

Introduction to Rust (2/2)

Jacques-Henri Jourdan

February 9th, 2022

Unsafe Rust

Unsafe and aliasing
Safe abstractions

Interior mutability

Cell<T>
RefCell<T>
Rc<T>

Abstract from last week

Last week, we have seen many features of the Rust programming languages:

- algebraic data types, records;
- various kinds of pointers: `Box<T>`, `&mut T`, `&T`;
- how Rust controls ownership and aliasing, the concept of lifetime;
- traits: Rust's type classes
 - trait examples: iterators, closures
- why all that gives control, performance **and** safety
 - the notion of zero-cost abstraction

This week

Introduction to
Rust (2/2)

Jacques-Henri
Jourdan

Unsafe Rust

Unsafe and aliasing
Safe abstractions

Interior mutability

Cell<T>
RefCell<T>
Rc<T>

- How to escape the ownership discipline when needed?
 - unsafe code behind safe abstractions
 - interior mutability
- Concurrency
 - concurrency primitives and data structures with safety in Rust

Table of Contents

Introduction to
Rust (2/2)

Jacques-Henri
Jourdan

Unsafe Rust

Unsafe and aliasing
Safe abstractions

Interior mutability

Cell<T>
RefCell<T>
Rc<T>

1 Unsafe Rust

- Unsafe and aliasing
- Safe abstractions

2 Interior mutability

- Cell<T>
- RefCell<T>
- Rc<T>

Unsafe Rust

Unsafe and aliasing
Safe abstractions

Interior mutability

Cell<T>
RefCell<T>
Rc<T>

Unsafe

Despite zero-cost abstractions, the requirement of safety has some costs:

- no loop in memory graph
- shared ownership: restricted to borrows
- linking to external libraries: do not always follow ownership discipline
- bounds checks for `Vec` accesses

One can use **unsafe Rust** to avoid these costs.

- A set of features that extend Rust
- Only available in `unsafe` blocks or `unsafe` functions
- No guarantee of safety
 - One should **encapsulate unsafety behind safe abstractions**,
 - or mark functions with unsafe behavior as `unsafe`.

Unsafe features

Introduction to
Rust (2/2)

Jacques-Henri
Jourdan

Unsafe Rust

Unsafe and aliasing
Safe abstractions

Interior mutability

Cell<T>
RefCell<T>
Rc<T>

- dereference **raw pointers**
- call unsafe functions (e.g., accessing `Vec<T>` without bounds checks)
- implement `unsafe` traits (e.g., `Send` and `Sync`, see later)
- mutate global variables (called `static` variables in Rust)
 - (because that can lead to data races)
- access `union` types

Unsafe Rust

Unsafe and aliasing
Safe abstractions

Interior mutability

Cell<T>
RefCell<T>
Rc<T>

Undefined behaviors

Using these features is **unsafe**.

⇒ the compiled program can crash even if type-checked!

It is important to know when a program triggers **undefined behavior**.

This is sometime **very subtle**.

Rust provides an experimental **reference interpreter**, called **Miri**, which detects **some** undefined behavior.

- Can be used on concrete code for testing.

Still, writing unsafe code should be **reserved to experts**. There is a book dedicated to writing unsafe code: **Rustonomicon**.

So I will not teach this here. Instead, we will see why this is subtle.

Unsafe and aliasing

Common use of unsafe code: weaken aliasing restrictions.

Raw pointers `*mut T` and `*const T`:

- have no statically checked aliasing restriction
- can coerce from/to borrows (both shared and unique)
- can easily be used to break any aliasing policy

Unsafe and aliasing

Common use of unsafe code: weaken aliasing restrictions.

