Introduction to Rust

Jacques-Henri Jourdan

December 10th, 2024

Low-level programming

Some software requires a high level of control:

- over memory layout;
- over when memory is allocated and freed;
- over what computations are done and when...

The standard "answer" to these issues is C/C++.
These languages have shortcomings:

- very complicated semantics (pointer optimizations...),
- no safety guarantee,
- poor support for abstraction.

Rust

A language initially developed by Mozilla for rewriting parts of Firefox

High performance, high level of control with safe abstraction mechanisms

Rust

A language initially developed by Mozilla for rewriting parts of Firefox

High performance, high level of control with safe abstraction mechanisms

Zero-cost abstraction

Powerful abstraction mechanisms with no impact on performance:

- Sound type system
- Pointers: ownership + borrows with lifetimes
- Polymorphism with traits ($\simeq$ type classes)

When the type system is not expressive enough, we can use unsafe features behind safe abstractions.

Why study Rust here?

Many concepts from type systems, adapted them to a new context:

- Polymorphism, type traits, closures, algebraic types...

New type system features:

- Ownership types, borrows, lifetimes,
- Concurrency: fine-grained tracking of thread-(un)safe types.

New challenges in type system meta-theory:

- ownership,
- unsafe code behind safe abstractions.

What you will learn here

My goal is that at the end of the semester, you will:

- know how to write simple Rust programs;
- roughly understand how the type system works,
  - both formally and how it can be implemented;
- have a glimpse of how to formally study Rust:
  - How to use the type system to verify Rust programs?
  - How to build a logical relation to prove soundness?

Today: learn a bit of the language

Resources

To learn Rust as a developer:

- Plenty of references here: https://www.rust-lang.org/learn
- My favorites:
  - Rustlings small exercises to get started;
  - Rust 101 by Ralf Jung: accessible tutorial for the core language;
    - Many examples of this course taken from this tutorial
  - Rust by example: learn by example;
  - The Rust Book: comprehensive, but long, "official" tutorial for Rust and some of its ecosystem.

Metatheory of the type system:

- RustBelt: Securing the Foundations of the Rust Programming Language. Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, Derek Dreyer. POPL 2018.

Homework

During future courses, we will do some exercises together.

However, these exercises are a bit involved and expect you to know the basics of the language.

In order to practice, please do exercises 0-13 and 16 from Rustlings (`https://github.com/rust-lang/rustlings/`).

You have some time: the next Rust course is on February 4th, 2025.

The minimum of a sequence of integers

```rust
enum NumberOrNothing {
    Number(i32), Nothing
}


fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  let mut res = NumberOrNothing::Nothing;
  for el in vec {
    match res {
      NumberOrNothing::Nothing => {
        res = NumberOrNothing::Number(el);
      },
      NumberOrNothing::Number(n) => {
        let m = if n < el { n } else { el };
        res = NumberOrNothing::Number(m);
      }
    }
  }

  return res
}
```

A function for computing the maximum of a sequence of 32-bits integers.

- Parameters and return types need to be annotated.
- Types for local variables are inferred (most often).

The minimum of a sequence of integers

```
enum NumberOrNothing {
    Number(i32), Nothing
}


fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  let mut res = NumberOrNothing::Nothing;
  for el in vec {
    match res {
      NumberOrNothing::Nothing => {
        res = NumberOrNothing::Number(el);
      },
      NumberOrNothing::Number(n) => {
        let m = if n < el { n } else { el };
        res = NumberOrNothing::Number(m);
      }
    }
  }

  return res
}
```

Empty sequence
$\Rightarrow$ no defined minimum

We use an algebraic data type

- 32 bits integer, or nothing
- pattern matching
- constructors

The minimum of a sequence of integers

```rust
enum NumberOrNothing {
    Number(i32), Nothing
}
use NumberOrNothing::*;

fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  let mut res =                 Nothing;
  for el in vec {
    match res {
                     Nothing => {
        res =                Number(el);
      },
                     Number(n) => {
        let m = if n < el { n } else { el };
        res =                Number(m);
      }
    }
  }

  return res
}
```

Empty sequence
$\Rightarrow$ no defined minimum

We use an algebraic data type

- 32 bits integer, or nothing
- pattern matching
- constructors
  - In a specific namespace, can be imported

The minimum of a sequence of integers

```
enum NumberOrNothing {
    Number(i32), Nothing
}
use NumberOrNoth

fn vec_min(vec:
  let mut res =
  for el in vec
    match res {

        res =
      },

        let m =
        res =
      }
    }
  }

  return res
}
```

A value of type `NumberOrNothing` is a sequence of bytes (tag followed by `i32` payload).

