COMP 737011 - Memory Safety and Programming Language Design

Lecture 6: Rust Ownership

Hui Xu

xuh@fudan.edu.cn

Outline

- 1. Ownership
- 2. RAII and Lifetime
- 3. Unsafe Code

1. Ownership

Motivation of Design

- Dangling pointer is unacceptable.
- Causal of dangling pointer?
  - False reclaim: manual heap management with malloc/free
  - Automatic reclaim?
    - Static analysis is impossible because alias analysis is NP-hard
    - Shared pointer or garbage collection is inefficient
- Rust comes to rescue.
  - Static analysis without NP-hard problems.
  - How? Ownership-based heap management.

Overall Idea of Ownership

- Each object is owned by one variable.
  - Recall C++ unique pointer
- Ownership can be borrowed in two modes:
  - immutable: read only
  - mutable: read/write
- Exclusive mutability principle: two variables should not share mutable access to the same object at one program point.

Ownership & Borrowing

- Borrowed ownership will be returned automatically if no longer used.

```
let mut alice = Box::new(1);
let bob = alice;
println!("bob:{}", bob);
println!("alice:{}", alice);
```

alice owns the object

alice transfers the ownership to bob;

```
let mut alice = Box::new(1);
let bob = &alice;
println!("bob:{}", bob);
println!("alice:{}", alice);
```

bob borrows the ownership

bob returns the ownership to alice automatically

Violating Exclusive Mutability

```
let mut alice = 1;
let bob = &mut alice;
println!("bob:{}", bob);
println!("alice:{}", alice);
```

mutable borrow

bob returns the ownership

```
let mut alice = 1;
let bob = &mut alice;
println!("alice:{}", alice);
println!("bob:{}", bob);
```

violate exclusive mutability

Move Operator (=)

- If a type is not Copy (trait), move transfers the ownership.
  - e.g., Box<T> is not copy
- If a type is Copy, move does not transfer the ownership and only copies the value.

```
let mut alice = 1;
let bob = alice;
println!("bob:{}", bob);
println!("alice:{}", alice);
```

alice owns the object

copy: duplicate the object

Which Types Can be Copy?

- Primitive types on stack.
- Compound types with all fields implementing Copy.
- How to (deep) copy objects of other non-Copy types?
  - By implementing the Clone trait.
  - Each Copy type is also Clone.

```
let mut alice = Box::new(1);
let bob = alice.clone();
println!("bob:{}", bob);
println!("alice:{}", alice);
```

alice owns the object

clone the object for bob

Mutability

mutable object

```
let mut alice = 1;
alice+=1;
```

```
let alice = 1;
alice+=1;
```

mutable borrow

```
let mut alice = 1;
let bob = &mut alice;
*bob+=1;
```

```
let mut alice = 1;
let bob = &alice;
*bob+=1;
```

```
let alice = 1;
let bob = &mut alice;
*bob+=1;
```

Mutability cont'd

```
let mut alice = 1;
let mut carol = 1;
let mut bob = &mut alice;
*bob+=1;
bob = &mut carol;
*bob+=1;
```

mutable object bob + mutable borrow

```
let mut alice = 1;
let mut carol = 1;
let bob = &mut alice;
*bob+=1;
bob = &mut carol;
*bob+=1;
```

immutable object bob + mutable borrow

bob cannot be modified

Why Ownership is Effective for Static Analysis

- Restrict the complexity of the alias analysis problem.
  - There is only one mutable pointer at each program point.
  - Only mutable pointers can lead to dangling pointers.
  - We only need to trace the mutable pointer for each object.
    - e.g., via a stack-based approach.

Pros and Cons

- Benefit: compile-time method, no runtime cost
- Cons: when shared mutability is a Must…
  - Such as double linked lists?
  - Alternatives: shared pointer or unsafe code.

2. RAII and Lifetime

RAII: Resource Acquisition is Initialization

Idea of RAII

- Tie resources to lifetime.
- Each object is allocated and initialized once created.
  - all pointers refer to particular objects
  - no raw or dangling pointers
  - no uninitialized memory
- Deallocation automatically when the object is dead.
  - no manual deallocation is needed
  - achieved through static lifetime inference