Raw pointers `*mut T` and `*const T`:

- have no statically checked aliasing restriction
- can coerce from/to borrows (both shared and unique)
- can easily be used to break any aliasing policy

But the compiler may assume aliasing properties on borrows to perform some optimizations.

```
fn test_noalias(x: &mut i32, y: &mut i32) -> i32 {  
    // x, y cannot alias: they are unique borrows  
    *x = 42;  
    *y = 37;  
    return *x; // must return 42 -- can be optimized  
}
```

Unsafe and aliasing

Common use of unsafe code: weaken aliasing restrictions.

Raw pointers `*mut T` and `*const T`:

- have no statically checked aliasing restriction
- can coerce from/to borrows (both shared and unique)
- can easily be used to break any aliasing policy

But the compiler may assume aliasing properties on borrows to perform some optimizations.

```
fn test_unique(x: &mut i32) -> i32 {  
    *x = 42;  
    // unknown_function cannot have an alias to x  
    unknown_function();  
    return *x; // must return 42 -- can be optimized  
}
```

Unsafe and aliasing

Common use of unsafe code: weaken aliasing restrictions.

Raw pointers `*mut T` and `*const T`:

- have no statically checked aliasing restriction
- can coerce from/to borrows (both shared and unique)
- can easily be used to break any aliasing policy

But the compiler may assume aliasing properties on borrows to perform some optimizations.

```
fn test_shared(x: &i32) -> i32 {  
    let y = *x;  
    // unknown_function cannot have an alias to x  
    unknown_function();  
    return *x + y; // can be optimized to 2*y  
}
```

Undefined behavior and aliasing

Some rules are needed to tell what one can do with raw pointers.

These rules must be a balance between:

- flexibility for the programmer of unsafe code;
- allowing optimizations.

Choosing these rules is still an [open problem](#).

The Miri interpreter implements a set of rules called [Stacked Borrows](#):

- experimental and imperfect,
- but executable on concrete tests.

Undefined behavior and aliasing

Introduction to
Rust (2/2)

Jacques-Henri
Jourdan

Some rules are needed to tell what one can do with raw pointers.

These rules must be a balance between:

- flexibility for users
- allowing operations

Choosing these rules
The Miri interpreter

If you write unsafe code, you need to follow rules like these.

I told you, writing correct unsafe code is subtle...

- experimental and imperfect,
- but executable on concrete tests.

Unsafe Rust

Unsafe and aliasing

Safe abstractions

Interior mutability

Cell<T>

RefCell<T>

Rc<T>

Unsafe Rust

Unsafe and aliasing

Safe abstractions

Interior mutability

Cell<T>

RefCell<T>

Rc<T>

How to benefit from the power of unsafe code without paying the cost?

Use **libraries** written using unsafe features, but with safe interfaces.

Example: `Vec<T>`, resizable arrays:

- fully written in Rust with unsafe features,
- yet, most of the functions exposed by `Vec<T>` are safe!

Example of a safe abstraction: a queue based on a linked list

Let's say we would like to implement a FIFO queue using a singly linked list.

We need a pointer both at the beginning (for `pop`) and at the end of the list (for `push`).

Aliasing rules are violated, we need unsafe code.

Example of safe abstraction: Queue<T>

```
mod queue {  
    pub struct Queue<T> {  
        head: *mut Node<T>,  
        tail: *mut Node<T>  
    }  
    struct Node<T> {  
        elem: T,  
        next: *mut Node<T>,  
    }  
    ...  
}  
use queue::*;

fn (q: Queue<i32> /* Allowed */ {  
    ...  
    q.head /* Error */  
    ...  
    Queue { ... } /* Error */  
    ...  
    let x : Node<i32> /* Error */ = ... ;  
    ...  
}
```

We use **modules** as an encapsulation mechanism.

Some elements of the modules are marked with **pub**, they are public.

The other elements are private.

Fields of **struct** are also either public or private.

