32) -> i32`
- ```
  let a = 1;
  let plus_a = |x| x+a;
  ```
  → **unique**, unnamed type

But: We can **characterize** the closure:

```
Fn (i32) -> i32
```

- The `Fn` trait characterizes **both** functions and closures.

Functions and closures as input parameters

```
fn eval<Closure>(c: Closure) -> i32
where Closure: Fn (i32) -> i32 {
    c(3)
}

let a = 5;
let f = |x| x*x;
let c = |x| x+a;
println!("f: {}, c: {}", eval(f), eval(c));
```

```
f: 9, c: 8
```

Storing functions and closures

- **Functions** represented by their **function pointers**
  - fixed size
  - store as-is
  - or represent with a trait (to stay more generic)
- **Closures** represented by **traits**
  - access unknown amounts of non-parameter data
    Would lead to nasty bugs without the borrowing idiom!
  - put behind pointer with `Box<>` (trait objects)

Borrowing and moving with closures

Everything is **borrowed unless** it is consumed (not modified) inside the closure:

```
let a = "Hello".to_string();
let cl = || {
    let b = a;
    println!("You say: {}", b);
};   // no need to call cl()
println!("Simon says: {}", a);   // ERROR
```

Force consumption of values by closure with `move` keyword:

```
let a = "Hello".to_string();
let cl = move || {
    println!("You say: {}", a);
};   // also moved, like above
println!("Simon says: {}", a);   // ERROR
```

Call traits

fn pointers	no values are captured (only parameters, no parameters)
Fn trait	values are captured by reference
FnMut trait	values are captured by mutable reference
FnOnce trait	values are moved, i.e. closure may be called only once

Rule of minimality

$$fn \subseteq Fn \subseteq FnMut \subseteq FnOnce$$

The Rust compiler automatically implements the trait with the

- least requirements that
- still fits the closure.

Rule of minimality

```
fn consume<F>(f: F) where F: FnOnce () { f() }
let mut s = "hello".to_string();

// let a = || {};
// let b = || { println!("b: {}", &s) };
// let c = || { s.push_str(" world") };
// let d = move || { println!("d: {}", &s) };
consume(...);
println!("s: {}", s);
```

Imagine trying to feed variables `a`, `b`, `c`, `d` one by one to `fn consume`. What definitions will succeed?

1. Definitions `a` and `b`.
2. Definitions `a`, `b`, and `c`.
3. Definitions `a`, `b`, `c`, and `d`.
4. None of the combinations above.

Rule of minimality

```
fn consume<F>(f: F) where F: FnOnce () { f() }
let mut s = "hello".to_string();

// let a = || {};
// let b = || { println!("b: {}", &s) };
// let c = || { s.push_str(" world") };
// let d = move || { println!("d: {}", &s) };
consume(...);
println!("s: {}", s);
```

Imagine trying to feed variables `a`, `b`, `c`, `d` one by one to `fn consume`. What definitions will succeed?

1. Definitions `a` and `b`. `push_str` **mutates**, `move` **takes ownership**

Definition and call of `fn consume` can only further limit the options.

Higher-Ranked Trait Bounds (HRTB)

Lifetimes revisited

Like traits, **lifetimes** are

- specified by the **programmer** (even if implicit)
- **monomorphized** (one concrete lifetime is used)
- not automatically adapting to "circumstances"
  Lifetimes do not change because of function behavior.

```
fn len<'a>(s: &'a str) -> usize { s.len() }

fn main() {
   let t: String = "Higher-Ranked Trait Bounds".to_string();
   let l = len(&t);   // lifetime of &t used as param to len
   println!("{} has {} characters.", t, l);
}
```

References are provided by the caller.

⌄

Lifetime is provided by the caller.

Traits with references

With what lifetime bounds could the program compile?

```
trait Work<'a> { fn work(&self, i: &'a i32); }

fn foo<'a, T>(b: T) where T: Work<'a> {
    let x: i32 = 10;
    b.work(&x);   // ERROR: 'a outlives x
}
```

We cannot

- create a **reference locally** and
- **pass lifetime** bound to the caller.