- Control on memory representation, like in C/C++.
- Drawback: values have sizes, which the compiler need to know.
- In higher-level like OCaml, Java, Python, Haskell, ... this is different:
  - complex values are boxed;
  - values always use one memory word;
  - unavoidable implicit indirection.

Using vec_min

```
impl NumberOrNothing {
  fn print(self) {
    match self {
      Nothing =>
        println!("The number is: <nothing>"),
      Number(n) =>
        println!("The number is: {}", n),
    };
  }
}

fn main() {
  let v = vec![18,5,7,2,9,27];
  let min = vec_min(v);
  min.print()
}
```

We can define associated functions for types:

- like sealed methods in OO languages,
- no more than a function, but without polluting the global namespace.

Records, tuples, arrays, newtype

We (of course!) have other convenient ways to build types:

- Tuples `(T1, T2, ...)`:
  - construction: `(e1, e2, ...)`
  - projections: `t.0`, `t.1`, ...
- Records declared by `struct R { f1: T1, f2: T2, ... }`
  - construction: `R { f1: e1, f2: e2, ... }`
  - projections: `r.f1`, `r.f2`, ...
- Newtype (as in Haskell) declared by `struct NT(T)`
  - construction: `NT(e)`
  - projection: `nt.0`
- Fixed-length arrays `[T; 42]`
  - construction: `[e; 42]`
  - access: `a[i]`

Records, tuples, arrays, newtype

We (of course!) have other convenient ways to build types:

- Tuples `(T1, T2, ...)`:
  - construction: `(e1, e2, ...)`
  - projections: `t.0`, `t.1`, ...
- Records declared by `struct R { f1: T1, f2: T2, ... }`
  - construction: `R { f1: e1, f2: e2, ... }`
  - projections: `r.f1`, `r.f2`, ...
- Newtype (as in Haskell) declared by `struct NT(T)`
  - construction: `NT(e)`
  - projection: `nt.0`
- Fixed-length arrays `[T; 42]`
  - construction: `[e; 42]`
  - access: `a[i]`

Again, records, tuples, enum and arrays are stored with no implicit pointer indirection

- Values of tuples, records, arrays and enums types are the concatenation of their components.

Ownership: counter example in C++

```
void foo(std::vector<int> v) {
  int *first = &v[0];
  v.push_back(42);
  *first = 1337;
}
```

Where is the bug?

Ownership: counter example in C++

```
void foo(std::vector<int> v) {
  int *first = &v[0];
  v.push_back(42);
  *first = 1337;
}
```

- push_back may reallocate the vector,
  - invalidating inner pointers such as first.
- Somewhat perverse bug:
  - most of the time, push_back does not reallocate the vector.
- We mutated memory through two aliased pointers, first and v.

Bugs caused by aliasing

Iterator invalidation:

- Mutating a data structure invalidates its iterators.
  - In C++, the rule is complicated (depends on where you iterate and what mutation...).
  - Common source of bugs in C++/Java/...
  - Often undetected by the library: unexpected consequences.
  - Core of the problem: mutation and aliasing at the same time.

Double free:

- Can lead to state inconsistency of the memory management library.
- Another form of simultaneous mutation (free) and aliasing.

Data races:

- Particularly difficult to test/debug.
- By definition: mutation and aliasing.

The general aliasing rule

In Rust, we maintain the invariant: mutation XOR aliasing.

If we want to mutate, we need to have the unique alias.

- We are the owner of the memory location.
- We view memory as a resource which cannot be duplicated.

This also applies to freeing memory.

- There is no GC in Rust (better control of resources, better performances).