- Queue has no public field: **abstract type!**

Table of Contents

Introduction to
Rust (2/2)

Jacques-Henri
Jourdan

Unsafe Rust

Unsafe and aliasing
Safe abstractions

Interior mutability

Cell<T>
RefCell<T>
Rc<T>

1 Unsafe Rust

- Unsafe and aliasing
- Safe abstractions

2 Interior mutability

- Cell<T>
- RefCell<T>
- Rc<T>

Working around aliasing rules

Can we do better than `unsafe`?

On the one hand, Rust aliasing rules are strict;
on the other hand, `unsafe` code seems too subtle to write....

Can we do better?

Unsafe Rust

Unsafe and aliasing
Safe abstractions

Interior mutability

`Cell<T>`
`RefCell<T>`
`Rc<T>`

Working around aliasing rules

Can we do better than `unsafe`?

On the one hand, Rust aliasing rules are strict;
on the other hand, `unsafe` code seems too subtle to write....

Can we do better?

We can use `interior mutability`:

- libraries that relax aliasing rules, `safely`,
- written with `unsafe` code, but safely encapsulated!
- Common feature: updating memory using a shared borrow, with appropriate restrictions.
 - (Uses special annotation to disable some optimizations.)

Any idea of an API with interior mutability?

```
pub struct Cell<T> { ... }

impl<T> Cell<T> {
    pub fn new(value: T) -> Cell<T> { ... }
    pub fn into_inner(self) -> T { ... }
    pub fn set(&self, val: T) { ... }
    pub fn replace(&self, val: T) -> T { ... }
    pub fn get(&self) -> T where T : Copy { ... }
}
```

Unsafe Rust

Unsafe and aliasing
Safe abstractions

Interior mutability

Cell<T>
RefCell<T>
Rc<T>

Informally, why is this safe?

```
pub struct Cell<T> { ... }

impl<T> Cell<T> {
    pub fn new(value: T) -> Cell<T> { ... }
    pub fn into_inner(self) -> T { ... }
    pub fn set(&self, val: T) { ... }
    pub fn replace(&self, val: T) -> T { ... }
    pub fn get(&self) -> T where T : Copy { ... }
}
```

Informally, why is this safe?

From `&Cell<T>`, we can never get a (shared or mutable) borrow of the content. Hence, invariants on borrows of `T` cannot be violated.

We can only exchange values of type `T` or get a copy, but no internal borrow.

Unsafe Rust

Unsafe and aliasing
Safe abstractions

Interior mutability

Cell<T>
RefCell<T>
Rc<T>

```
pub struct Cell<T> { ... }

impl<T> Cell<T> {
    pub fn new(value: T) -> Cell<T> { ... }
    pub fn into_inner(self) -> T { ... }
    pub fn set(&self, val: T) { ... }
    pub fn replace(&self, val: T) -> T { ... }
    pub fn get(&self) -> T where T : Copy { ... }
}
```

Unsafe Rust

Unsafe and aliasing

Safe abstractions

Interior mutability

Cell<T>

RefCell<T>

Rc<T>

Informally, why is this safe?

From `&Cell<T>`, we can never get a (shared or mutable) borrow of the content. Hence, invariants on borrows of `T` cannot be violated.

We can only exchange values of type `T` or get a copy, but no internal borrow.

And what if we **do want** an internal borrow?

RefCell<T> API(1/2)

