Rust Memory Management

Kenjiro Taura

2025/07/07

Contents

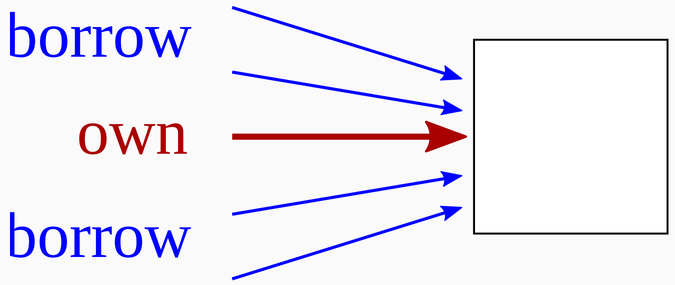
Contents

Introduction

Rust's basic idea to memory management

- Rust maintains that, for any live object,
  1. there is *one and only one pointer that "owns" it (the owning pointer)*
  2. there are any number of non-owning pointers to it *(borrowing pointers)*
  3. *borrowing pointers cannot be dereferenced after the owning pointer goes away*
- ⇒ *it can safely reclaim the data when the owning pointer goes away*

"*single-ownership rule*"



```
{
  let a = S{x: …, y: …};
  …
} // what a points to will be gone here
```

The rules are enforced statically

- Rust enforces the rules (or, detect violations thereof)
  - *statically*, not *dynamically*
  - *compile-time*, not at *runtime*
  - *before* execution, not *during* execution

“*borrow checker*”

Escaping from the single ownership model

- there are actually some ways to get around the rules

1. reference counting pointers (≈ multiple owning pointers)
   - counts the number of owners *at runtime*, and reclaim the data when all these pointers are gone
2. unsafe/raw pointers (≈ totally up to you)

they are not specific to Rust, and we'll not cover them below

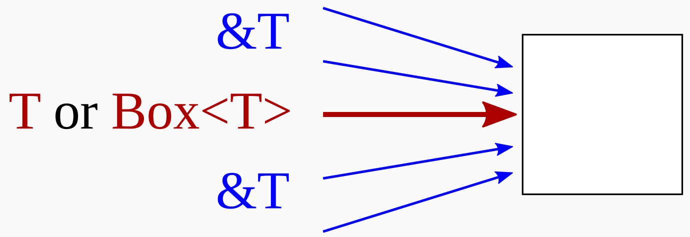
Rust Basics

Pointer-like data types in Rust

given a type T (`i32`, struct, enum, ...), below are types representing "references (pointers) to T"

1. T : owning pointer to T
2. `Box<`T`>` (pronounced "*box* T") : owning pointer to T
3. &T (pronounced "*ref* T") : borrowing pointer to T
4. `Rc<`T`>` and `Arc<`T`>` : shared (reference-counting) owning pointer to T
5. $*T$: unsafe pointer to T

following discussions are focused on
T, `Box<`T`>` *and* &T



Pointer-making expressions

given an expression e of type T, below are expressions that make pointers to the value of e (besides e itself)

- `Box::new(`e`)` (of type `Box<`T`>`) : an owning pointer
- `&`e (of type `&`T) : a borrowing pointer

An example

```
{
  let a: S = S{x: …};                // allocate memory for S
                                     // and make an owning pointer to it
  let b: S = a                       // an owning pointer
  let c: Box<S> = Box::<S>::new(a)   // an owning pointer
  let d: &S = &a                     // a borrowing pointer
}
```

- note: type of variables can be omitted (spelled out for clarity)
- note: the above program violates several rules so it does not compile

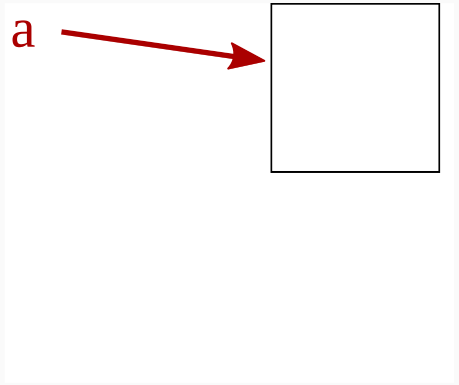
Owning Pointers

Assignments of owning pointers

- to maintain the "single-owner" rule, an assignment of owning pointers in Rust *does not copy, but moves it* out of the righthand side, disallowing further use of it

`b = a;` *// a cannot be used below*

```
fn foo() {
  let a = S{x: …, y: …};
  … a.x …; // OK, as expected
  … a.y …; // OK, as expected



}
```

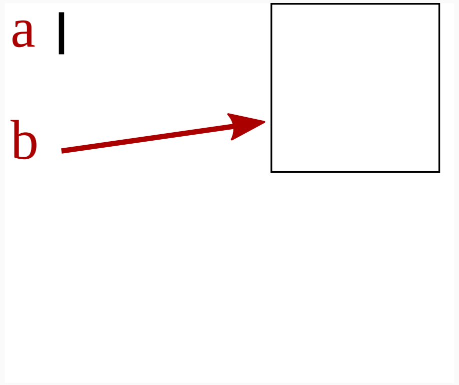


Assignments of owning pointers

- to maintain the “single-owner” rule, an assignment of owning pointers in Rust *does not copy, but moves it* out of the righthand side, disallowing further use of it

`b = a;` *// a cannot be used below*

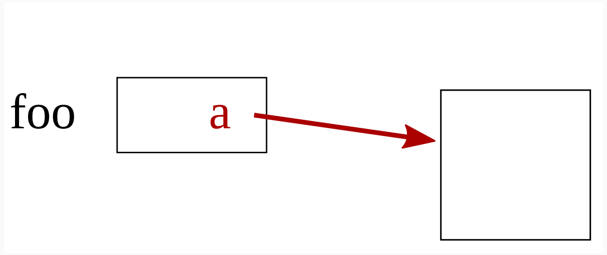
```
fn foo() {
  let a = S{x: …, y: …};
  … a.x …;  // OK, as expected
  … a.y …;  // OK, as expected
  // the reference moves out from a
  let b = a;
  a.x;  // NG, the value has moved out
  b.x;  // OK
}
```



Argument-passing also moves the reference

- passing a value to a function also moves the reference out of the source

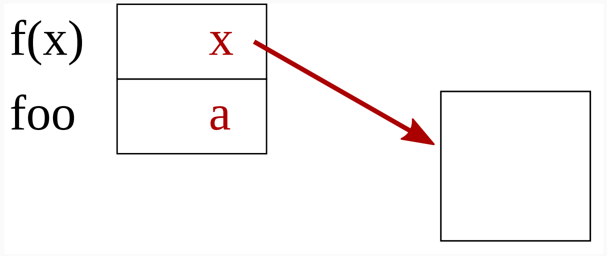
```
fn foo() {
  let a = S{x: …, y: …};
  … a.x …;  // OK, as expected
  … a.y …;  // OK, as expected




}
```



Argument-passing also moves the reference

- passing a value to a function also moves the reference out of the source

```
fn foo() {
  let a = S{x: …, y: …};
  … a.x …; // OK, as expected
  … a.y …; // OK, as expected
  // moves the reference out of a
  f(a);
  a.x; // NG, the reference has moved
}
```



Exceptions to “assignment moves the reference”

