Lecture 6: System safety

Alexander Stanovoy

April 26, 2022

alex.stanovoy@gmail.com

In this lecture

- Lifetimes
- Lifetime elision
- Higher-Rank Trait Bounds
- Subtyping
- Variance
- Drop checker
- Conclusion

Lifetimes

A word about Rust in general

Every language has its main aims. They affect how the language is developed, its priorities, and define the weak and strong sides.

Rust is not an exception. The three main aims of Rust are:

- **Performance**: Fast and memory efficient.
- **Reliability** (or safety): Safe and don't let us even compile unsafe code.
- **Productivity**: Overall speed of development. It's about community, documentation, and tooling.

Today, we're going to discuss the Rust's type system, its weak and strong sides, and how it's connected with the **performance** and **safety**.

A word about Rust in general

To be safe, Rust should guarantee that there's no (it's not a comprehensive list):

- Double free.
- Null pointer dereference.
- Aliasing.
- Invalid memory access.
- Use after free.
- Data races.

Fixing double frees is quite easy - even when you write C++ code you likely won't do such a silly mistake because of RAII.

To deal with null pointers, we should either check them or never create them.

But what about the others?

Aliasing

As you already know, compilers want to optimize the code, and we want it to be correct at the same time. You should already know about **aliasing** from the previous courses.

Because two of the aims of Rust are **performance** and **productivity**, we want the language to give as much space to make code fast and readable as possible.

So why should we care about aliasing?

Aliasing

Consider this simple function:

```
fn compute(input: &u32, output: &mut u32) {
    if *input > 10 {
        *output = 1;
    }
    if *input > 5 {
        *output *= 2;
    }
    // remember that `output` will be `2` if `input > 10`
}
```

Aliasing

Our compiler would like to be able to optimize it to the following function:

```
fn compute(input: &u32, output: &mut u32) {
    // keep `*input` in a register
    let cached_input = *input;
    if cached_input > 10 {
        *output = 2;
    } else if cached_input > 5 {
        *output *= 2;
    }
}
```

For almost any other language, this optimization is not sound. This is because the optimization relies on knowing that aliasing doesn't occur.

Aliasing

Rust's borrow checker just checks whether we have one *mutable* OR multiple *shared* references. This rule is named *aliasing XOR mutability*, or *AXM* in research papers.

Therefore we know this input should be impossible because `&mut` isn't allowed to be aliased.

Aliasing

This is why alias analysis is important: it lets the compiler perform useful optimizations! Some examples:

- Keeping values in registers by proving no pointers access the value's memory.
- Eliminating reads by proving some memory hasn't been written to since last we read it.
- Eliminating writes by proving some memory is never read before the next write to it.
- Moving or reordering reads and writes by proving they don't depend on each other.

Even more: aliasing is unsafe and could lead to unsound code, and Rust forbids it.

A word about Rust in general

Although we know about AXM and have already written some Rust code, we don’t know how it works in depth!

As you might already know, any type system rejects some of the wrong programs and at the same time rejects some of the normal ones.

A word about Rust in general

The main problems we're facing are:

- Invalid memory access.
- Use after free.

A word about Rust in general

The main problems we're facing are:

- Invalid memory access.
- Use after free.

Question: How does Rust decides when the references and the variables go out of scope?

Lifetimes

Rust enforces ownership rules through *lifetimes*. Informelly, lifetime is **a named region of code**.

The core idea is to limit the lifetimes of references by giving them a special type that includes not only the type of reference, but the region where this reference is valid too!

Actually, lifetimes are not just *scope* since the lifetime of the variable can branch or even have holes!

Lifetimes

Lifetimes are denoted by the leading quote and usually with just a one symbol name: `'a`, `'b`, `'static`. The lifetimes are written before other generic parameters.

```
pub struct Person<'a, T> {
    pub first_name: &'a str,
    pub last_name: &'a str,
    pub age: usize,
    pub private_property: T,
}
```

Lifetimes

Note that lifetimes are not types, **but the part of the type** and **behave like a type**. They are erased after the borrow check and used only to check ownership rules.

In local scopes, you usually don't need to specify lifetimes: here, Rust lifetime inference works as good as type inference.

Lifetimes

Let's desugar the lifetimes at this simple example!

```
let x = 0;
let y = &x;
let z = &y;

// Note: `'a: {` and `&'b x` is not a valid syntax!
'a: {
    // 'x' has the lifetime 'a
    let x: i32 = 0;
    'b: {
        //
        let y: &'b i32 = &'b x;
        'c: {
            let z: &'c &'b i32 = &'c y;
        }
    }
}
```

Lifetimes

The second example:

```
let x = 0;
let z;
let y = &x;
z = y;
```

Lifetimes

Desugared:

```
'a: {
    let x: i32 = 0;
    'b: {
        let z: &'b i32;
        'c: {
            // Must use 'b here because this reference is
            // being passed to that scope.
            let y: &'b i32 = &'b x;
            z = y;
        }
    }
}
```

Lifetimes

Let's find a mistake using lifetimes in the third example:

```
fn as_str(data: &u32) -> &str {
    let s = format!("{}", data);
    &s
}
```

Desugared:

```
fn as_str<'a>(data: &'a u32) -> &'a str {
    'b: {
        let s = format!("{}", data);
        return &'a s;
    }
}
```

Lifetimes

And the fourth example. How the compiler will see the error?

```
let mut data = vec![1, 2, 3];
let x = &data[0];
data.push(4);
println!("{}", x);
```

Desugared:

```
'a: {
    let mut data: Vec<i32> = vec![1, 2, 3];
    'b: {
        let x: &'b i32 = Index::index::<'b>(&'b data, 0);
        'c: {
            Vec::push(&'c mut data, 4);
        }
        println!("{}", x);
    }
}
```