Solution: Separate lifetime of trait from method call site.

Higher-Ranked Trait Bounds (HRTB)

- Trait will be instantiated by the caller.
- Lifetime stays with the callee.

Syntax of Higher-Ranked Trait Bounds

Higher-Ranked Trait Bounds are introduced with

```
for<>
```

```
trait Work<'a> { fn work(&self, i: &'a i32); }

fn foo<T>(b: T) where T: for<'a> Work<'a> {
    let x: i32 = 10;
    b.work(&x);   // OK
}
```

With **Higher-Ranked Trait Bounds (HRTB)**, trait bounds are **ranked higher** than lifetimes: The trait bound is valid for all lifetimes.

Comparison: Regular trait bounds/HRTB

```
trait Work<’a> { fn work(&self, i: &’a i32); }

fn regular<’a>(_w: Box<        Work<&’a>>) {}
fn hrtb       (_w: Box<for<’a> Work<&’a>>) {}
```

Lifetime of `i32` reference is

- without HRTB: determined by the **caller**.
- with HRTB: not part of the function signature but determined by the **called function**.

Example HRTB

```
impl<T> Option<T> {
    fn filter<F>(self, f: F) -> Option<T>
    where F: FnOnce(&T) -> bool {
        match self {
            Some(value) if f(&value) => Some(value),
            None => None
        }
    } }
```

How to expand the above function signature to include lifetimes?

1. `fn filter<'a, F>(self, f:  F) -> Option<T>`
   `where F: FnOnce(&'a T) -> bool`
2. `fn filter<F>(self, f:  F) -> Option<T>`
   `where F: for<'a> FnOnce(&'a T) -> bool`
3. `fn filter<F>(&'a self, f:  F) -> Option<T>`
   `where for<'a> F: FnOnce(&'a T) -> bool`
4. None of the above.

Example HRTB

```
impl<T> Option<T> {
    fn filter<F>(self, f: F) -> Option<T>
    where F: FnOnce(&T) -> bool {
        match self {
            Some(value) if f(&value) => Some(value),
            None => None
        }
    } }
```

How to expand the above function signature to include lifetimes?

2
```
fn filter<F>(self, f:  F) -> Option<T>
where F: for<'a> FnOnce(&'a T) -> bool
```

Smart Pointers

Memory management

Approaches for **allocating/freeing heap memory**

manual approach — C/C++
: programmer has to be **aware of pointer usage**

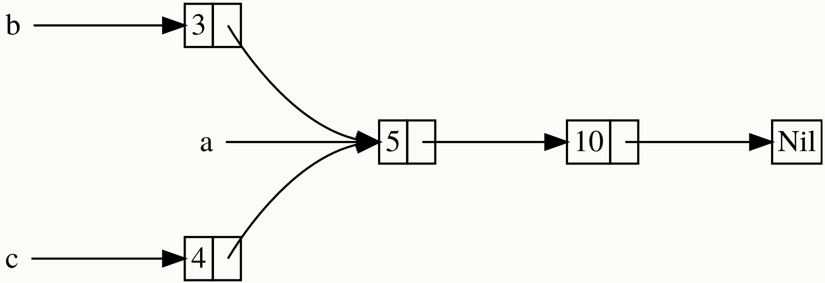
garbage collection — Java, Haskell, etc.
: regular checks whether memory is still needed, always a **heuristic**, **slows** program down, uses **additional memory**

ownership system — Rust
: memory freed when owner goes out of scope,
`Box<>` **owns** its data: **no sharing** → next slide

reference counting
: **count** pointers pointing to a memory location,
is **problematic with cycles**

Shared data

We aim to represent the following data:



source: The Rust Programming Language (2018 edition, chapter 15.4)

Common pattern, for example in

- graphs,
- doubly linked lists,
- ring buffers, etc.