Lifetime

- Each object has a lifetime constraint.
- The object is reclaimed automatically after death.
- A variable cannot borrow an object with a shorter lifetime.

```
let alice;
{
    let bob = 5;
    alice = &bob;
}
println!("alice:{}", alice);
```

bob lives in this subscope

alice points to an expired object

Review Move for Non-Copy Types

- Expand the lifetime of the object on heap

```
let alice;
{
    let bob = Box::new(1);
    alice = bob;
}
println!("alice:{}", alice);
```

transfer the ownership to alice

```
fn testret() -> Box<u64>{
    Box::new(1)
}

let r = testret();
println!("return:{}", r);
```

move the ownership to the ret value

Lifetime Declaration

- Lifetime should be declared during function declaration

```
fn longer<'a>(x:&'a String, y:&'a String)->&'a String{
    if x.len()>y.len(){
        x
    } else {
        y
    }
}
fn stringcmp(){
    let str1 = String::from("alice");
    let str2 = String::from("bob111");
    let result = longer(&str1, &str2);
    println!("The longer string is {}", result);
}
```

Partial Order of Lifetime

- <'a: 'b, 'b> means a is relatively larger than b

```
fn longer<'a:'b,'b>(x:&'a String, y:&'b String)->&'b String{
    if x.len()>y.len(){
        x
    } else {
        y
    }
}
fn stringcmp(){
    let str1 = String::from("alice");
    let result;
    //{
        let str2 = String::from("bob111");
        result = longer(&str1, &str2);
    //}
    println!("The longer string is {}", result);
}
```

The return value cannot be 'a. Why?

Non-lexical Lifetime

- The default mode is non-lexical unless necessary.
- Rust compiler tries to minimize the lifespan.

```
'a: { let str1 = "alice";
    'b: { let str2 = "bob";
        'c: { let result = longer(str1,str2);
              println!("The longer string is {}", result);
        }
    }
}
```

Lifetime Elision to Be More Ergonomic

- Lifetime declaration can be elided if
  - there is only one input lifetime position
  - there are multiple positions, but one is &self or &mut self
    - assign the lifetime of self to elided output lifetimes

```
fn foo<'a>(s: &'a str, until: usize) -> &'a str;
```

```
fn foo(s: &str, until: usize) -> &str;
```

```
fn foo(s: &str, t: &str) -> &str; // illegal
```

```
fn foo<'a, 'b>(&'a mut self, t: &'b str) -> &str;
```

```
fn foo(&mut self, t: &str) -> &str;
```

https://doc.rust-lang.org/nomicon/lifetime-elision.html

More About Lifetime

- A static object means it lives for the entire lifetime.
  - use the reserved lifetime 'static.
  - all strings are static by default.
- The lifetime can be unbounded.
  - Similar to static but more flexible during type check

```
let s: &str = "hello world";
```

equivalent to

```
let s: &'static str = "hello world";
```

```
fn foo<'a>(s: *const String) -> &'a str;
```

unbounded lifetime

https://doc.rust-lang.org/nomicon/unbounded-lifetimes.html

Automatic Reclaim/Drop

- Objects of Copy type (stack) can be reclaimed automatically.
- For other objects with heap data?
  - Drop (trait) unused objects by calling the destructor.
  - Recursively call the destructor of each field.
- Drop and Copy are exclusive in Rust.
  - Box<T> has the Drop trait, but not Copy

```
struct MyType {a:u8, b:Box<u64>}

impl Drop for MyType {
    fn drop(&mut self){
        println!("dropping MyType object...");
    }
}
fn testdrop(){
    let v = MyType {a:1, b:Box::new(2)};
}
```

automatically executed when the lifetime ends

Option<T> for Uninitialized Objects

- Option: an enumerate type
  - Some(T): the object type
  - None: if the object is uninitialized (null pointer)

```
pub enum Option<T> {
    None,
    Some(T),
}

let v = Some(…)
match v.next {
    Some(n) => …,
    None => panic!(),
}
```

Example with a Singly-linked List

```
struct List {
    val: u64,
    next: Option<Box<List>>,
}
impl List {
    fn new(val: u64) -> Self {
        List { val, next: None }
    }
    fn prepend(self, val: u64) -> Self {
        List {
            val,
            next: Some(Box::new(self)),
        }
    }
    fn append(&mut self, val: u64) {
        let mut current = self;
        while let Some(ref mut next) = current.next {
            current = next;
        }
        current.next = Some(Box::new(List::new(val)));
    }
}
```

RAII for Thread Panic

- In a multi-threaded application, what happens when one thread exit exceptionally ?
  - abort: directly terminate the thread
  - panic: perform stack unwinding before exit
- Importance of RAII during stack unwinding
  - release locks (mutex)
  - release opened file descriptors
  - release allocated memories on heap