- you may notice the moving assignment contradicts what you have seen

  `b = a;` *// a cannot be used after this*

- if it applies everywhere, does the following program violate it?

```
fn foo() -> f64 {
  let a = 123.456;
  let b = a; // does the reference to 123.456 move out from a!?
  a + 0.789    // if so, is this invalid!?
}
```

- answer: no, it does *not* apply to primitive types like `i32`, `f64`, etc.
- more generally, it does not apply to data types that implement `Copy` trait

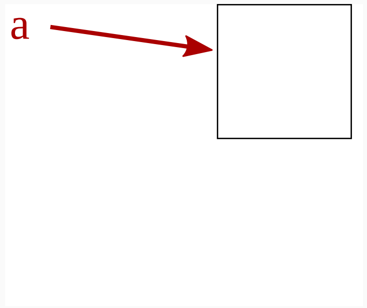
Copy trait

- define your struct with `#[derive(Copy, Clone)]` like

```
#[derive(Copy, Clone)]
struct S { … }
```

- $\Rightarrow$ assignment or argument-passing of `S` *copies* the righthand side

```
fn foo() {
  let a = S{x: …, y: …};
  a.x;  // OK, as expected
  a.y;  // OK, as expected



}
```



- note: copy types trivially maintain the single-owner rule

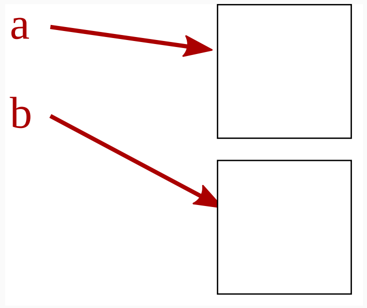
Copy trait

- define your struct with `#[derive(Copy, Clone)]` like

```
#[derive(Copy, Clone)]
struct S { … }
```

- ⇒ assignment or argument-passing of `S` *copies* the righthand side

```
fn foo() {
  let a = S{x: …, y: …};
  a.x;  // OK, as expected
  a.y;  // OK, as expected
  // the value is copied
  let b = a;
  a.x;  // OK
  b.x;  // OK, too
}
```



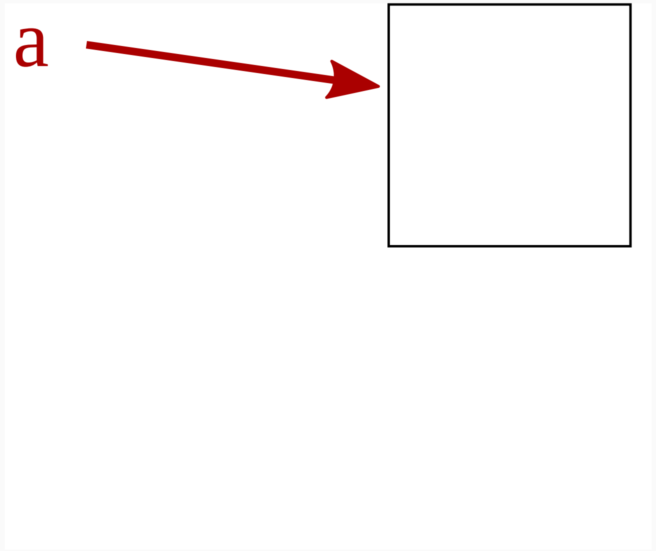
- note: copy types trivially maintain the single-owner rule

Box<T> type

Box<T> makes an owning pointer

- making a pointer by `Box::new(`v`)` moves the reference out of v, too, and `Box::new(`v`)` becomes the owning pointer

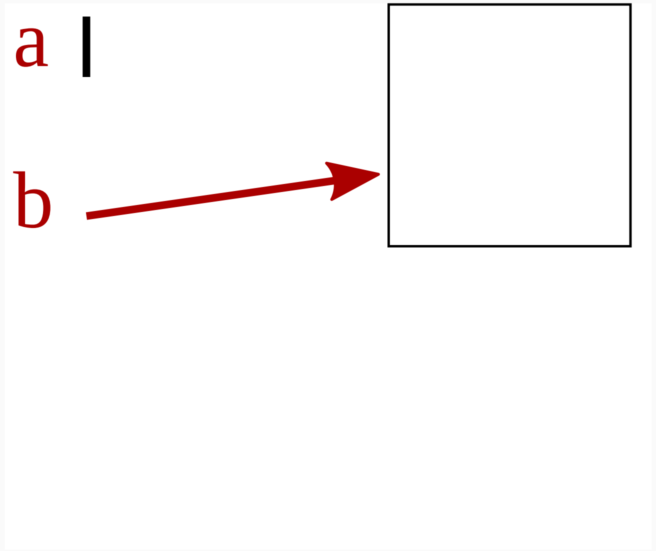
```
fn foo() {
  let a = S{x: …, y: …};
  a.x;  // OK, as expected
  a.y;  // OK, as expected



}
```



Box<T> makes an owning pointer

- making a pointer by `Box::new(`v`)` moves the reference out of v, too, and `Box::new(`v`)` becomes the owning pointer

```
fn foo() {
  let a = S{x: …, y: …};
  a.x;  // OK, as expected
  a.y;  // OK, as expected
  // OK, now b is the owning pointer
  let b = Box::new(a)
  a.x;  // NG, the value has moved out
  (*b).x;  // OK
  b.x;  // OK. abbreviation of (*b).x
}
```



Difference between T and `Box`<T>?

- as you have seen, the effects of

```
let b = a
```

and

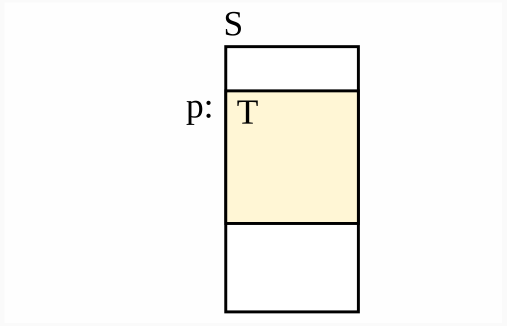
```
let b = Box::new(a)
```

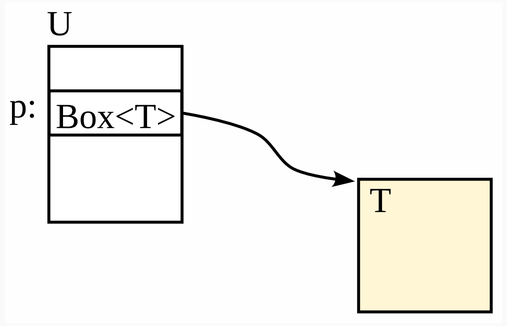
look very similar (identical)

- as far as data lifetime is concerned, it is in fact safe to say they are
- Rust has distinction between them for
  1. specifying data layout
  2. allowing dynamic dispatch only for `Box`<T>
  3. specifying where data are allocated (stack vs. heap)

Data layout differences between T and `Box<`T`>`

- `S` and `U` below have different data layouts
  - `struct S { …, p: `T`, }` "embeds" a T into `S`
  - `struct U { …, p: Box<`T`>, }` has `p` point to a separately allocated T