Naive approach to shared data

```
enum List { Cons(i32, Box<List>), Nil }
use  List::{Cons, Nil};

let a = Cons(5,
             Box::new(Cons(10,
                           Box::new(Nil))));
let b = Cons(3, Box::new(a)); // ERROR: value moved here
let c = Cons(4, Box::new(a)); // ERROR: used here after move
```

Solutions:

- Put **references** into list.
  Where to put the original data?
- **Clone** list.
  What if I need to update values? Memory consumption? Speed?
- Use reference counting.

Reference counting with `rc::Rc<T>`

Reference counting

- uses pointers
- good for **non-cyclic** data
- **allocate** explicitly: `fn std::rc::Rc<T>::new(T)`
- **increment** reference count when calling `fn std::rc::Rc<T>::clone()`
- **deallocate** after last use: reference count **dropped to zero**

Shared data with `rc::Rc<T>`

```
enum List { Cons(i32, Rc<List>), Nil }
use  List::{Cons, Nil};
use  std::rc::Rc;

let a = Rc::new(Cons(5,
                    Rc::new(Cons(10,
                                 Rc::new(Nil)))));
let b = Cons(3, Rc::clone(&a));
let c = Cons(4, Rc::clone(&a));
// now, a may go out of scope, leaving b, c intact
```

- `Rc<T>::new()` is used instead of `Box<T>::new()`
- `Rc<T>::clone(&self)` is relatively cheap, increments reference count
- `Rc<T>` type is light-weight but **not thread-safe** → atomic operation

Atomically Reference Counted: `std::sync::Arc<T>`

- functionally equivalent to `rc::Rc<T>`
- uses atomic operation for incrementing reference count, higher overhead
- `Rc<T>` is not `Send`
- `Arc<T>` is `Send` (if `T` is `Send`)

```
use std::sync::Arc;

let xlii = Arc::new(42);
for _ in 0..10 {
    let xlii = Arc::clone(&xlii);

    std::thread::spawn(move || {
        // ^ compiler error if xlii: Rc<i32>
        println!("{:?}", xlii);
    });
}
```

Interior mutability

Issues with `rc::Rc<T>`

```
use std::rc::Rc;

let a: Vec<_> = vec![Rc::new("Hello".to_string())];
let mut b = a.clone();
b[0].push_str(", Rust!");
```

results in:

```
error[E0596]: cannot borrow immutable borrowed content
  |           as mutable
6 | b[0].push_str(", Rust!");
  | ^^^^ cannot borrow as mutable
```

Attempt to gain mutability with `rc::Rc<T>`

```
use std::rc::Rc;

let mut s                              = "Hello".to_string();
let mut rc                             = Rc::new(&mut s);
let mut v: Vec<&mut Rc<&mut String>> = vec![&mut rc];
let mut rc2:   &mut Rc<&mut String>  = &mut v[0].clone();
let mut s2                             = Rc::get_mut(rc2);
s2.unwrap().push_str(", Rust!");
```

Awful code, and yet:

```
thread 'main' panicked at 'called `Option::unwrap()`
on a `None` value'
```

`Rc` does **not provide mutability** if values are shared between multiple parties.
Solution: Interior mutability.

What is interior mutability?

Interior mutability:

- **Mutate data despite immutable references to it.**
- Borrow checker is sidestepped.
- Run-time checks necessary. Slower!

Use cases:

- data sharing with reference counting
- evaluation cache, example: memoization
- graphs

std::rc::RefCell<T>

	owner(s)	borrows	checked	thread-safe?
Box<T>	single	mut. + immut.	compile-time	yes
Rc<T>	multiple	immutable	run-time	no
RefCell<T>	single	mut. + immut.	run-time	no

```
use std::rc::Rc;
use std::cell::RefCell;

let a: Vec<_> =
    vec![Rc::new(RefCell::new("Hello".to_string()))];
let mut b = a.clone();
b[0].borrow_mut().push_str(", Rust!");
println!("{:?}", a);
```

```
[RefCell { value: "Hello, Rust!" }]
```

Foreign Function Interface (FFI)

Unsafe Rust

The `unsafe` keyword **enables new features** in Rust.
The **programmer is responsible** for not breaking Rust **invariants**!