Lifetimes

Lifetimes can be more difficult than just a scope:

```
let mut data = vec![1, 2, 3];
let x = &data[0];  // 'x' is defined
if some_condition() {
    // This is the last use of 'x' in this branch
    println!("{}", x);
    data.push(4);
} else {
    // There's no use of 'x' in here, so effectively the last use i
    // creation of x at the top of the example.
    data.push(5);
}
```

Lifetimes

...And have holes:

```
let mut data = vec![1, 2, 3];
// This mut allows us to change where the reference points to
let mut x = &data[0];
// Last use of this borrow
println!("{}", x);

data.push(4);
// We start a new borrow here
x = &data[3];
println!("{}", x);
```

Lifetimes

...And have holes:

```
let mut data = vec![1, 2, 3];
// This mut allows us to change where the reference points to
let mut x = &data[0];
// Last use of this borrow
println!("{}", x);

data.push(4);
// We start a new borrow here
x = &data[3];
println!("{}", x);
```

After these examples, we can also conclude: `let` statement creates a new lifetime.

Lifetimes

Unfortunately, borrow checker is not ideal; it can forbid code that should've compile:

```
fn get_default<K, V>(map: &mut HashMap<K, V>, key: K) -> &mut V
where
    K: Clone + Eq + Hash,
    V: Default,
{
    match map.get_mut(&key) {
        Some(value) => value,
        None => {
            map.insert(key.clone(), V::default());
            map.get_mut(&key).unwrap()
        }
    }
}
```

Lifetimes

```
error[E0499]: cannot borrow `*map` as mutable more than once at a time
  --> src/main.rs:12:13
...
9  |       match map.get_mut(&key) {
   |       -     ----------------- first mutable borrow occurs here
   |  _____|
   | |
11 | |         None => {
12 | |             map.insert(key.clone(), V::default());
   | |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ second
   | |                               mutable borrow occurs here
...
15 | |     }
   | |_____- returning this value requires that `*map`
   |         is borrowed for `'1`
```

Lifetime elision

As you already seen, Rust don’t require from you to type the lifetimes every single time. This property is called a *lifetime elision*.

Lifetime elision

As you already seen, Rust don’t require from you to type the lifetimes every single time. This property is called a *lifetime elision*.

Lifetime positions can appear as either “input”or “output”.

Lifetime elision

As you already seen, Rust don't require from you to type the lifetimes every single time. This property is called a *lifetime elision*.

Lifetime positions can appear as either "input"or "output".

- For `fn` definitions, `fn` types, and the traits `Fn`, `FnMut`, and `FnOnce`, input refers to the types of the formal arguments, while output refers to result types, so that

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

  Has elided one lifetime in input position and two lifetimes in output position.

Lifetime elision

As you already seen, Rust don't require from you to type the lifetimes every single time. This property is called a *lifetime elision*.

Lifetime positions can appear as either "input"or "output".

- For `fn` definitions, `fn` types, and the traits `Fn`, `FnMut`, and `FnOnce`, input refers to the types of the formal arguments, while output refers to result types, so that

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

  Has elided one lifetime in input position and two lifetimes in output position.
- For `impl`'s, all types are input, so that

  ```
  impl Trait<&T> for Struct<&T>
  ```

  Has elided two lifetimes in input position, while

  ```
  impl Struct<&T>
  ```

  Has elided one.

Lifetime elision

Elision rules are as follows:

- Each **elided** lifetime in input position becomes a distinct lifetime parameter.

```
fn print(s: &str);
fn print<'a>(s: &'a str);

fn debug(lvl: usize, s: &str);
fn debug<'a>(lvl: usize, s: &'a str);

fn compare_strs(s: &str, t: &str) -> bool;
fn compare_strs<'a, 'b>(s: &'a str, t: &'b str) -> bool;

fn get_user_data_from<'a>(user: &'a User,
    server: &Server) -> Data;
fn get_user_data_from<'a, 'b>(user: &'a User,
    server: &'b Server) -> Data;
```

Lifetime elision

Elision rules are as follows:

- Each **elided** lifetime in input position becomes a distinct lifetime parameter.
- If there is **exactly** one input lifetime position (elided or not), that lifetime is assigned to all elided output lifetimes.

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

// This won't work!
fn frob(s: &str, t: &str) -> &str;
```

Lifetime elision

Elision rules are as follows:

- Each **elided** lifetime in input position becomes a distinct lifetime parameter.
- If there is **exactly** one input lifetime position (elided or not), that lifetime is assigned to all elided output lifetimes.
- If there are multiple input lifetime positions, but one of them is `&self` or `&mut self`, the lifetime of `self` is assigned to all elided output lifetimes.

```
fn get_mut(&mut self) -> &mut T;
fn get_mut<'a>(&'a mut self) -> &'a mut T;

fn args<T: ToString>(&mut self, args: &[T])
    -> &mut Command;
fn args<'a, 'b, T: ToString>(&'a mut self, args: &'b [T])
    -> &'a mut Command;
```

Lifetime elision

Elision rules are as follows:

- Each **elided** lifetime in input position becomes a distinct lifetime parameter.
- If there is **exactly** one input lifetime position (elided or not), that lifetime is assigned to all elided output lifetimes.
- If there are multiple input lifetime positions, but one of them is `&self` or `&mut self`, the lifetime of `self` is assigned to all elided output lifetimes.
- Otherwise, it is an error to elide an output lifetime!