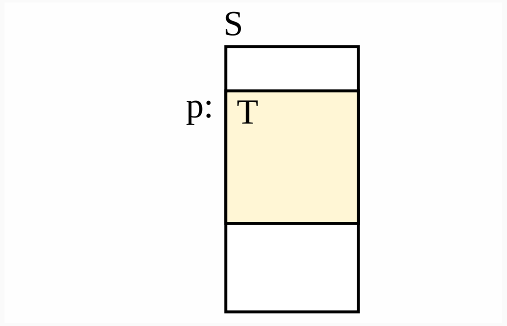


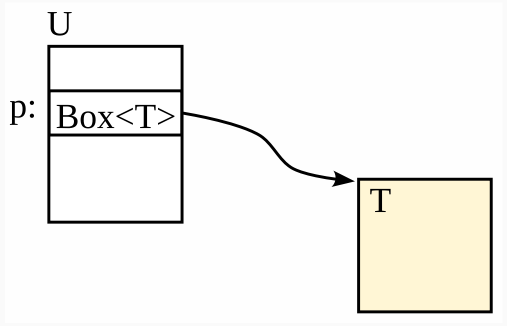
Data layout differences between T and `Box<`T`>`

- in particular, `Box<`T`>` is essential to define recursive data structures
  - `struct S { …, p: S, }` is not allowed, whereas
  - `struct U { …, p: Box<U>, }` is
- note: `U` above can never be constructed; a recursive data structure typically looks like
  - `struct U { …, p: Option<Box<U>>, }`

Data layout differences between T and `Box`<T>

- the distinction is insignificant when discussing lifetimes





- in both cases, data of T (yellow box) is gone exactly when the enclosing structure is gone
- another difference is that Rust allocates T on stack and move it to heap when `Box`<T> is made
  - but again, it has nothing to do with lifetime (unlike C/C++)

Owning pointers and control flows

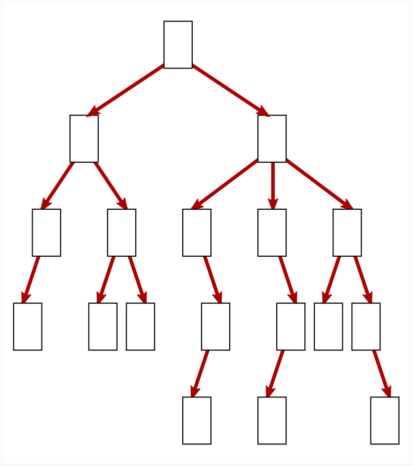
- Rust compiler determines, for each variable of owning pointer type (T or `Box`<T>), at which point the variable can be *used* (i.e., the value has not been moved out)
- it may be a *conservative* estimate

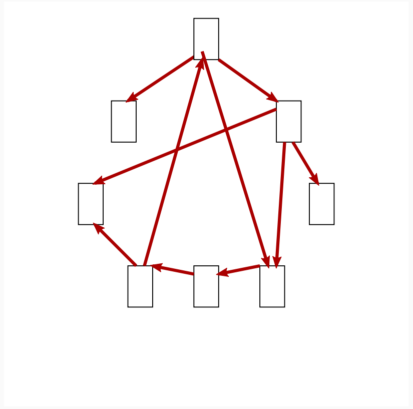
```
fn foo() {
  let a = S{x: …, y: …};
  if … {
    let b = a;
  }
  … a.x … // NG
}
```

```
fn foo() {
  let a = S{x: …, y: …};
  for … {
    let b = a; // NG
  }
}
```

A (huge) implication of the single-owner rule

- with only owning pointers (T and `Box<`T`>`),
  - you can make *a tree* of data,
  - but you *cannot make a general graph* with joins or cycles, where *a node may be pointed to by multiple nodes*
- to make a graph whose nodes are T, use either
  - &T to represent edges, or
  - `Vec<`T`>` to represent nodes and `Vec<(i32, i32)>` to represent edges

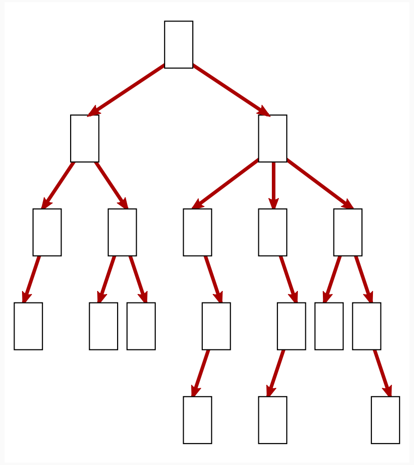




The (huge) implication to memory management

- with only owning pointers (i.e., no borrowing pointers)
- *whenever an owning pointer is gone* (e.g.,
  - a variable goes out of scope or
  - a variable or field is overwritten),

  the entire tree rooted from the pointer can be safely reclaimed



- Rust exactly does that, with the additional guarantee that *borrowing pointers are never dereferenced after its owning pointer is gone*

Borrowing pointers (&T)

Basics

- you can derive any number of borrowing pointers (&T) from T or `Box<`T`>`
- the owning pointer remains valid after a borrowing pointer has been made

```
let a = S{x: .., y: ..};
let b = &a;
… a.x + b.x … // OK
```

- the issue is how to prevent a program from *dereferencing borrowing pointers after its owning pointer is gone*

Borrowers rule in action

- a borrowing pointer cannot be dereferenced after its owning pointer is gone

```
fn foo() -> i32 {
  let c: &S; // a reference to S








}
```

Borrowers rule in action

- a borrowing pointer cannot be dereferenced after its owning pointer is gone

```
fn foo() -> i32 {
  let c: &S;  // a reference to S
  {  // an inner block
    let b: &S;  // another reference







  }

}
```

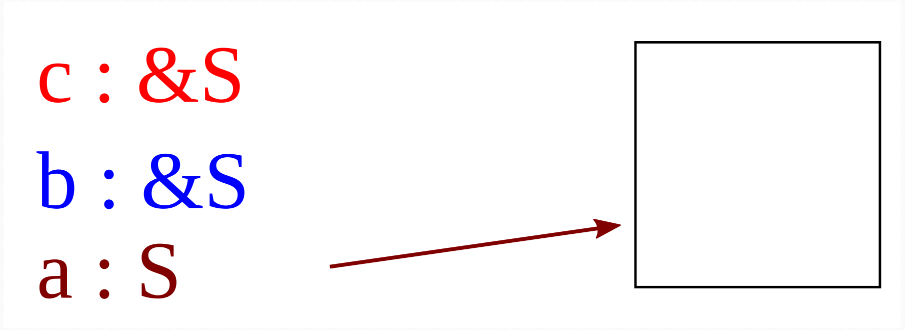
c : &S

b : &S

Borrowers rule in action

- a borrowing pointer cannot be dereferenced after its owning pointer is gone

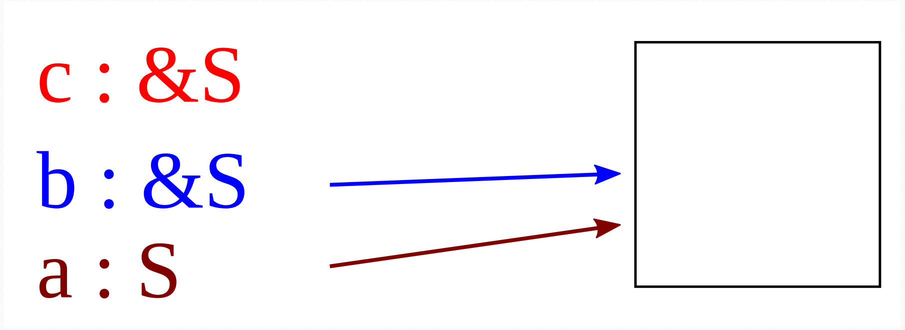
```
fn foo() -> i32 {
  let c: &S;  // a reference to S
  { // an inner block
    let b: &S;  // another reference
    let a = S{x: …};  // allocate S



  }

}
```