Sample Multi-thread Program

- When a spawned thread panics, unwind its stack.
- The main thread continues execution.
- Ineffective for fatal errors: *e.g.,* stack overflow

```
let handle = thread::spawn(|| {
    for i in 0..5 {
        println!("new thread print {}", i);
        thread::sleep(Duration::from_millis(10));
    }
    panic!();
    //recursive();
});
for i in 0..10 {
    println!("main thread print {}", i);
    thread::sleep(Duration::from_millis(10));
}
handle.join();
```

3. Unsafe Rust

Problem for Double-Linked List

- A node is owned by both its prev and next node.
- Violate exclusive mutability.


```
struct Node {
    val: u64,
    prev: Option<Box<Node>>,
    next: Option<Box<Node>>,
}
```

Unsafe Code

- Operations that may lead to undefined behaviors are unsafe.
  - dereference raw pointers
  - call unsafe functions
  - call functions of foreign language (FFI)
- Unsafe code can only be used with the unsafe marker.

```
let mut num = 5;
let r1 = &num as *const i32;
println!("r1 is: {}", unsafe { *r1 });
```

r1 is a raw pointer

raw pointer dereference

Dereference raw pointers

```
unsafe fn foo() {
    let addr = 0x012345usize;
    let r = addr as *const i32;
}
unsafe { foo(); }
```

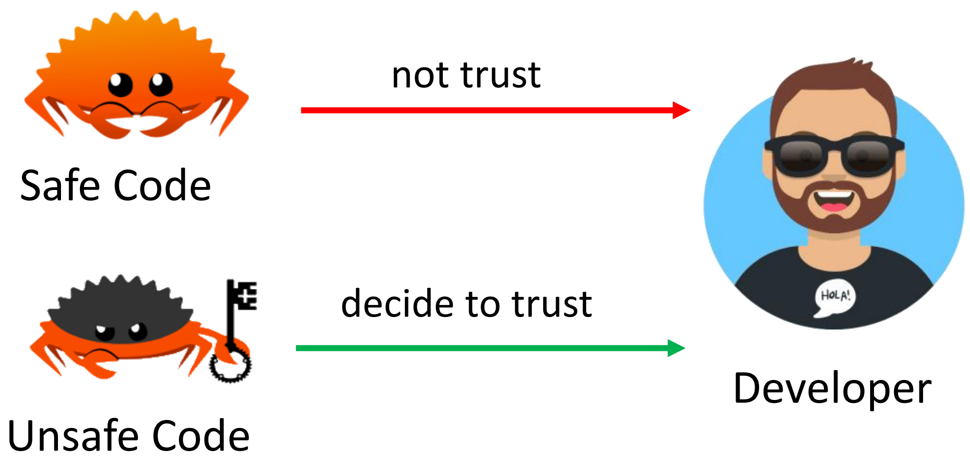
define an unsafe function

call the unsafe function

Call unsafe functions

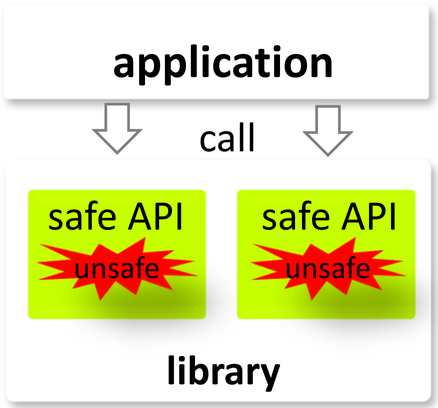
Trust Model

- Rust does not trust developers, so only safe code is allowed.
- If a developer declares he knows the risk, Rust will trust him.



Principle of Rust

- Safe APIs should not incur undefined behaviors.
- Interior unsafe: wrap unsafe code into safe APIs.
- Avoid using unsafe code unless necessary.



Unsafe Version with Raw Pointers

- The objects pointed by raw pointers are not owned.
- The resource may not be dropped automatically.
- It is also prone to dangling pointers (not RAII).

```
struct Node {
    val: u64,
    next: *mut List,
    prev: *mut List,
}
impl Node {
    fn new(val: u64) -> *mut Node {
        Box::into_raw(Box::new(Node { val,
            next: ptr::null_mut(), prev: ptr::null_mut() }))
    }
    unsafe fn prepend(head: *mut List, val: u64) -> *mut List {
        let new_node = List::new(val);
        if !head.is_null() {
            (*new_node).prev = head;
            (*head).next = new_node;
        }
        new_node
    }
}
```

Solution Hint with RC<T> and Weak<T>

- RC<T>/Weak<T>: single-thread reference-counting pointer
  - enable shared mutable aliases
- RefCell<T>: enable mutable access

```
struct Node {
    val: u64,
    prev: Option<Weak<RefCell<Node>>>,
    next: Option<Rc<RefCell<Node>>>,
}
```