```
// This won't work!
fn get_str() -> &str;
```

Lifetime elision

You can also use wildcard to elide lifetime and let Rust decide automatically what do you need:

```
fn get_iter(buf: &[u8]) -> std::slice::Iter<'_, u8>;
fn get_iter<'a>(buf: &'a [u8]) -> std::slice::Iter<'a, u8>;
```

Lifetimes and possible mistakes

It's possible to write a compiling program that's in the same time is not semantically correct. Let's write an iterator over byte slice:

```
struct ByteIter<'a> {
    remainder: &'a [u8]
}
impl<'a> ByteIter<'a> {
    fn next(&mut self) -> Option<&u8> {
        if self.remainder.is_empty() {
            None
        } else {
            let byte = &self.remainder[0];
            self.remainder = &self.remainder[1..];
            Some(byte)
        }
    }
}
```

Lifetimes and possible mistakes

This code will work just fine:

```
let mut bytes = ByteIter { remainder: b"1" };
assert_eq!(Some(&b'1'), bytes.next());
assert_eq!(None, bytes.next());
```

Lifetimes and possible mistakes

But what about a bit different example?

```
let mut bytes = ByteIter { remainder: b"1123" };
let byte_1 = bytes.next();
let byte_2 = bytes.next();
if byte_1 == byte_2 { ... }
```

Lifetimes and possible mistakes

Nooo, Rust, why!

```
error[E0499]: cannot borrow `bytes` as mutable more
              than once at a time
  --> src/main.rs:20:18
   |
19 |     let byte_1 = bytes.next();
   |                  ----- first mutable borrow occurs here
20 |     let byte_2 = bytes.next();
   |                  ^^^^^ second mutable borrow occurs here
21 |     if byte_1 == byte_2 {
   |        ------ first borrow later used here
```

Lifetimes and possible mistakes

What I'll tell you that the problem lies in our lifetimes? First, we need to desugar our `ByteIter` using the rules of lifetime elision:

```
struct ByteIter<'a> {
    remainder: &'a [u8]
}
impl<'a> ByteIter<'a> {
    fn next(&mut self) -> Option<&u8> {
        if self.remainder.is_empty() {
            None
        } else {
            let byte = &self.remainder[0];
            self.remainder = &self.remainder[1..];
            Some(byte)
        }
    }
}
```

Lifetimes and possible mistakes

What I'll tell you that the problem lies in our lifetimes? First, we need to desugar our `ByteIter` using the rules of lifetime elision:

```
struct ByteIter<'remainder> {
    remainder: &'remainder [u8]
}
impl<'remainder> ByteIter<'remainder> {
    fn next<'rself>(&'rself mut self) -> Option<&'rself u8> {
        if self.remainder.is_empty() {
            None
        } else {
            let byte = &self.remainder[0];
            self.remainder = &self.remainder[1..];
            Some(byte)
        }
    }
}
```

Lifetimes and possible mistakes

Rust thinks that our `Option` returns the link to the `ByteIter`, not to the `remainder`, so because we have a link to the `&mut ByteIter` we cannot call `next` one more time, since it will violate the AXM rule.

Lifetimes and possible mistakes

We need to fix that by using the correct lifetime.

```
struct ByteIter<'remainder> {
    remainder: &'remainder [u8]
}
impl<'remainder> ByteIter<'remainder> {
    fn next(&mut self) -> Option<&'remainder u8> {
        if self.remainder.is_empty() {
            None
        } else {
            let byte = &self.remainder[0];
            self.remainder = &self.remainder[1..];
            Some(byte)
        }
    }
}
```

Lifetimes and possible mistakes

But why Rust compiled the first case?

Lifetimes and possible mistakes

But why Rust compiled the first case?

Because it's correct with respect to the type system, and that implies we do not produce undefined behaviour and memory unsafety, so our program is safe.

Higher-Rank Trait Bounds

Higher-Rank Trait Bounds (HRTBs)

Let's reimplement `.filter()` of `Option`. What we want is to return a `Some(T)` when the value satisfies some predicate and `None` otherwise.

This function accepts some function that takes a reference to a `T` and returns a reference *to inside that* `T`:

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

Higher-Rank Trait Bounds (HRTBs)

This will compile. But we are learning lifetimes here, right? Let's desugar it:

```
impl<T> Option<T> {
    fn filter<F>(self, f: F) -> Option<T>
    where
        F: FnOnce(&'??? T) -> bool
    {
        'b: {
            if let Some(value) = self {
                // 'b is a local lifetime of 'value'
                if f(&'b value) {
                    return Some(value)
                }
            }
        }
        None
    }
}
```

Higher-Rank Trait Bounds (HRTBs)

How are we supposed to express the lifetimes on `F`'s trait bound? We need to provide some lifetime there, but the lifetime we care about can't be named until we enter the body of `filter`!

Question: Why not just write `'a` to the function signature?

Higher-Rank Trait Bounds (HRTBs)

```
impl<T> Option<T> {
    fn filter<'a, F>(self, f: F) -> Option<T>
    where
        F: FnOnce(&'a T) -> bool
    {
        'b: {
            if let Some(value) = self {
                // 'b is a local lifetime of 'value'
                if f(&'b value) {
                    return Some(value)
                }
            }
        }
        None
    }
}
```

Higher-Rank Trait Bounds (HRTBs)

```
error[E0309]: the parameter type `T` may not live long enough
  --> src/main.rs:29:12
   |
26 | impl<T> Option<T> {
   |      - help: consider adding an explicit
   |              lifetime bound...: `T: 'a`
...
29 |         F: FnOnce(&'a T) -> bool
   |            ^^^^^^^^^^^^^^^^^^^^ ...so that the reference
   |                                 type `&'a T` does not
   |                                 outlive the data it points
```

Higher-Rank Trait Bounds (HRTBs)

```
impl<T> Option<T> {
    fn filter<'a, F>(self, f: F) -> Option<T>
    where
        T: 'a,
        F: FnOnce(&'a T) -> bool
    {
        'b: {
            if let Some(value) = self {
                // 'b is a local lifetime of 'value'
                if f(&'b value) {
                    return Some(value)
                }
            }
        }
        None
    }
}
```

Higher-Rank Trait Bounds (HRTBs)