RefCell, RefMut

```
pub struct RefCell<T> { ... }

pub struct RefMut<'b, T> where T: 'b { ... }

impl<T> RefCell<T> {
    pub fn new(value: T) -> RefCell<T> { ... }
    pub fn into_inner(self) -> T { ... }

    /* Checks there is no borrow. Marks as uniquely borrowed. */
    pub fn borrow_mut<'a>(&'a self) -> RefMut<'a, T> { ... }
}

/* This DerefMut instance means RefMut<'b, T> can be used as &'b mut T*/
impl<'b, T> DerefMut for RefMut<'b, T> {
    pub fn deref_mut<'a>(&'a mut self) -> &'a mut T /* where 'b: 'a */ { ... }
}
/* This Deref instance means RefMut<'b, T> can be used as &'b T */
impl<'b, T> Deref for RefMut<'b, T> {
    type Target = T
    pub fn deref<'a>(&'a self) -> &'a T /* where 'b: 'a */ { ... }
}

/* Destructor. */
impl<'a, T> Drop for RefMut<'a, T> {
    /* Mark RefCell as not borrowed. */
    pub fn drop(&mut self) { ... }
}
```

```
fn use_refcell(x : &RefCell<Vec<i32>>) {  
    let mut v: RefMut<'_, Vec<i32>> = x.borrow_mut();  
    v.push(42); // v can be used just like a unique borrow  
  
    /* Panics: there already is a unique borrow. */  
    /* let v2 = x.borrow_mut().push(16); */  
  
    /* Implicit : v.drop(); */  
}  
  
/* The RefMut is dropped, I can create another one: */  
println!("{}", x.borrow_mut()[0]);  
}
```

RefCell<T> API (2/2)

Ref

```
...
pub struct Ref<'b, T> where T: 'b { ... }

impl<T> RefCell<T> {
    ...
    /* Checks there is no unique borrow. Increments borrow count. */
    pub fn borrow<'a>(&'a self) -> Ref<'a, T> { ... }
}

/* This Deref instance means Ref<'b, T> can be used as &'b T */
impl<'b, T> Deref for Ref<'b, T> {
    type Target = T
    pub fn deref<'a>(&'a self) -> &'a T /* where 'b: 'a */ { ... }
}

/* Destructor. */
impl<'a, T> Drop for Ref<'a, T> {
    /* Decrement borrow count. */
    pub fn drop(&mut self) { ... }
}
```

Soundness

Introduction to
Rust (2/2)

Jacques-Henri
Jourdan

Why is RefCell sound?

Unsafe Rust

Unsafe and aliasing

Safe abstractions

Interior mutability

Cell<T>

RefCell<T>

Rc<T>

Soundness

Why is RefCell sound?

The aliasing rule (aliasing XOR mutation) is enforced dynamically, through an internal counter.

We can see RefCell as a non-concurrent reader/writer lock.

Subtle API question regarding lifetimes

In the standard library:

```
impl<'b, T> RefMut<'b, T> {
    pub fn map<U, F>(orig: RefMut<'b, T>, f: F) -> RefMut<'b, U>
        where F: FnOnce(&mut T) -> &mut U
    { ... }
}

impl<'b, T> Ref<'b, T> {
    pub fn map<U, F>(orig: Ref<'b, T>, f: F) -> Ref<'b, U>
        where F: FnOnce(&T) -> &U
    { ... }
}
```

It can be used e.g., to transform a `RefMut<'b, T>` to a `RefMut` to one of the fields of `T`.

Subtle API question regarding lifetimes

In the standard library:

```
impl<'b, T> RefMut<'b, T> {  
    pub fn map<U, F>(orig: RefMut<'b, T>, f: F) -> RefMut<'b, U>  
        where F: FnOnce(&mut T) -> &mut U  
    { ... }  
}  
  
impl<'b, T> Ref<'b, T> {  
    pub fn map<U, F>(orig: Ref<'b, T>, f: F) -> Ref<'b, U>  
        where F: FnOnce(&T) -> &U  
    { ... }  
}
```

It can be used e.g., to transform a `RefMut<'b, T>` to a `RefMut` to one of the fields of `T`.

Exercise: what are the lifetimes of borrows  used in closures?
Give (counter-)examples.