Only RC<T> is not Enough

- RC<T> provides interior mutability via get_mut()
  - if counter = 1 (consider both shared and weak)
  - if cloned, get_mut() returns None during runtime

```
let mut x = Rc::new(1);
let _w: Weak<_> = Rc::downgrade(&x);
println!("a strong count: {:?}, weak count: {:?}",
        Rc::strong_count(&x), Rc::weak_count(&x));
let t1 = Rc::get_mut(&mut x).unwrap();
//*t1 = 2;
//assert_eq!(*x, 2);
```

none

RefCell<T>

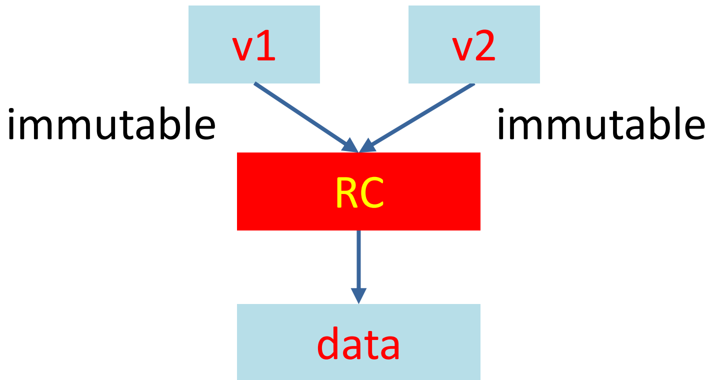
- A mutable memory location.
- Perform borrow check during runtime.

```
let x = RefCell::new(1);
{
    let mut y = x.borrow_mut();
    //let z = x.borrow_mut();
    *y = 2;
}
assert_eq!(2, *x.borrow());
```