```
error[E0597]: `value` does not live long enough
--> src/main.rs:33:18
   |
27 |     fn filter<'a, F>(self, f: F) -> Option<T>
   |               -- lifetime `'a` defined here
...
33 |             if f(&value) {
   |                --^^^^^^-
   |                | |
   |                | borrowed value does not live long enough
   |                argument requires that `value` is borrowed
   |                for `'a`
...
36 |         }
   |         - `value` dropped here while still borrowed
```

Higher-Rank Trait Bounds (HRTBs)

Remember: lifetimes in function signatures strictly longer than the lifetime of any local variable in this function. In this case, `'a` would be a local lifetime!

Higher-Rank Trait Bounds (HRTBs)

Remember: lifetimes in function signatures strictly longer than the lifetime of any local variable in this function. In this case, `'a` would be a local lifetime!

Function signature lifetimes specify what Rust expects from caller's variables to verify this function is safe and what's the lifetime of the return value to verify the caller will handle the output correctly.

Higher-Rank Trait Bounds (HRTBs)

This is the strong side of Rust: syntax explains what function do, and community strive to write readable code.

Higher-Rank Trait Bounds (HRTBs)

This is the strong side of Rust: syntax explains what function do, and community strive to write readable code.

Moreover, as we'll see today, it's not just an idiom that the community follows - matching function signature is sufficient condition to declare it's usage safe!

Higher-Rank Trait Bounds (HRTBs)

This is how this problem is solved:

```
impl<T> Option<T> {
    fn filter<F>(self, f: F) -> Option<T>
    where
        F: for<'a> FnOnce(&'a T) -> bool
    {
        if let Some(value) = self {
            if f(&value) {
                return Some(value)
            }
        }
        None
    }
}
```

This syntax is called *higher-rank trait bound (HRTB)*: we are generic over the whole spectrum of lifetimes!

Higher-Rank Trait Bounds (HRTBs)

Most of time, you won't need HRTBs: our example will compile without it since compiler will add HRTB for us.

Currently, there're only 3 places in standard library with HRTBs!

T, &T and &mut T

It's a common mistake to think that, for instance, `T` is always an owning type, so we need to understand what means `T`, `&T` and `&mut T` when we write generics. Let's create a table to understand.

Suppose we write the following in our code:

```
impl<T> Trait for T {}
impl<T> Trait for &T {}
impl<T> Trait for &mut T {}
```

This types will match on the following `impl`'s:

T	&T	&mut T
i32, &i32, &mut i32, &&i32, &mut &mut i32	&i32, &&i32, &&mut i32	&mut i32, &mut &mut i32, &mut &i32

`T`, `&T` and `&mut T`

Conclusions:

- `T` is the superset of `&T` and `&mut T`.
- `&T` and `&mut T` are disjoint sets.

Unbounded lifetime

Sometimes you'll need an *unbounded lifetime*, the lifetime that is bigger than any other lifetime. It's definition is `'static`. For instance:

```
let static_str: &'static str = "Hello world!";
```

The `'static` lifetime is basically the definition of lifetime of the whole program.

`T: 'a` and `&'a T`

Sometimes, you'll see the lifetimes for *types*:

```
impl<'a> MyTrait for MyType
where
    Self: 'a,
{
    /* ... */
}
```

This means: "the type is constrained by the lifetime `'a`", i.e **all lifetime parameters** of `T` outlive `'a`.

Let's understand what it means.

`T: 'a` and `&'a T`

Take a look at this simple example: since our `Ref<'a>` cannot outlive the reference inside it, it satisfies `Self: 'a`. But when we try to require `'static` from it, we find out our `Ref` cannot live more then `'a`!

```
trait RequireLifetime<'a> where Self: 'a {}
struct Ref<'a> {
    r: &'a i32,
}

impl<'a> RequireLifetime<'a> for Ref<'a> {}

// lifetime bound not satisfied!
// impl<'a> RequireLifetime<'static> for Ref<'a> {}
```

T: 'a and &'a T

Look: the same applies to `Vec<T>`. We don't have explicit `'a` bound among its generics, but it's limited by the lifetime constraits of `T`.

```
trait RequireLifetime<'a> where Self: 'a {}

impl<'a> RequireLifetime<'a> for Vec<&'a i32> {}

// lifetime bound not satisfied!
// impl<'a> RequireLifetime<'static> for Vec<&'a i32> {}
```

T: 'a and &'a T

Even if we'll have multiple lifetimes and generic types in the structure, it will be constrained by the most strict of constraints:

```
struct Holder<'a, 'b, T, U> {
    s: &'a str,
    t: T,
    u: U,
    ur: &'b U,
}
```

Here, `T` and `U` also have some lifetime, namely `'t` and `'u`, and compiler will choose the strictest of `'a`, `'b`, `'t` and `'u`.

`T: 'a` and `&'a T`

Question: What means `T: 'static`?

`T: 'a` and `&'a T`

Question: What means `T: 'static`?

- `T` doesn't have **any** lifetime constraints, and you can store it for as long as you want. For instance, any owning type such as `i32`, `Vec<String>`, `Range<usize>` or even `static VALUE: str = "hello"` could be stored as much time as you want.

`T: 'a` and `&'a T`

Question: What means `T: 'static`?

- `T` doesn't have **any** lifetime constraints, and you can store it for as long as you want. For instance, any owning type such as `i32`, `Vec<String>`, `Range<usize>` or even `static VALUE: str = "hello"` could be stored as much time as you want.
- But you're not supposed to drop them exactly at the end of the program, of course! This means only that there's no lifetime constraints.

`T: 'a` and `&'a T`

More generally:

- `T: 'a` is a set of all types that are bounded by the lifetime `'a`.

`T: 'a` and `&'a T`

More generally:

- `T: 'a` is a set of all types that are bounded by the lifetime `'a`.
- The set of possible `T: 'a` includes `&'a T`.

Subtyping

Subtyping

Although Rust doesn't have any notion of structural inheritance, it does include subtyping.

Informally, **subtyping** is a concept used to show that *one object is at least as useful as another.* Here, `Cat` `Dog` are subtypes of `Animal`, and `Animal` is their supertype.