Rc<T>

A pointer to T, with reference counting

Jacques-Henri
Jourdan

```
struct Rc<T> { ... }

impl<T> Rc<T> {
    pub fn new(value: T) -> Rc<T> { ... }
}

/* This Deref instance means Rc<T> can be used as &T */
impl<T> Deref for Rc<T> {
    type Target = T
    pub fn deref<'a>(&'a self) -> &'a T { ... }
}

impl<T> Clone for Rc<T> {
    /* Copy the pointer, increment the reference count. */
    pub fn clone(&self) -> Rc<T> { ... }
}

impl<T> Drop for Rc<T> {
    /* Drop the pointer, decrement the reference count and recursively
       drop+deallocate if count is 0. */
    pub fn clone(&self) -> Rc<T> { ... }
}
```

Unsafe Rust

Unsafe and aliasing
Safe abstractions

Interior mutability

Cell<T>
RefCell<T>
Rc<T>

Rc<T>

A pointer to T, with reference counting

```
struct Rc<T> { ... }

impl<T> Rc<T> {
    pub fn new(value: T) -> Rc<T> { ... }
}

/* This Deref implements Deref for Rc<T>
 * Copy the Target from self
 */
impl<T> Deref for Rc<T> {
    type Target = T;
    pub fn deref(self) -> &T {
        &self.0
    }
}

impl<T> Clone for Rc<T> {
    /* Copy the Target from self
     * Create a new Rc with the same count
     */
    pub fn clone(&self) -> Rc<T> { ... }
}

impl<T> Drop for Rc<T> {
    /* Drop the pointer, decrement the reference count and recursively
     * drop+deallocate if count is 0. */
    pub fn drop(self) -> () {
        self.0 = None;
    }
}
```

This is typically used for implementing **data structures with sharing**.
Example: purely functional maps, BDDs...

Why do I say this is interior mutability?

Rc<T>

A pointer to T, with reference counting

```
struct Rc<T> { ... }

impl<T> Rc<T> {
    pub fn new(value: T) -> Rc<T> { ... }
}

/* This Deref implements Deref for Rc<T>
 * Copy the value
 */
impl<T> Deref for Rc<T> {
    type Target = T;
    pub fn deref(self) -> T {
        self.0.deref()
    }
}

impl<T> Clone for Rc<T> {
    /* Copy the pointer */
    pub fn clone(&self) -> Rc<T> { ... }
}

impl<T> Drop for Rc<T> {
    /* Drop the pointer, decrement the reference count and recursively
     * drop+deallocate if count is 0. */
    pub fn drop(self) {
        self.0.drop();
        self.0 = None;
    }
}
```

This is typically used for implementing **data structures with sharing**.
Example: purely functional maps, BDDs...

Interior mutability is limited to the reference count.

Getting immutable references

A priori, an `Rc<T>` can be aliased, so we don't have a `DerefMut` instance.

But we have:

```
impl<T> Rc<T> {  
    /* Checks the count is equal to 1. */  
    pub fn get_mut(this: &mut Rc<T>) -> Option<&mut T> { ... }  
  
    /* Clone-on-write: clone the content to a fresh location if the count is not 1. */  
    pub fn make_mut(this: &mut Rc<T>) -> &mut T  
}
```

Of course, this prevent mutation and aliasing.

How would you get a functionality close to an OCaml's ref type?

Getting immutable references

A priori, an `Rc<T>` can be aliased, so we don't have a `DerefMut` instance.

But we have:

```
impl<T> Rc<T> {  
    /* Checks the count is equal to 1. */  
    pub fn get_mut(this: &mut Rc<T>) -> Option<&mut T> { ... }  
  
    /* Clone-on-write: clone the content to a fresh location if the count is not 1. */  
    pub fn make_mut(this: &mut Rc<T>) -> &mut T  
}
```

Of course, this prevent mutation and aliasing.

How would you get a functionality close to an OCaml's ref type?

Answer: `Rc<RefCell<T>>`. This is a common pattern.

A note on performances

Reference counting is sometimes considered as slow.

This is because it usually require a lot of updates to the reference counts (ex. parameter passing, assignments to a variable...)

In Rust, we can mix reference counting and borrowing:

- Use borrows when doing a read-only traversal of a data structure.
- Increment the count only when creating a new long-lived reference.

This gives better control, and better performances.