Borrowers rule in action

- a borrowing pointer cannot be dereferenced after its owning pointer is gone

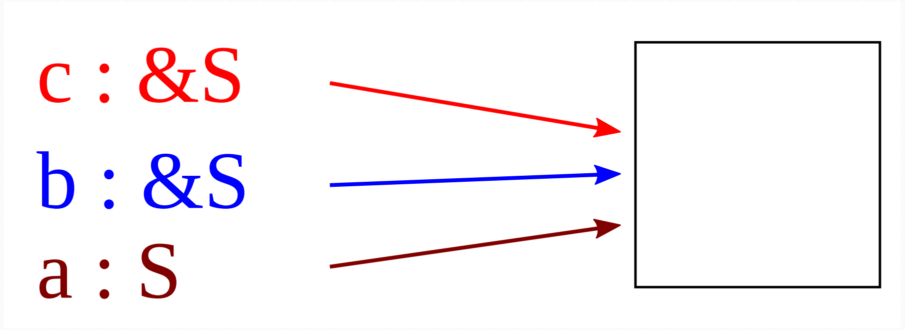
```
fn foo() -> i32 {
  let c: &S; // a reference to S
  { // an inner block
    let b: &S; // another reference
    let a = S{x: …}; // allocate S
    // OK (both a and b live only until the end of
the inner block)
    b = &a;

  }

}
```



Borrowers rule in action

- a borrowing pointer cannot be dereferenced after its owning pointer is gone

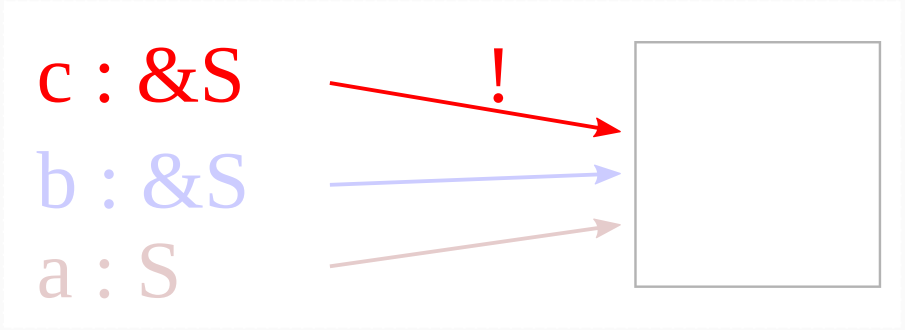
```
fn foo() -> i32 {
  let c: &S;  // a reference to S
  {  // an inner block
    let b: &S;  // another reference
    let a = S{x: …};  // allocate S
    // OK (both a and b live only until the end of the inner block)
    b = &a;
    c = b;  // dangerous (c outlives a)
  }

}
```



Borrowers rule in action

- a borrowing pointer cannot be dereferenced after its owning pointer is gone

```
fn foo() -> i32 {
  let c: &S; // a reference to S
  { // an inner block
    let b: &S; // another reference
    let a = S{x: …}; // allocate S
    // OK (both a and b live only until the end of the inner block)
    b = &a;
    c = b; // dangerous (c outlives a)
  } // a dies here, making c a dangling pointer
  c.x // NG (deref a dangling pointer)
}
```



A *mutable* borrowing reference (`&mut` T)

- data cannot be modified through ordinary borrowing references &T

```
let a : S = S{x: 10, y: 20};
let b : &S = &a;
b.x = 100; // NG
```

- i.e., &T is the type of *immutable* references
- you can modify data only through *a mutable reference* (`&mut` T)

```
let mut a : S = S{x: 10, y: 20};
let b : &mut S = &mut a;
b.x = 100; // OK
```

- the difference is largely orthogonal to memory management

Borrow-checking details

A technical remark about the borrow-checking

- it’s *not* dangling pointers, *per se*, that are prevented, but their *dereferencing*
- *the previous code compiles* as long as c is not dereferenced

```
fn foo() -> i32 {
  let c: &S; // a reference to S
  { // an inner block
    let b: &S; // another reference
    let a = S{x: …}; // allocate S
    // OK (both a and b live only until the end of the inner block)
    b = &a;
    c = b; // dangerous (c outlives a)
  } // a dies here, making c a dangling pointer
  // c.x don’t deref c
}
```

How borrow-checking works : lifetime

- *lifetime* of data
  - $=$ program points where the data has not been deallocated
  - $=$ program points where the data's owning pointer is valid
- for each borrowing pointer, Rust compiler determines the *lifetime* of data it points to *(referent lifetime)* as its static type
- upon assignment $p = q$ between borrowing pointers, it demands

$$\text{referent lifetime of } p \subset \text{referent lifetime of } q$$

How borrow-checking basically works

```
fn foo() -> i32 {
  let c: &S;
  {
    let b: &S;
    let a = S{x: …}; // lives until α


  }

}
```

1. the owning pointer `a`'s lifetime is the inner block; call it α ($\boxed{\ldots}$)
2. let β and γ be referent lifetimes of `b` and `c`, respectively

How borrow-checking basically works

```
fn foo() -> i32 {
  let c: &S;
  {
    let b: &S;
    let a = S{x: …}; // lives until α
    b = &a;  // b's referent lifetime ⊂ a's = α
    c = b;  // c's referent lifetime ⊂ b's = α
  }                                              … α

}
```

1. the owning pointer `a`'s lifetime is the inner block; call it α ($\boxed{\ldots}$)
2. let β and γ be referent lifetimes of `b` and `c`, respectively
3. due to the assignments,
   - `b = &a` $\Rightarrow \beta \subset \alpha$
   - `c = b` $\Rightarrow \gamma \subset \beta \; (\subset \alpha)$

How borrow-checking basically works

```
fn foo() -> i32 {
  let c: &S;
  {
    let b: &S;
    let a = S{x: …}; // lives until α
    b = &a; // b’s referent lifetime ⊂ a’s = α
    c = b;  // c’s referent lifetime ⊂ b’s = α    … α
  }
  c.x // NG (deref outside c’s referent lifetime = α)
}
```

1. the owning pointer `a`’s lifetime is the inner block; call it α ($\boxed{\ldots}$)
2. let β and γ be referent lifetimes of `b` and `c`, respectively
3. due to the assignments,
   - `b = &a` $\Rightarrow \beta \subset \alpha$
   - `c = b` $\Rightarrow \gamma \subset \beta \; (\subset \alpha)$
4. dereference `c.x` must be $\subset \gamma \; (\subset \alpha)$, which is not the case (i.e., invalid)

Programming with borrowing references

- in more general cases, programs using borrowing references must help compilers track their referent lifetimes
- this must be done for functions called from unknown places, function calls to unknown functions and data structures
- to this end, the programmer sometimes must annotate *reference types with their referent lifetimes*

References in function calls

- how to check the validity of a functions call without knowing its body?

```
{
  let r : &i32;
  let a = 123;
  {
    let b = 456;
    {
      let c = 789;
      r = foo(&a, &b, &c);  … γ
    }                          … β
  }
  *r // (†)                      … α
}
```