```
trait Animal {
    fn snuggle(&self);
}
trait Cat: Animal {
    fn meow(&self);
}
trait Dog: Animal {
    fn bark(&self);
}
```

Subtyping

The following will work because of subtyping:

```
fn love(pet: &dyn Animal) {
    pet.snuggle();
}

struct SomeCat;
impl Animal for SomeCat { ... }
impl Cat for SomeCat { ... }

let cat = SomeCat;
love(&cat);
```

Subtyping

Moreover, subtyping also works with lifetimes!

```
fn shortest<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() < b.len() { a } else { b }
}

let a = "hello";
let b = String::from("students");
println!("{}", shortest(a, b.as_str()));
```

Remember that lifetimes behave like types and they are related to subtyping. Even more: they are the main reason the subtyping exists in Rust!

Question: `'static` vs some `'a` - where's the subtype?

'a: 'b

We can also require from lifetime to **not strictly** outlive another lifetime

How this code works?

```
fn foo<'a, 'b: 'a>(one: &'b str, two: &'a str) -> &'a str {
    one
}

fn main() {
    let string_one = "foo".to_string();
    {
        let string_two = "bar".to_string();
        println!("{}", foo(&string_one, &string_two));
    }
}
```

Variance

Variance

Let's try to answer the following questions:

- Is `&'static T` is a subtype of `&'a T`?

Variance

Let's try to answer the following questions:

- Is `&'static T` is a subtype of `&'a T`? **Yes.**

Variance

Let's try to answer the following questions:

- Is `&'static T` is a subtype of `&'a T`? **Yes.**
- Is `&'a T` is a subtype of `&'a U` when `T` is a subtype of `U`?

Variance

Let's try to answer the following questions:

- Is `&'static T` is a subtype of `&'a T`? **Yes.**
- Is `&'a T` is a subtype of `&'a U` when `T` is a subtype of `U`? **Yes.**

Variance

Let's try to answer the following questions:

- Is `&'static T` is a subtype of `&'a T`? **Yes.**
- Is `&'a T` is a subtype of `&'a U` when `T` is a subtype of `U`? **Yes.**
- Is `&'static mut T` is a subtype of `&'a mut T`?

Variance

Let's try to answer the following questions:

- Is `&'static T` is a subtype of `&'a T`? **Yes.**
- Is `&'a T` is a subtype of `&'a U` when `T` is a subtype of `U`? **Yes.**
- Is `&'static mut T` is a subtype of `&'a mut T`? **Yes.**

Variance

Let's try to answer the following questions:

- Is `&'static T` is a subtype of `&'a T`? **Yes.**
- Is `&'a T` is a subtype of `&'a U` when `T` is a subtype of `U`? **Yes.**
- Is `&'static mut T` is a subtype of `&'a mut T`? **Yes.**
- Is `&'a mut T` is a subtype of `&'a mut U` when `T` is a subtype of `U`?

Variance

Let's try to answer the following questions:

- Is `&'static T` is a subtype of `&'a T`? **Yes.**
- Is `&'a T` is a subtype of `&'a U` when `T` is a subtype of `U`? **Yes.**
- Is `&'static mut T` is a subtype of `&'a mut T`? **Yes.**
- Is `&'a mut T` is a subtype of `&'a mut U` when `T` is a subtype of `U`? **No.**

Variance

Let's try to answer the following questions:

- Is `&'static T` is a subtype of `&'a T`? **Yes.**
- Is `&'a T` is a subtype of `&'a U` when `T` is a subtype of `U`? **Yes.**
- Is `&'static mut T` is a subtype of `&'a mut T`? **Yes.**
- Is `&'a mut T` is a subtype of `&'a mut U` when `T` is a subtype of `U`? **No.**
  It may seem non-intuitive, but imagine if a function will take a `&mut Vec<&'a str>`, and you provide it with a `&mut Vec<&'static str>`. Then you'll be able to put a short-lived reference to `Vec`!

Variance

Example:

```
fn invariant<'a>(vec: &mut Vec<&'a str>, s: &'a str) {
    vec.push(s)
}

// Good to go!
let s = String::from("some &'a str");
let mut vec = vec![];
invariant(&mut vec, s.as_str());

// And this code won't compile!
let mut vec = vec!["some &'static str"];
{
    let s = String::from("some &'a str");
    invariant(&mut vec, s.as_str());
}
println!("{vec:?}");  // It's necessary!
```

Variance

Actually, there's a name to this phenomenon: **variance**, and all types have it. It defines what other *similar* types can be used in that type's place.

There are three kinds of variance: *covariant*, *invariant*, and *contravariant*.

Covariance

A type is **covariant** if you can use a **subtype** in place of the type. It's the most common type of variance.

For instance, `&'a T` is covariant in `'a`. `&'a T` is also covariant in `T`, so you can pass a `&Vec<&'static str>` to a function that takes `&Vec<&'a str>`.

```
fn covariant(x: &dyn Cat) { ... }

covariant(&domestic_cat);
covariant(&cat);
// Won't compile!
// covariant(&animal);
```

Invariance

A type is **invariant** when you must provide *exactly* the given type. Without subtyping, all types would have been invariant!

The main example of invariance is `&'a mut T`. It is *covariant* in `'a` and **invariant** in `T`.

```
// An example just as above, just shortened
fn invariant<'a>(vec: &mut Vec<&'a str>, s: &'a str) {
    vec.push(s)
}

// Won't compile!
let mut vec = vec!["some &'static str"];
{
    let s = String::from("some &'a str");
    invariant(&mut vec, s.as_str());
}
println!("{vec:?}");
```

Contravariance

A type is **contravariant** if you can use **supertype** in the place of the type. The only source of contravariance in the language is the arguments to a function!

In simple terms, `fn(T) -> U` is **contravariant** in `T`, i.e you can use any supertype in place of `T`, and *covariant* in `U`.