Ownership in vec_min

```
impl NumberOrNothing {
  fn print(self) { ... }
}

fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  ...
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(v).print();


}
```

Ownership in vec_min

```rust
impl NumberOrNothing {
  fn print(self) { ... }
}

fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  ...
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(v).print();
  v.push(42);
  vec_min(v).print();
}
```

Ownership in vec_min

```
impl NumberOrNothing {
  fn print(self) { ... }
}

fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  ...
}

fn main() {
  let mut v = ve
  vec_min(v).pri
  v.push(42);
  vec_min(v).pri
}
```

```
error[E0382]: borrow of moved value: ‘v‘
  --> src/main.rs:37:3
    |
35  |   let mut v = vec![18,5,7,2,9,27];
        ...
36  |   vec_min(v).print();
    |           - value moved here
37  |   v.push(42);
    |   ^^^^^^^^^^ value borrowed here after move
```

Ownership in vec_min

```
impl NumberOrNothing {
  fn print(self) { ... }
}

fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  ...
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(v).print();
  v.push(42);
  vec_min(v).print();
}
```

Passing `v` as a parameter to `vec_min` triggered a move.

- We lost its ownership. We can no longer use it.
- That's fortunate, because `vec_min` automatically freed the vector.
  - RAII: destructor is called as soon as variable leaves its scope (except if already moved).

The iconic ownership type: Box<T>

`Box<T>` is the type of owned pointers to `T`.

- Creation function:

  ```
  fn new(x: T) -> Box<T>
  ```

  - Move of `x` to dynamically allocated memory (i.e., `malloc`).
  - Transfer of ownership.
- Can be dereferenced (for read/write): `*p`.
- RAII: Automatically freed when leaves the scope.

Example of use: singly linked list:

```
type List = Option<Box<Node>>;
struct Node { elem: i32, next: List, }
```

Exception to ownership tracking

Is ownership tracking a good thing for all variables?

Exception to ownership tracking

No! Some values are bare data, and can be copied freely.

- Examples: `i32`, `(i64, u64)`, `bool`, ...

We say that these types implement the `Copy` trait

- Tells the compiler to disable ownership tracking for values of these types.

By default, types declared with `struct` or `enum` are not `Copy`.

- Because they could be used to represent resources not know by the compiler.
- To make it so, this is easy:

```rust
#[derive(Copy, Clone)]
enum NumberOrNothing {
    Number(i32), Nothing
}
```

- (More explanations on traits later.)

Shared borrows

Variant of `vec_min` to avoid the move:

```
fn vec_min(vec: &Vec<i32>) -> NumberOrNothing {
  let mut res = Nothing;
  for el in vec {
    match res {
      Nothing => { res = Number(*el); },
      Number(n) => {
        let m = if n < *el { n } else { *el };
        res = Number(m);
      }
    }
  }
  return res
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(&v).print();
  v.push(42);
  vec_min(&v).print();
}
```

Instead of passing the ownership of the vector we pass a shared borrow of it.

- At runtime: a pointer to `v`
- Here, the shared borrow is only valid for the duration of the function.
- `&T` implements `Copy`.
- `&T` does not allow mutation (usually... see next course).
- aka. immutable borrow, immutable reference.

Shared borrows

Variant of `vec_min` to avoid the move:

```
fn vec_min(vec: &Vec<i32>) -> NumberOrNothing {
  let mut res = Nothing;
  for el in vec {
    match res {
      Nothing => { res = Number(*el); },
      Number(n) => {
        let m = if n < *el { n } else { *el };
        res = Number(m);
      }
    }
  }
  return res
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(&v).print();
  v.push(42);
  vec_min(&v).print();
}
```

What is the type of `el`? Why?

Shared borrows

Variant of `vec_min` to avoid the move:

```
fn vec_min(vec: &Vec<i32>) -> NumberOrNothing {
  let mut res = Nothing;
  for el in vec {
    match res {
      Nothing => { res = Number(*el); },
      Number(n) => {
        let m = if n < *el { n } else { *el };
        res = Number(m);
      }
    }
  }
  return res
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(&v).print();
  v.push(42);
  vec_min(&v).print();
}
```

When we access elements of a shared borrow of a vector (e.g., by iterating with a `for` loop), we get shared borrows of the content, because, in general, we cannot transfer ownership out of the vector.

Thus we need to dereference `el`!

Unique borrows

```
fn vec_inc(vec: &mut Vec<i32>) {
  for el in vec {
    *el += 1
  }
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_inc(&mut v);
  v.push(42); /* ... */
}
```

Unique borrows are similar to shared borrows.

Differences:

- Allows mutation.
- ...