  - *r should be safe if `f(p, q, r)` returns a reference whose referent lifetime contains (†); i.e., `p`

References in data structures

- how to check the validity of dereferencing references obtained from a data structure

```
struct A { b : &B }
struct B { c : &C }
struct C { x : i32 }
  …
  let c = C{x : 123};
  let b = B{c : &c};
  let mut a = A{b : &b};




  a.b.c.x // OK?
```

References in data structures

- how to check the validity of dereferencing references obtained from a data structure

```
struct A { b : &B }
struct B { c : &C }
struct C { x : i32 }
  …
  let c = C{x : 123};
  let b = B{c : &c};
  let mut a = A{b : &b};
  {
    let b2 = B{c : &c};
    a.b = &b2;
  }
  a.b.c.x // OK?
```

References in function parameters

- how to check the validity of functions taking references or structures containing references, *without knowing all its callers*

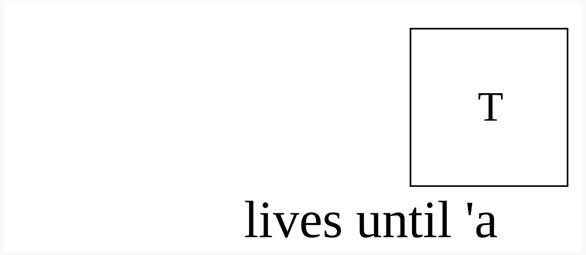
```
fn bar(a : &mut &i32, b : &i32) {
  *a = b;
}
```

- what if references are in structures …

```
fn baz(a : &mut A, b: &B) {
  a.b = b
}
```

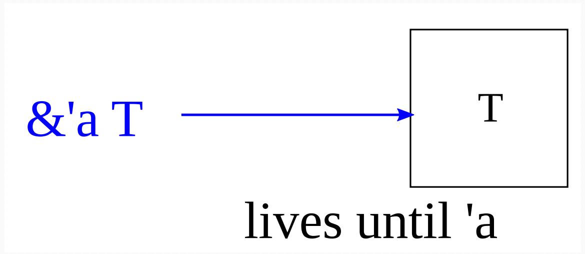
Reference type with a lifetime parameter

- to address these problems, Rust's borrowing reference types ($\&T$ or `&mut` T) carry *lifetime parameter representing their referent lifetimes*



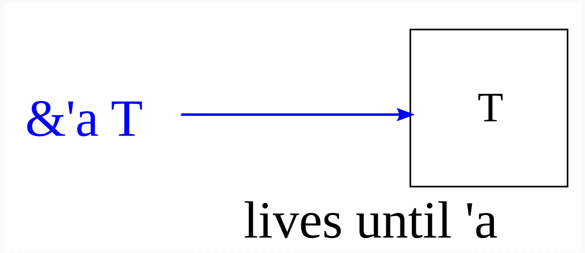
Reference type with a lifetime parameter

- to address these problems, Rust's borrowing reference types (`&`T or `&mut` T) carry *lifetime parameter representing their referent lifetimes*
- syntax:
  - `&'`a T : reference to "T whose lifetime is `'`a"
  - `&'`a `mut` T : ditto; except you can modify data through it



Reference type with a lifetime parameter

- to address these problems, Rust's borrowing reference types (`&`T or `&mut` T) carry *lifetime parameter representing their referent lifetimes*
- syntax:
  - `&'`a T : reference to "T whose lifetime is `'`a"
  - `&'`a `mut` T : ditto; except you can modify data through it
- *every* reference carries a lifetime parameter, though there are places you can omit them
- roughly, you must write them explicitly in function parameters, return types, and struct/enum fields; and can omit them for local variables



Attaching lifetime parameters

- rule: reference types that appear in function parameters, return types, and struct/enum fields must have explicit lifetime paramters

Attaching lifetime parameters to functions

- therefore the following does not compile:

```
fn foo(ra: &i32, rb: &i32, rc: &i32) -> &i32 {
  ra
}
```

with errors like:

```
|
| fn foo(ra: &i32, rb: &i32, rc: &i32) -> &i32 {
|            ----      ----      ----     ^ expected named lifetime parameter
|
= help: this function's return type contains a borrowed value, but the signature does not say
whether it is borrowed from `ra`, `rb`, or `rc`
help: consider introducing a named lifetime parameter
|
| fn foo<'a>(ra: &'a i32, rb: &'a i32, rc: &'a i32) -> &'a i32 {
|       ++++     ++           ++           ++          ++
```

Why do we need an annotation, *fundamentally*?

- without any annotation, how to know whether this is safe, *without knowing the body of `foo`?*

```
{
  let r : &i32;
  let a = 123;
  {
    let b = 456;
    {
      let c = 789;
      r = foo(&a, &b, &c);
    }
  }
  *r
}
```

- essentially, the compiler complains “tell me what kind of referent lifetime the reference returned by `foo(&a, &b, &c)` has”
- it must be inferred without knowing the body of `foo`, only from its type

Attaching lifetime parameters

- functions

  fn f<'a,'b,'c,...> (p_0 : T_0, p_1 : T_1, …) -> T_r { … }

- structs/enums

  ```
  struct A<'a,'b,'c,...> {
    f0 : T0,
    f1 : T1,
    …
  }
  ```

- T_0, T_1, …, and T_r may use `'a`, `'b`, `'c`, … as lifetime parameters (e.g., `&'a i32`)

One way to attach lifetime parameters to the example

```
fn foo<'a>(ra: &'a i32, rb: &'a i32, rc: &'a i32) -> &'a i32
```

- effect: the return value is assumed to point to the shortest of the three
- why? generally, when Rust compiler finds `foo(x, y, z)`, it tries to determine `'a` so that `'a` $\subset$ referent lifetimes of `x, y`, and `z`
- in this case,

`'a` $\subset$ (life time of `a`) $\cap$ (life time of `b`) $\cap$ (life time of `c`) $=$ life time of `c`

- as a result, our program does not compile, even if `foo(&a,&b,&c)` in fact returns &a

```
{
  let r: &i32;
  let a = 123;
  {
    let b = 456;
    {
      let c = 789;
      r = foo(&a, &b, &c);
      // 'a ← α ∩ β ∩ γ = γ
      // and r's type becomes &γ i32
    } // c's lifetime (= γ) ends here
  } // b's lifetime (= β) ends here
  *r // NG, as we are outside γ
} // a's lifetime (= α) ends here
```

An annotation that works

```
fn foo<'a,'b,'c>(ra: &'a i32, rb: &'b i32, rc: &'c i32)->&'a i32
```

- signifies that the return value points to data whose lifetime is `ra`'s referent lifetime (and has nothing to do with `rb`'s or `rc`'s)
- for `foo(x, y, z)`, Rust compiler tries to determine `'a` so that `'a` $\subset$ referent lifetimes of `x`
- as a result, the program we are discussing compiles

```
{
  let r: &i32;
  let a = 123;
  {
    let b = 456;
    {
      let c = 789;
      r = foo(&a, &b, &c);
      // 'a ← α
      // and r's type becomes &α i32
    }  // c's lifetime (= γ) ends here
  }  // b's lifetime (= β) ends here
  *r  // OK, as here is within α
}  // a's lifetime (= α) ends here
```

Types with lifetime parameters capture/constrain the function's behavior

- what if you try to fool the compiler by:

```
fn foo<'a,'b,'c>(ra: &'a i32, rb: &'b i32, rc: &'c i32) -> &'a i32 {
  rb
}
```