```
fn contravariant(f: fn(&'static str)) {}

fn f1(s1: &'static str) {}
fn f2<'a>(s1: &'a str) {}

contravariant(f1);
contravariant(f2);

// In contrast:
// &'static str <: &'a str
// fn(&'a str) <: &'static str
```

Variance

Actually, there's a table with variances of fundamental types, and everything derives from it. You can find it The Rust Reference.

Type	Variance in `'a`	Variance in `T`
`&'a T`	covariant	covariant
`&'a mut T`	covariant	invariant
`*const T`		covariant
`*mut T`		invariant
`[T]` and `[T; n]`		covariant
`fn() -> T`		covariant
`fn(T) -> ()`		contravariant
`fn(T) -> T`		invariant
`std::cell::UnsafeCell<T>`		invariant
`std::marker::PhantomData<T>`		covariant
`dyn Trait<T> + 'a`	covariant	invariant

Variance

Control question: What's the variance of this type?

```
&'a (dyn Trait<'b> + 'c)
```

Variance

Control question: What's the variance of this type?

```
&'a (dyn Trait<'b> + 'c)
```

- Covariant in `'a`.

Variance

Control question: What's the variance of this type?

```
&'a (dyn Trait<'b> + 'c)
```

- Covariant in `'a`.
- Invariant in `'b`.

Variance

Control question: What's the variance of this type?

```
&'a (dyn Trait<'b> + 'c)
```

- Covariant in `'a`.
- Invariant in `'b`.
- Covariant in `'c`.

Variance

Do you remember that actually `dyn Trait` is a type erasure? It must be a mistake to erase the lifeteime!

```
let s = my_string.as_str();
let b: Box<dyn Trait> = Box::new(s);
return b;  // Ooops!
```

That's the reason why `dyn Trait + 'a` have a lifetime associated with it!

Variance

Question: Why this code does not compile?

```
fn evil_feeder<T>(input: &mut T, val: T) {
    *input = val;
}

fn main() {
    let mut mr_snuggles: &'static str = "meow! :3";
    {
        let spike = String::from("bark! >:V");
        let spike_str: &str = spike.as_str();
        evil_feeder(&mut mr_snuggles, spike_str);
    }
}
```

Variance

Desugared:

```
fn evil_feeder<'a>(input: &mut &'a str, val: &'a str) {
    *input = val;
}

fn main() {
    let mut mr_snuggles: &'static str = "meow! :3";
    {
        let spike = String::from("bark! >:V");
        let spike_str: &str = spike.as_str();
        evil_feeder(&mut mr_snuggles, spike_str);
    }
}
```

Variance

Question: How often do you actually need variance?

Variance

Question: How often do you actually need variance?

Getting variance right matters for soundness. But you should not need to think about it at all if you are not writing `unsafe`: it is solely compiler's job until `unsafe` gets involved.

Variance and `unsafe`

We need to find out why variance is important for `unsafe`, so we'll implement our `Cell`.

```
struct MyCell<T> {
    value: T
}
impl<T: Copy> MyCell<T> {
    fn set(&self, new_value: T) {
        // pub unsafe fn write<T>(dst: *mut T, src: T)
        unsafe {
            std::ptr::write(
                &self.value as *const T as *mut T,
                new_value
            );
        }
    }
}
```

Variance and `unsafe`

Question: What's the output of this code?

```
fn foo(rcell: &MyCell<&i32>) {
    let val: i32 = 13;
    rcell.set(&val);
    println!("foo set value: {}", rcell.value);
}

fn main() {
    static X: i32 = 10;
    let cell = MyCell { value: &X };
    foo(&cell);
    println!("end value: {}", cell.value);
}
```

Variance and `unsafe`

Question: What's the output of this code?

```
fn foo(rcell: &MyCell<&i32>) {
    let val: i32 = 13;
    rcell.set(&val);
    println!("foo set value: {}", rcell.value);
}

fn main() {
    static X: i32 = 10;
    let cell = MyCell::new(&X);
    foo(&cell);
    println!("end value: {}", cell.value);
}
```

```
foo set value: 13
end value: 32766  // possible output
```

Variance and `unsafe`

What happened?

Variance and `unsafe`

What happened?

- We've stored a `&'a i32` reference to the cell that can only store `&'static i32` references!

Variance and `unsafe`

What happened?

- We've stored a `&'a i32` reference to the cell that can only store `&'static i32` references!
- By default, compiler won't let us write such code, but in this case `MyCell<T>` is covariant in `T`.

Variance and `unsafe`

What happened?

- We've stored a `&'a i32` reference to the cell that can only store `&'static i32` references!
- By default, compiler won't let us write such code, but in this case `MyCell<T>` is covariant in `T`.
- That resulted in `MyCell` having reference with too small lifetime inside it.

Variance and `unsafe`

Why this code does not compile?

```
struct MyCell<T> { value: T }
impl<T> MyCell<T> {
    fn set(&mut self, value: T) { ... }
}

fn main() {
    static X: i32 = 42;
    let x: &'static i32 = &X;
    let mut cell: MyCell<&'static i32> = MyCell { value: x };
    {
        let y = 228;
        cell.set(&y);
    }
    drop(cell);
}
```

Variance and `unsafe`

Desugared for `T=i32`:

```
struct MyCell<T> { value: T }
impl<T> MyCell<T> {
    // fn set(&mut self, value: T) { ... }
    fn set<'a, 'b>(s: &'a mut MyCell<&'a i32>, value: &'b i32) {
        /* ... */
    }
}
```

Variance and `unsafe`

At the same time, this code won't compile!