Unique borrows

```
fn vec_inc(vec: &mut Vec<i32>) {
  for el in vec {
    *el += 1
  }
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_inc(&mut v);
  v.push(42); /* ... */
}
```

Unique borrows are similar to shared borrows.

Differences:

- Allows mutation.
- Does not implement `Copy`.
- aka. mutable borrow, mutable reference.

Borrows and methods

```
impl NumberOrNothing {
  fn print( self) {
    match self {
      Nothing =>
        println!("The number is: <nothing>"),
      Number(n) =>
        println!("The number is: {}", n),
    };
  }
}
```

Call:

- copy of `self`
- full ownership transfer

Can we pass a borrow for `self`?

Borrows and methods

```rust
impl NumberOrNothing {
  fn print(&self) {
    match self {
      Nothing =>
        println!("The number is: <nothing>"),
      Number(n) =>
        println!("The number is: {}", n),
    };
  }
}
```

We can tell that `print` takes a borrow to `self`.

Same syntax at call site:

```rust
let n = Nothing;
n.print();
```

Borrowing is automatic at call-site for `self`.

Of course, this also works with `&mut`.

Pointers in data structures

Imagine we have a vector of non-`Copy` elements. Say, `Vec<Vec<i32>>`.
How do we mutate the content?

One possibility:

```
fn set(i: i32, v: Vec<i32>, l: &mut Vec<Vec<i32>>)
```

The problem: cannot modify a `Vec<i32>` which is already in the vector!
We would need to move out the `Vec<i32>` and then move it in back. Not very efficient.

Pointers in data structures

Imagine we have a vector of non-`Copy` elements. Say, `Vec<Vec<i32>>`.
How do we mutate the content?

Another possibility

```
fn get_mut(i: i32, l: &mut Vec<Vec<i32>>) -> &mut Vec<i32>
```

- But I told you that borrows end when function terminate, which would mean that the return value cannot be used (it aliases the borrowed variable).
- The truth is more complex: borrows have lifetimes.

Lifetimes

Borrows have lifetimes, noted `'a`, `'b`, `'c`...

- In general, for a borrow we write `&'a T` or `&mut 'a T`.
  - In some situations, this can be abbreviated to `&T` / `&mut T`.

Functions can be lifetime polymorphic to return inner pointer.
Example adapted from `Vec` library:

```
fn last_mut<'a, T>(v: &'a mut Vec<T>) -> Option<&'a mut T>
```

- Returns a borrow with the same lifetime as the parameter
- The calling function can update the vector through the returned reference, as long as it does not access the vector directly at the same time
  - This preserves the ownership invariant (mutation XOR aliasing)
- The lifetime is automatically inferred when calling the function

Lifetimes, example

```rust
fn last_mut<'a, T>(v: &'a mut Vec<T>)
  -> Option<&'a mut T>
{ ... }

fn f() {
  let mut v: Vec<i32> = vec![1,2,2];

  let opt = last_mut(&mut v);



  match opt {
    Some(last) => *last = 3,
    None => panic!()
  }


}
```

- When creating a borrow (with `&mut v`), Rust creates a lifetime variable `'a`.
- `v` is marked as mutably borrowed for `'a`.
  - `v` cannot be used as long as `'a` is alive.
- This lifetime variable goes through type inference.
  $\Longrightarrow$ variable `last` has type `&'a mut i32`
- We have only one constraint for `'a`: be alive when `last` is used
  $\Longrightarrow$ `'a` is inferred to be the zone.

Lifetimes, example

```
fn last_mut<'a, T>(v: &'a mut Vec<T>)
  -> Option<&'a mut T>
{ ... }

fn f() {
  let mut v: Vec<i32> = vec![1,2,2];

  let opt = last_mut(&mut v);



  match opt {
    Some(last) => *last = 3,
    None => panic!()
  }

  v.push(4)
}
```

Let's add a mutation of `v` after the end of `'a`.

Rust checks the status of `v` at the call.
$\Longrightarrow$ The lifetime `'a` has ended, `v` is available.
$\Longrightarrow$ Call allowed.