- the compiler rejects returning `rb` (of type `&'b`) when the function's return type is `&'a`, as it cannot infer

  lifetime represented by `'a` $\subset$ lifetime represented by `'b`

References in data structures

```
struct A { b : &B }
struct B { c : &C }
struct C { x : i32 }
```

$\Rightarrow$

```
struct C { x : i32 }
```

```
fn baz(a : &mut A, b: &B) {
  a.b = b
}
```

does not compile

References in data structures

```
struct A { b : &B }
struct B { c : &C }
struct C { x : i32 }
```

$\Rightarrow$

```
struct B<'c> { c : &'c C }
struct C { x : i32 }
```

```
fn baz(a : &mut A, b: &B) {
  a.b = b
}
```

does not compile

References in data structures

```
struct A { b : &B }
struct B { c : &C }
struct C { x : i32 }
```

$\Rightarrow$

```
struct A<'b,'c> { b : &'b B<'c> }
struct B<'c> { c : &'c C }
struct C { x : i32 }
```

```
fn baz(a : &mut A, b: &B) {
  a.b = b
}
```

does not compile

References in data structures

```
struct A { b : &B }
struct B { c : &C }
struct C { x : i32 }

fn baz(a : &mut A, b: &B) {
  a.b = b
}
```

does not compile

$\Rightarrow$

```
struct A<'b,'c> { b : &'b B<'c> }
struct B<'c> { c : &'c C }
struct C { x : i32 }
fn baz<'a,'b,'c','d,'e> (a : &'a mut A<'b,'c>,
                         b: &'d B<'e>) {
  a.b = b
}
```

References in data structures

```
struct A { b : &B }
struct B { c : &C }
struct C { x : i32 }

fn baz(a : &mut A, b: &B) {
  a.b = b
}
```

does not compile

$\Rightarrow$

```
struct A<'b,'c> { b : &'b B<'c> }
struct B<'c> { c : &'c C }
struct C { x : i32 }
fn baz<'a,'b,'c','d,'e> (a : &'a mut A<'b,'c>,
                         b: &'d B<'e>) {
  a.b = b
}
```

does not compile

References in data structures

```
struct A { b : &B }
struct B { c : &C }
struct C { x : i32 }

fn baz(a : &mut A, b: &B) {
  a.b = b
}
```

does not compile

$\Rightarrow$

```
struct A<'b,'c> { b : &'b B<'c> }
struct B<'c> { c : &'c C }
struct C { x : i32 }

fn baz<'a,'b,'c'>(a : &'a mut A<'b,'c>,
                  b: &'b B<'c>) {
  a.b = b
}
```

References in data structures

```
struct A { b : &B }
struct B { c : &C }
struct C { x : i32 }

fn baz(a : &mut A, b: &B) {
  a.b = b
}
```

does not compile

$\Rightarrow$

```
struct A<'b,'c> { b : &'b B<'c> }
struct B<'c> { c : &'c C }
struct C { x : i32 }

fn baz<'a,'b,'c'>(a : &'a mut A<'b,'c>,
                  b: &'b B<'c>) {
  a.b = b
}
```

does compile

Dereferencing data structure

- as stated earlier, dereferencing a borrowing pointer of type `&'a ...` is allowed at program point p when:

$$p \subset \text{lifetime represented by } \texttt{'a}$$

- the rule is actually more strict; for types involving lifetime parameters (e.g., `A<'a,'b,'c,...>`), the above applies to *all* parameters

Dereferencing data structure

- the following program is *safe*, but rejected by the compiler

```
struct S<'a,'b> {
  a : &'a i32,
  b : &'b i32,
}
  …
  let a = 123;
  let mut s = S{a: &a, b: &a};
  {
    let b = 456;
    s.b = &b;
  }
```

// s.b is a dangling pointer, but s.a is not

```
  *s.a … (†)
```

Dereferencing data structure

- the following program is *safe*, but rejected by the compiler

```
struct S<'a,'b> {
  a : &'a i32,
  b : &'b i32,
}
  …
  let a = 123;
  let mut s = S{a: &a, b: &a};
  {
    let b = 456;
    s.b = &b;
  }                   … β
  // s.b is a dangling pointer, but s.a is not
  *s.a … (†)
```

```
error[E0597]: `b` does not live long enough
  --> str.rs:11:15
   |
10 |          let b = 456;
   |              - binding `b` declared here
11 |          s.b = &b;
   |                ^^ borrowed value does not live long enough
12 |      }
   |      - `b` dropped here while still borrowed
13 |      *s.a
   |      ---- borrow later used here
```

Dereferencing data structure

- the following program is *safe*, but rejected by the compiler

```
struct S<'a,'b> {
  a : &'a i32,
  b : &'b i32,
}
  …
  let a = 123;
  let mut s = S{a: &a, b: &a};
  {
    let b = 456;
    s.b = &b;
  }                 … β
  // s.b is a dangling pointer, but s.a is not
  *s.a … (†)
```

```
error[E0597]: `b` does not live long enough
  --> str.rs:11:15
   |
10 |         let b = 456;
   |             - binding `b` declared here
11 |         s.b = &b;
   |               ^^ borrowed value does not live long enough
12 |     }
   |     - `b` dropped here while still borrowed
13 |     *s.a
   |     ---- borrow later used here
```

- `s.a` is not allowed, because:
  - the type of s is `S<'a,'b>` and
  - `'b` $\subset \beta$ ($\because$ `s.b = &b`);
  - $\therefore \dagger \notin$ `'b`

Lifetime parameters in a function

- because of this restriction, the compiler can assume all lifetime parameters that appear in the function parameters contain the function body
- the compiler deduces dereferencing `a.b` below is safe based on this assumption

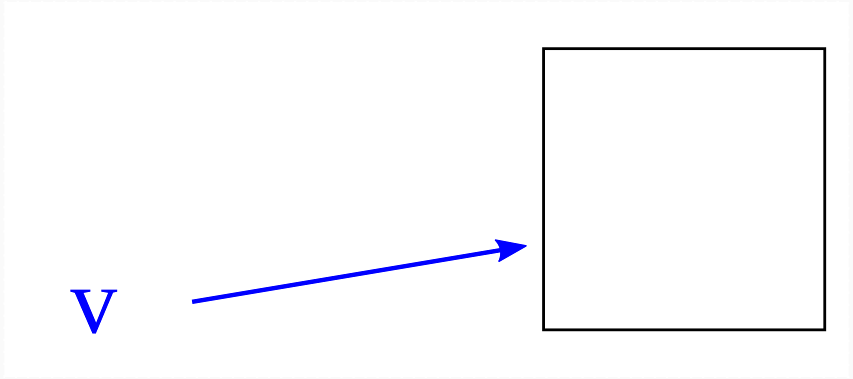
```
fn baz<'a,'b,'c'>(a : &'a mut A<'b,'c>,
                  b: &'b B<'c>) {
  a.b = b
}
```

Summary

Why memory management is difficult

- *every* language wants to prevent *dereferencing a pointer to an already-reclaimed memory block (dangling pointer)*
- the problem would have been trivial if *you could reclaim v's referent as soon as v goes out of scope*
- this is not the case, as *v's referent may still be reachable from other variables when v goes out of scope*