```
fn foo(rcell: &Cell<&i32>) {
    let val: i32 = 13;
    rcell.set(&val);
}

fn main() {
    static X: i32 = 10;
    let cell = Cell::new(&X);
    foo(&cell);
}
```

Variance and `unsafe`

```
error[E0597]: `val` does not live long enough
 --> src/main.rs:7:15
  |
5 | fn foo(rcell: &Cell<&i32>) {
  |                     - let's call the lifetime of
  |                       this reference `'1`
6 |     let val: i32 = 13;
7 |     rcell.set(&val);
  |     ----------^^^^- 
  |     |         |
  |     |         borrowed value does not live long enough
  |     argument requires that `val` is borrowed for `'1`
8 | }
  | - `val` dropped here while still borrowed
```

Variance and `unsafe`

Control question: why this code compiles?

```
struct MyCell<T> { value: T }
impl<T> MyCell<T> {
    fn set(&self, value: T) { ... }
}

fn main() {
    static X: i32 = 42;
    let x: &'static i32 = &X;
    let mut cell: MyCell<&'static i32> = MyCell { value: x };
    {
        let y = 228;
        cell.set(&y);
    }
    drop(cell);
}
```

Drop checker

Drop checker

Like in C++, most of lime we exactly know the drop order of variables: it's reversed definition order.

```
let x;
let y;
```

Desugared to scopes:

```
{
    let x; // drops second
    {
        let y; // drops first
    }
}
```

Drop checker

There are some more complex situations which are not possible to desugar using scopes, but the order is still defined – variables are dropped in the reverse order of their definition, fields of structs and tuples in order of their definition.

```
let tuple = (vec![], vec![]);
```

Question: Whereas the left vector is dropped first, does it mean the right one strictly outlives it in the eyes of the borrow checker?

Drop checker

There are some more complex situations which are not possible to desugar using scopes, but the order is still defined – variables are dropped in the reverse order of their definition, fields of structs and tuples in order of their definition.

```
let tuple = (vec![], vec![]);
```

Question: Whereas the left vector is dropped first, does it mean the right one strictly outlives it in the eyes of the borrow checker? **No**.

Drop checker

So why do we care? We care because if the type system isn't careful, it could accidentally make dangling pointers. Consider the following code. Will it compile?

```
struct Inspector<'a>(&'a u8);
struct World<'a> {
    inspector: Option<Inspector<'a>>,
    days: Box<u8>,
}
fn main() {
    let mut world = World {
        inspector: None,
        days: Box::new(1),
    };
    world.inspector = Some(Inspector(&world.days));
}
```

Drop checker

So why do we care? We care because if the type system isn't careful, it could accidentally make dangling pointers. Consider the following code. Will it compile? Yes.

```
struct Inspector<'a>(&'a u8);
struct World<'a> {
    inspector: Option<Inspector<'a>>,
    days: Box<u8>,
}
fn main() {
    let mut world = World {
        inspector: None,
        days: Box::new(1),
    };
    world.inspector = Some(Inspector(&world.days));
}
```

Drop checker

But what if we add a destructor?

```
struct Inspector<'a>(&'a u8);
impl<'a> Drop for Inspector<'a> {
    fn drop(&mut self) {
        println!("I was only {} days from retirement!", self.0);
    }
}
struct World<'a> {
    inspector: Option<Inspector<'a>>, days: Box<u8>
}
fn main() {
    let mut world = World {
        inspector: None,
        days: Box::new(1),
    };
    world.inspector = Some(Inspector(&world.days));
}
```

Drop checker

Let's say `days` happens to get dropped first. Then when `Inspector` is dropped, it will try to read free'd memory!

Drop checker

Let's say `days` happens to get dropped first. Then when `Inspector` is dropped, it will try to read free'd memory!

Implementing `Drop` lets the `Inspector` execute some arbitrary code during its death. This means it can potentially observe that types that are supposed to live as long as it does actually were destroyed first.

Drop checker

Let's say `days` happens to get dropped first. Then when `Inspector` is dropped, it will try to read free'd memory!

Implementing `Drop` lets the `Inspector` execute some arbitrary code during its death. This means it can potentially observe that types that are supposed to live as long as it does actually were destroyed first.

Interestingly, only generic types need to worry about this. If they aren't generic, then the only lifetimes they can harbor are `'static`, which will truly live forever. This is why this problem is referred to as **sound generic drop**. Sound generic drop is enforced by the **drop checker**.

Drop checker

Question: What is the requirements for generic type to soundly implement drop?

Drop checker

Question: What is the requirements for generic type to soundly implement drop?

All generic types' generics arguments must strictly outlive them.

Drop checker

Question: What is the requirements for generic type to soundly implement drop?

All generic types' generics arguments must strictly outlive them.

Up to this point we've only ever interacted with the outlives relationship in an inclusive manner. That is, when we talked about `'a: 'b`, it was ok for `'a` to live exactly as long as `'b`.

Drop checker

Take a look: we **never** access data in **Drop**, but the code doesn't compile!

```
struct Inspector<'a>(&'a u8, &'static str);
impl<'a> Drop for Inspector<'a> {
    fn drop(&mut self) {
        println!("Inspector(_, {})", self.1);
    }
}
struct World<'a> {
    inspector: Option<Inspector<'a>>, days: Box<u8>,
}
fn main() {
    let mut world = World {
        inspector: None,
        days: Box::new(1),
    };
    world.inspector = Some(Inspector(&world.days, "gadget"));
}
```

Drop checker

And another example that doesn't compile even if we don't access data inside:

```
struct Inspector<T>(T, &'static str);
impl<T> Drop for Inspector<T> {
    fn drop(&mut self) {
        println!("Inspector(_, {})", self.1);
    }
}
struct World<T> {
    inspector: Option<Inspector<T>>, days: Box<u8>,
}
fn main() {
    let mut world = World {
        inspector: None,
        days: Box::new(1),
    };
    world.inspector = Some(Inspector(&world.days, "gadget"));
}
```

PhantomData

It appears that everything is good with this definition of `Vec`.