Lifetimes, example

```
fn last_mut<'a, T>(v: &'a mut Vec<T>)
  -> Option<&'a mut T>
{ ... }

fn f() {
  let mut v: Vec<i32> = vec![1,2,2];

  let opt = last_mut(&mut v);

  v.push(42); // Danger!

  match opt {
    Some(last) => *last = 3,
    None => panic!()
  }


}
```

If we try to mutate `v` when it is still borrowed

Rust checks the status of `v` at the call.
The lifetime has NOT ended, the call is impossible.

Lifetimes, example

```
fn last_mut<'a, T>(v: &'a mut Vec<T>)
  -> Option<&'a mut T>
{ ... }

fn f() {
  let mut v: Vec

  let opt = last

  v.push(42); //

  match opt {
    Some(last) =>
    None => pani
  }

}
```

If we try to mutate v when it is still borrowed

```
error[E0499]: cannot borrow `v` as mutable more than once at a time
  --> src/lib.rs:10:3
   |
8  |   let opt = last_mut(&mut v);
   |                      ------ first mutable borrow occurs here
9  |
10 |   v.push(42); // Danger!
   |   ^^^^^^^^^^ second mutable borrow occurs here
11 |
12 |   match opt {
   |         --- first borrow later used here
```

Basic operations on borrows

Copying a shared borrow
(recall: `&T: Copy`):

```
let x = &1;
let y = x;
println!("{} {}", *x, *y);
```

Downcasting mutability:

```
let x = &mut 1;     // x: &'a mut i32
let y: &i32 = x;  // y: &'a i32
```

Splitting a borrow (shared or unique):

```
let x = &mut (1, 2);  // x: &'a mut (i32, i32)
let (x1, x2) = x;     // x1: &'a mut i32   x2: &'b mut i32
```

Directly on a pair/record/enum:

```
let mut p = (42, 12);
let x = &mut p.0;
let y = &p.1; // Allowed even if p.0 is uniquely borrowed
let z = &p.1; // Allowed even if p.1 is already borrowed immutably
let t = p.1;  // Idem
*x = 43; // x: &mut i32
println!("{} {} {}", *x, *y, *z); // y, z: &i32
```

Reborrowing

```
fn last_mut<'a, T>(v: &'a mut Vec<T>)
  -> Option<&'a mut T>
{ ... }

let v = &mut vec![1, 2, 3];
match last_mut(&mut(*v)) { ... }
match last_mut(&mut(*v)) { ... }
```

Reborrowing creates a shorter borrow of the content of a borrow.
The old borrow is reactivated when the new borrow ends.

- Allows using a mutable borrow several times in a row.
- Happens implicitly in the vast majority of cases.
- The lifetime of the original borrow needs to be longer than the reborrowing lifetime.

Lifetime subtyping

Reborrowing uses a notion of lifetime inclusion

- `'a` outlives `'b`, written `'a: 'b`.

This order relation: lifetime subtyping.

- The only source of subtyping in Rust.
- Constrained lifetime polymorphism:

```
fn f<'a, 'b: 'a>(...) -> .. { ... }
fn f<'a, 'b>(...) -> .. where 'b: 'a { ... }
```

- Exercise: what are the variances of `&`? `&mut`? `Box< >`? `Vec< >`?
  - With respect to the lifetime parameter?
  - With respect to the type parameter?

Lifetime inference

Most of the lifetime information is inferred by the compiler.

- We only need to annotate function types (and type declarations).

Key component of `rustc`: the borrow checker.

- Runs after traditional type-checking, and uses information from it.
- Uses variable liveness information from dataflow analysis.
- Tracks ownership (e.g., `Box<T>` value used only once) and infers lifetimes.
- Implemented on MIR, an intermediate representation in CFG form.

The borrow checker

Lifetime inference

The borrow checker interprets each lifetime `'a` as a set $[\![\texttt{'a}]\!]$ containing:

- nodes of the CFG,
- elements `end('a)` for lifetimes `'a` generalized at the function's prototype.

Type-checking generates fresh lifetime variables and outlives constraints "`'a: 'b`":

- Examples: reborrowing, function calls, coercions, ...
- Interpreted as set inclusion $[\![\texttt{'b}]\!] \subseteq [\![\texttt{'a}]\!]$

Further constraints are added:

- $\texttt{L} \in [\![\texttt{'a}]\!]$ when `'a` appears in the type of a live variable at CFG node `L`;
- $\texttt{end('a)} \in [\![\texttt{'a}]\!]$ for any universal variable `'a`;
- $\texttt{L} \in [\![\texttt{'a}]\!]$ for any universal variable `'a` and any CFG node `L`.