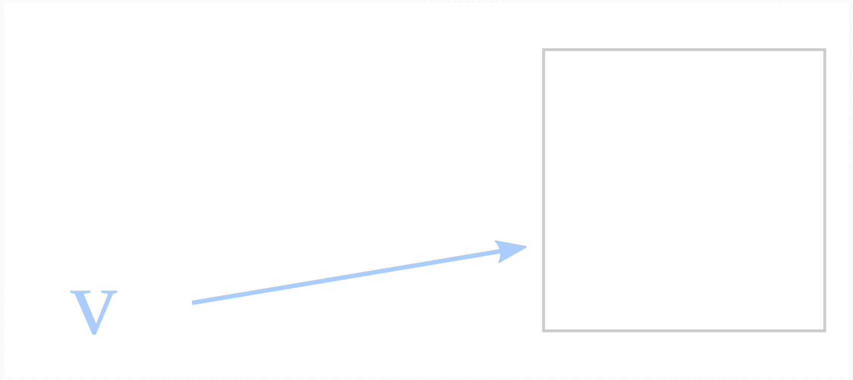
```
let p : &T;
{
  let v = T{x: …};
  …
  p = &v;
} // v never used below, but its referent is
… p.x …
```



Why memory management is difficult

- *every* language wants to prevent *dereferencing a pointer to an already-reclaimed memory block (dangling pointer)*
- the problem would have been trivial if *you could reclaim v's referent as soon as v goes out of scope*
- this is not the case, as *v's referent may still be reachable from other variables when v goes out of scope*

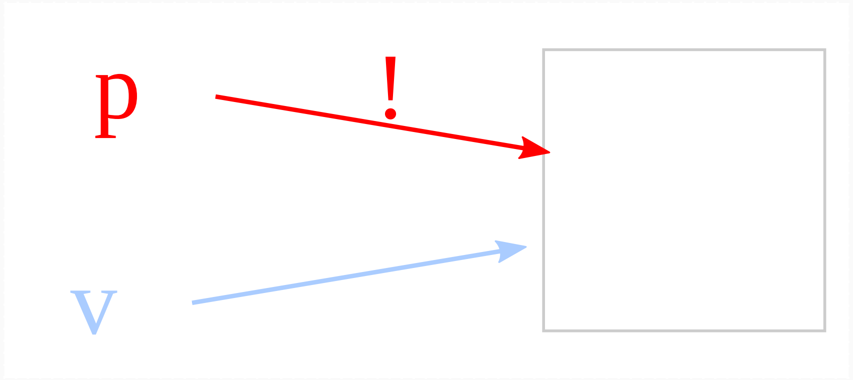
```
let p : &T;
{
  let v = T{x: …};
  …
  p = &v;
} // v never used below, but its referent is
… p.x …
```



Why memory management is difficult

- *every* language wants to prevent *dereferencing a pointer to an already-reclaimed memory block (dangling pointer)*
- the problem would have been trivial if *you could reclaim v's referent as soon as v goes out of scope*
- this is not the case, as *v's referent may still be reachable from other variables when v goes out of scope*

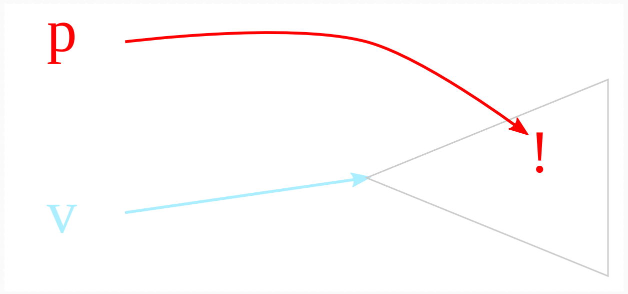
```
let p : &T;
{
  let v = T{x: …};
  …
  p = &v;
} // v never used below, but its referent is
… p.x …
```



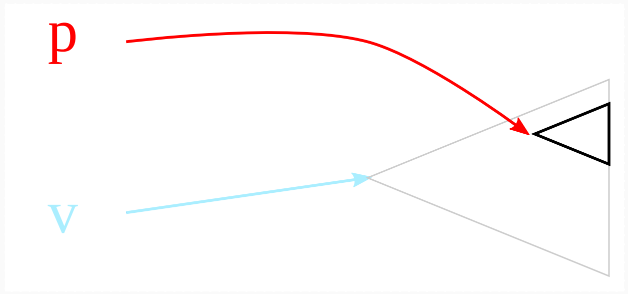
C vs. GC vs. Rust

- C/C++ : it's up to you
- GC : if it is reachable from other variables, I retain it for you
- Rust : when v goes out of scope,
  1. I reclaim T_v, all data *reachable from v through owning pointers*
  2. T_v may be reachable from other variables via borrowing references, but I guarantee such references are never dereferenced

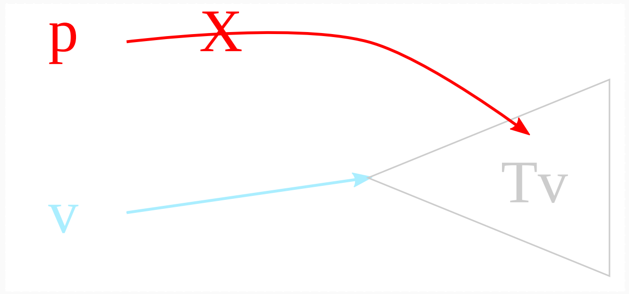
C/C++



GC

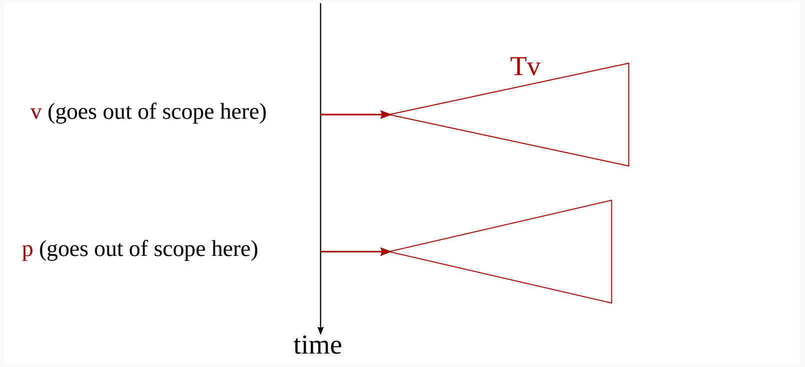


Rust



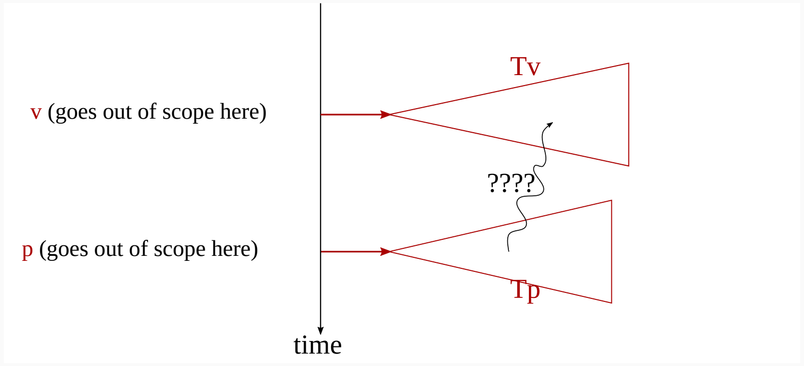
How Rust achieves it?