```
struct Vec<T> {
    data: *const T,
    len: usize,
    cap: usize,
}
```

The drop checker will determine that `Vec<T>` does not own any values of type `T`. This will in turn make it conclude that it doesn't need to worry about `Vec` dropping any `T`'s in its destructor for determining drop check soundness. This will in turn allow people to create unsoundness using `Vec`'s destructor!

PhantomData

So, sometimes we want to simulate a field of the given type for the purpose of static analysis.

In order to tell drop checker that we do own values of type `T`, and therefore may drop some `T`'s when we drop, we must add an extra `PhantomData` saying exactly that.

```
struct Vec<T> {
    data: *const T,
    len: usize,
    cap: usize,
    _marker: marker::PhantomData<T>,
}
```

`PhantomData` is ZST used to mark things that "act like" they own a `T`.

Conclusion

Conclusion

Let’s summarize our new knowledge about Rust’s type system.

Conclusion

Let's summarize our new knowledge about Rust's type system.

1. The core idea of system safety is to track object's lifetimes.

Conclusion

Let's summarize our new knowledge about Rust's type system.

1. The core idea of system safety is to track object's lifetimes.
2. Lifetimes are nested, so we need only to track local scopes and generic definitions (formal proof can be found in papers).

Conclusion

Let's summarize our new knowledge about Rust's type system.

1. The core idea of system safety is to track object's lifetimes.
2. Lifetimes are nested, so we need only to track local scopes and generic definitions (formal proof can be found in papers).
3. Every generic definition includes different lifetimes and generic types.

Conclusion

Let's summarize our new knowledge about Rust's type system.

1. The core idea of system safety is to track object's lifetimes.
2. Lifetimes are nested, so we need only to track local scopes and generic definitions (formal proof can be found in papers).
3. Every generic definition includes different lifetimes and generic types.
4. By default, you have no lifetime parameters. They usually appear in references and mean "I cannot outlive that lifetime"

Conclusion

Let's summarize our new knowledge about Rust's type system.

1. The core idea of system safety is to track object's lifetimes.
2. Lifetimes are nested, so we need only to track local scopes and generic definitions (formal proof can be found in papers).
3. Every generic definition includes different lifetimes and generic types.
4. By default, you have no lifetime parameters. They usually appear in references and mean "I cannot outlive that lifetime"
5. Or in `PhantomData` when you want not to outlive some object.

Conclusion

1. Every function, structure and enumeration have a **set** of possible input parameters. For instance:

```
fn example<'a, T, U>(t: T, u: U)
where
    T: IntoIterator,
    <T as IntoIterator>::IntoIter: Iterator<Item=U>,
    U: Clone + 'a {}
```

Conclusion

1. Every function, structure and enumeration have a **set** of possible input parameters. For instance:

```
fn example<'a, T, U>(t: T, u: U)
where
    T: IntoIterator,
    <T as IntoIterator>::IntoIter: Iterator<Item=U>,
    U: Clone + 'a {}
```

2. After that, we check whether our input types are correct. We do this with respect to the variance of this types.

Conclusion

1. Every function, structure and enumeration have a **set** of possible input parameters. For instance:

```
fn example<'a, T, U>(t: T, u: U)
where
    T: IntoIterator,
    <T as IntoIterator>::IntoIter: Iterator<Item=U>,
    U: Clone + 'a {}
```

2. After that, we check whether our input types are correct. We do this with respect to the variance of this types.
3. It can be proved that it's enough to have a memory safe application!

Conclusion

1. When Rust checks `impl` blocks for intersection, it finds whether there's an intersections in these sets of possible input parameters.

Conclusion

1. When Rust checks `impl` blocks for intersection, it finds whether there's an intersections in these sets of possible input parameters.
2. It's appears that if we have a destructors, it's not enough to require just "lives at least as" realation. Instead, we have to *strictly outlive* all of the type's parameters!

Conclusion

1. When Rust checks `impl` blocks for intersection, it finds whether there's an intersections in these sets of possible input parameters.
2. It's appears that if we have a destructors, it's not enough to require just "lives at least as" realation. Instead, we have to *strictly outlive* all of the type's parameters!
3. After all checks, Rust erases lifetimes: semantically, for different lifetimes you have different implementations, but in the stage of code generation compiler creates one implementation. So, the only purpose of lifetimes is to make strict type system that solves memory safety issues.

Conclusion

1. When Rust checks `impl` blocks for intersection, it finds whether there's an intersections in these sets of possible input parameters.
2. It's appears that if we have a destructors, it's not enough to require just "lives at least as" realation. Instead, we have to *strictly outlive* all of the type's parameters!
3. After all checks, Rust erases lifetimes: semantically, for different lifetimes you have different implementations, but in the stage of code generation compiler creates one implementation. So, the only purpose of lifetimes is to make strict type system that solves memory safety issues.
4. And the last: we've already discussed that Rust doesn't allow to coerse freely types (eg. `void*` to `int*` in C++).

Conclusion

1. When Rust checks `impl` blocks for intersection, it finds whether there's an intersections in these sets of possible input parameters.
2. It's appears that if we have a destructors, it's not enough to require just "lives at least as" realation. Instead, we have to *strictly outlive* all of the type's parameters!
3. After all checks, Rust erases lifetimes: semantically, for different lifetimes you have different implementations, but in the stage of code generation compiler creates one implementation. So, the only purpose of lifetimes is to make strict type system that solves memory safety issues.
4. And the last: we've already discussed that Rust doesn't allow to coerse freely types (eg. `void*` to `int*` in C++).
5. Some coersions are allowed, but this list is quite restrictive.