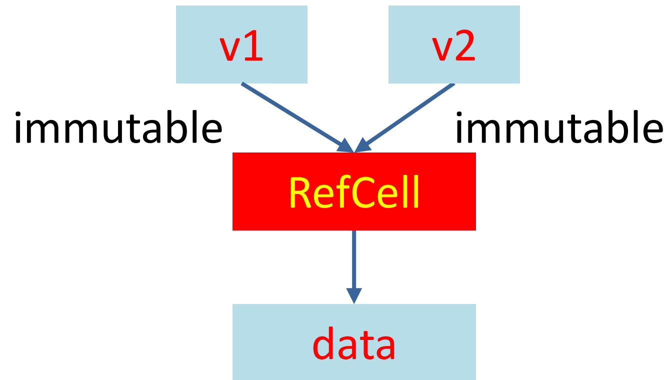
panic during runtime

RC vs RefCell

Compile Time

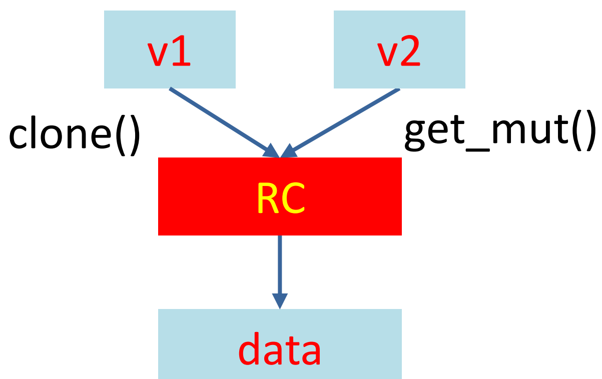


Valid

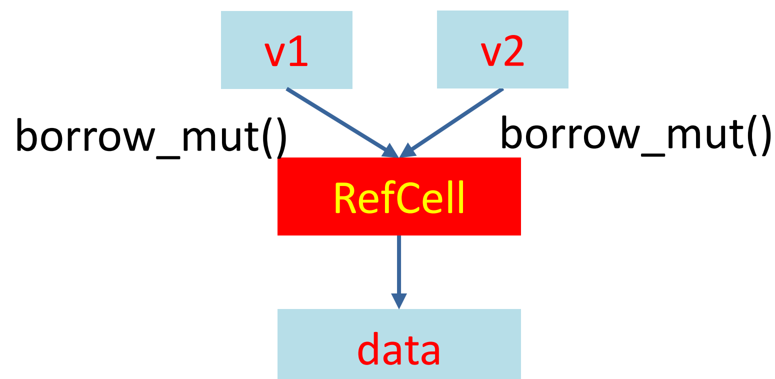


Valid

Runtime



get_mut() may return None if cloned



the second borrow_mut() triggers panic

In-class Practice

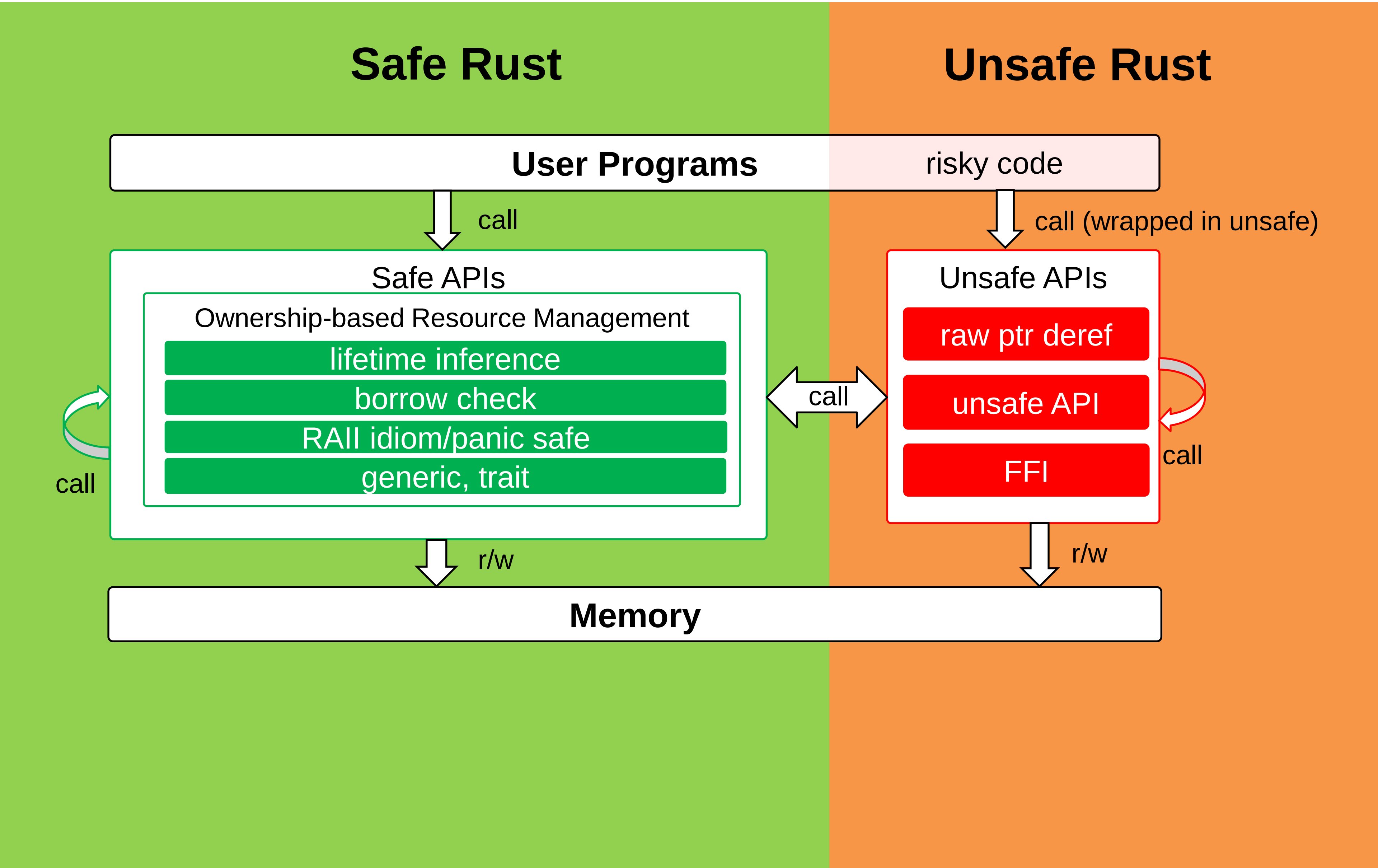
- Option 1: Implement a double linked list with Safe Rust
  - new
  - insert
  - delete
  - search
- Option 2: Implement a binary search tree with Safe Rust
  - new
  - insert
  - delete
  - search

Reference

- https://doc.rust-lang.org/book/
- https://doc.rust-lang.org/stable/nomicon/

Backup Slides

Rust Overview



Cell

- Perform borrow check during compile time;
- Check whether the content is Copy when calling get()

```
fn testcell(){
    let mut x = Cell::new(1);
    //let mut x = Cell::new(Box::new(1));
    //x.set(2);
    {
        let mut y = x.get_mut();
        //let z = x.get();
        *y = 2;
    }
    assert_eq!(2, x.get());
}
```

doesn’t work

compile error

get() copies the value