- say two data structures T_v rooted at variable `v` and T_p rooted at variable `p`
- assume `v` goes out of scope earlier than `p`
- we wish to guarantee when `v` goes out of scope, it is safe to reclaim the entire T_v
- generally it is of course not the case, as there may be pointers somewhere in $T_p \rightarrow$ somewhere in T_v



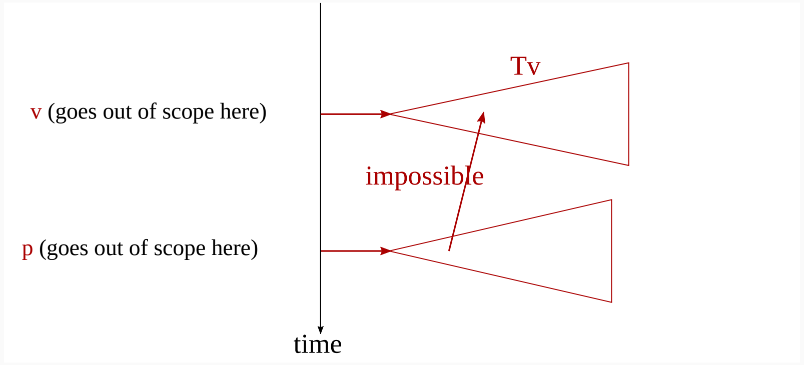
How Rust achieves it?

- say two data structures T_v rooted at variable `v` and T_p rooted at variable `p`
- assume `v` goes out of scope earlier than `p`
- we wish to guarantee when `v` goes out of scope, it is safe to reclaim the entire T_v
- generally it is of course not the case, as there may be pointers somewhere in $T_p \rightarrow$ somewhere in T_v



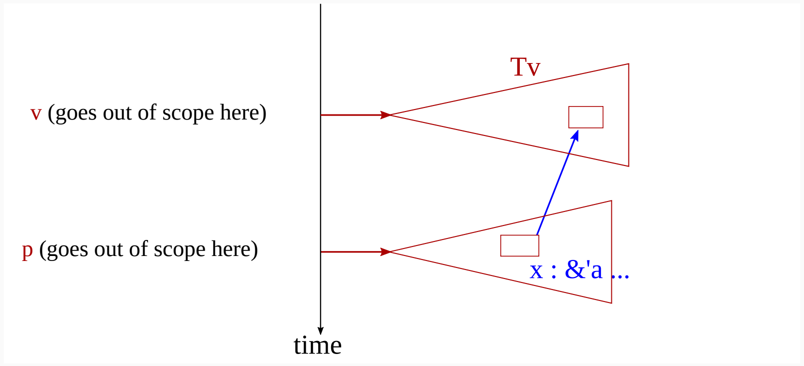
How Rust achieves it?

- recall the “single-owner rule,” which guarantees there is only one owning pointer to any node
- $\Rightarrow$ there can be *no owning pointers* from outside T_v to inside T_v



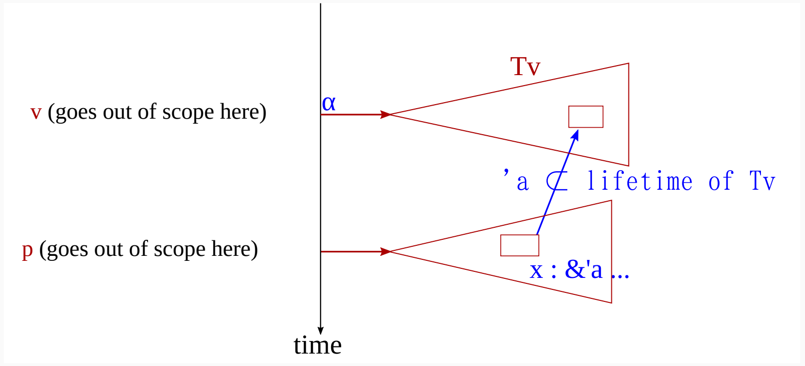
How Rust achieves it?

- recall the "single-owner rule," which guarantees there is only one owning pointer to any node
- $\Rightarrow$ there can be *no owning pointers* from outside T_v to inside T_v
- $\Rightarrow$ any such pointer must be *a borrowing pointer*



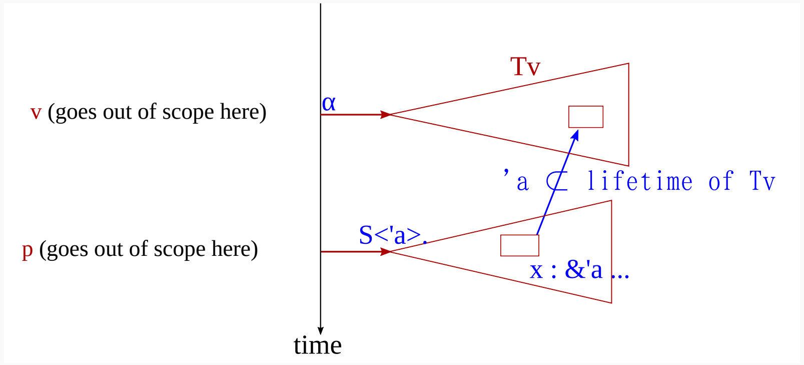
How Rust achieves it?

- recall the "single-owner rule," which guarantees there is only one owning pointer to any node
- $\Rightarrow$ there can be *no owning pointers* from outside T_v to inside T_v
- $\Rightarrow$ any such pointer must be *a borrowing pointer*
- recall that a borrowing pointer must have a lifetime parameter; e.g., `'a`
- it must hold that `'a` $\subset$ lifetime of T_v



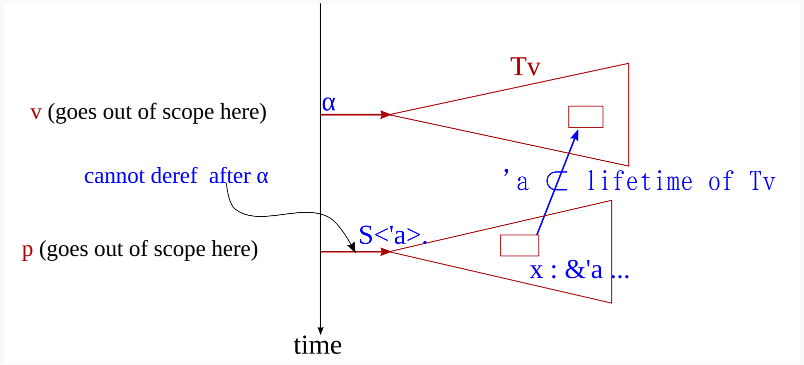
How Rust achieves it?

- any structure containing borrowing pointers must have these parameters as part of its type (e.g., `S<'a>`)
- by `'a` $\subset$ lifetime of T_v, the containing data structure (of type `S<'a>`) cannot be dereferenced after T_v is gone



How Rust achieves it?

- any structure containing borrowing pointers must have these parameters as part of its type (e.g., `S<'a>`)
- by `'a` $\subset$ lifetime of T_v, the containing data structure (of type `S<'a>`) cannot be dereferenced after T_v is gone


Tv
v (goes out of scope here)
α
cannot deref after α
'a ⊄ lifetime of Tv
S<'a>
p (goes out of scope here)
x : &'a ...
time