Borrow checker finds the smallest solution of the set of constraints (fixpoint algorithm).

The borrow checker

Checking the program

Once sets $[\![$'a$]\!]$ are computed, it remains to check:

- that variables accesses are valid:
  - No read if not initialized or moved out.
  - No access if existing mutable loan with alive lifetime.
  - No write if existing immutable load with alive lifetime.
  - Question: when should borrowing be allowed?

- that outlives constraints at the function prototype are sufficient:
  - For any universally quantified lifetimes `'a` and `'b`, if `end('a)` $\in [\![$`'b`$]\!]$, then outlives constraints should imply `'b: 'a`.

The borrow checker

Checking the program

Once sets ⟦'a⟧ …

- that variab…
  - No rea…
  - No acc…
  - No wri…
  - Questi…
- that outliv…
  - For an… constra… should imply 'b: 'a.

Implementing the borrow checker for a much simplified version of Rust will be your programming project!

Additional complications in `rustc`:

- Instead of variables: places
  - E.g., `rustc` treats `p.0` and `p.1` independently.
- Language features: closures...
- Efficient algorithm for fixpoint computation.
- ...

1 Basic types

2 Aliasing control
- Ownership in Rust
- Borrows and lifetimes

3 Unsafe Rust
- Unsafe and aliasing
- Safe abstractions

4 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 (statically known lifetime),
- linking to external libraries: 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

- Dereference raw pointers.
- Call unsafe functions (e.g., accessing `Vec<T>` without bounds checks).
- Implement `unsafe` traits (e.g., `Send` and `Sync`, see next week).

And things we won't discuss:

- Mutate global variables (called `static` variables in Rust)
  - (because that can lead to data races).
- Access `union` types.

Undefined behaviors

Using these features is unsafe.
$\Longrightarrow$ the compiled program can crash even if type-checked!

It is important to know when a program triggers undefined behavior.
This is sometimes 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 known 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
}
```

$\Longrightarrow$ The programmer should take care of not breaking these aliasing guarantees using raw pointers!

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 known 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
}
```

$\Longrightarrow$ The programmer should take care of not breaking these aliasing guarantees using raw pointers!

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 known aliasing properties on borrows to perform some optimizations:

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

$\Longrightarrow$ The programmer should take care of not breaking these aliasing guarantees using raw pointers!

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 two sets of rules called Stacked Borrows and Tree Borrows:

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

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 fo
- allowing op

Choosing these
The Miri interpr
Borrows:

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

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

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

Safe abstractions

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!

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?

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?

Cell<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 { ... }
}
```

Informally, why is this safe?

Cell<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 { ... }
}
```

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.

Cell<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 { ... }
}
```

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> {
    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
    fn deref<'a>(&'a self) -> &'a T /* where 'b: 'a */ { ... }
}

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

RefCell<T> Example

```
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
    fn deref<'a>(&'a self) -> &'a T /* where 'b: 'a */ { ... }
}

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

Soundness

Why is `RefCell` sound?

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

```
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
    fn deref<'a>(&'a self) -> &'a T { ... }
}

impl<T> Clone for Rc<T> {
    /* Copy the pointer, and increment the reference count. */
    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. */
    fn drop(&mut self) { ... }
}
```

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 in
impl<T> Deref fo
    type Target
    fn deref<'a>
}

impl<T> Clone fo
    /* Copy the p
    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. */
    fn drop(&mut self) { ... }
}
```

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 in
impl<T> Deref fo
    type Target
    fn deref<'a>
}

impl<T> Clone fo
    /* Copy the
    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. */
    fn drop(&mut self) { ... }
}
```

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

Interior mutability is limited to the reference count.

Getting mutable 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
      where T : Clone { ... }
}
```

Of course, this prevents mutation and aliasing.
How would you get a functionality close to an OCaml's `ref` type?

Getting mutable 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
      where T : Clone { ... }
}
```

Of course, this prevents mutation and aliasing.
How would you get a functionality close to an OCaml's `ref` type?

Answer: `Rc<RefCell<T>>`.

A note on performances

Reference counting is sometimes considered slow.
This is because it usually requires 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.