Keyword `unsafe` is used:

1. in code that uses the Foreign Function Interface (FFI)
2. when working with **raw pointers** (access to OS, hardware)
3. when interior mutability does not suffice

Two variants:

- unsafe blocks: `unsafe { ... }`
- unsafe functions: `unsafe fn dangerous() { ... }`
  Function **definition** and function **call** need to be marked unsafe.

Unsafe functions can only be called in unsafe blocks.

Raw pointers

- **constant** `*const T` and **mutable** `*mut T` raw pointers
- asterisk is part of the type name **and** used for dereferencing
- **no** safety or liveness **guarantees** (borrowing rules not enforced)
- no automatic cleanup (implement Drop trait)
- dereferencing a raw pointer is `unsafe` (pointers may be invalid)

```
let mut num = 5;
let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    *r2 = 2;
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

- keyword `as` enables type casting between two types

Calling C functions from Rust

1. Create C files (example: `clib.c`) with functions
2. Create `build.rs` **script** to create library `libclib.a` from C code (see next slide)
3. Adapt `Cargo.toml` to **name build script and library** for linking with Rust code

```
[package]
...
links = "clib"
build = "build.rs"

[dependencies]
libc = "0.2.43"
```

4. Declare C **functions in Rust** with keyword `extern` and proper types (see later)
5. Use `#[link(name = "clib")]` attribute on `extern` block to **connect functions to library name**

Build script build.rs for building C code

```
use std::process::Command;
use std::path::Path;

fn main() {
   let out_dir = std::env::var("OUT_DIR").unwrap();

   Command::new("gcc").args(&["clib.c", "-c", "-fPIC", "-o"])
                      .arg(&format!("{}/clib.o", out_dir))
                      .status().expect("GCC failed");
   Command::new("ar" ).args(&["crus", "libclib.a", "clib.o"])
                      .current_dir(&Path::new(&out_dir))
                      .status().expect("Linker ar failed");

   println!("cargo:rustc-link-search=native={}", out_dir);
   println!("cargo:rustc-link-lib=static=clib");
}
```

Declaring external functions in Rust

- Include crate `libc` for **common data types**
- Indicate **which library to use** with the `#[link(name = "...")]` attribute
- Specify `extern "C"` to convey **call convention** to the linker

```
extern crate libc;
use libc::{c_void, size_t, uint64_t};

#[link(name = "clib")]
extern "C" {
    fn heapalloc(size: size_t) -> *mut c_void;
    fn heapfree(ptr: *mut c_void);
    fn rdtsc() -> uint64_t;
}
```

The Intel x86/x64 instruction RDTSC (ReaD Time-Stamp Counter) copies the clock cycles since bootup into the EDX:EAX registers.

Use externally declared functions in Rust

```
struct Mem<'a>(&'a mut [u8]);

impl<'a> Mem<'a> { fn new(size: usize) -> Mem<'a> {
    unsafe {
        let ptr = heapalloc(size as size_t);
        Mem(std::slice::from_raw_parts_mut
            (ptr as *mut u8, size)) }}}

impl<'a> Drop for Mem<'a> { fn drop(&mut self) {
    unsafe { heapfree(self.0 as *mut _ as *mut c_void); }}}

fn main() {
    let m = Mem::new(16);
    for b in m.0.iter() {
        print!("{:02x} ", b); } println!(); }
```

Noteworthy documentation

- The Embedded Rust Book
  Rust programming on bare metal systems, such as micro-controllers
- The Rust Discovery Book
  Registers, timers, serial communication, Bluetooth, I2C with Rust

Summary

Today's goals:

1. Understand function pointers, closures, and their traits.
2. Become more acquainted with pointers.
3. Get our feet wet with threads.
4. Recognize that programming languages are tools that become more powerful if they provide loopholes to escape their ideology.
5. Be able to implement a Rust binding to a C library.

We have learned about…

- how Rust implements functions and closures
- functions as first-class citizens
- higher-ranked trait bounds (HRTB)
- memory management with smart pointers, interior mutability
- the foreign function interface (FFI)