Stacked Borrows:
An Aliasing Model for Rust (Work in Progress)

Ralf Jung, Hai Dang, and Derek Dreyer
MPI-SWS, Germany

WG2.8, Bordeaux, May 2019

Rust – Mozilla's replacement for C/C++

A safe & modern systems PL

Rust – Mozilla's replacement for C/C++

A safe & modern systems PL

- First-class functions
- Polymorphism/generics
- Algebraic datatypes
- Traits $\approx$ Type classes

Rust – Mozilla's replacement for C/C++

A safe & modern systems PL

- First-class functions
- Polymorphism/generics
- Algebraic datatypes
- Traits $\approx$ Type classes
- Control over resource management
  (e.g., memory allocation and data layout)

Rust – Mozilla's replacement for C/C++

A safe & modern systems PL

- First-class functions
- Polymorphism/generics
- Algebraic datatypes
- Traits ≈ Type classes
- Control over resource management (e.g., memory allocation and data layout)
- Strong type system guarantees:
  - Type & memory safety; data-race freedom

Rust – Mozilla's replacement for C/C++

Goal of RustBelt project:
Prove safety of Rust and its standard library.

Rust – Mozilla's replacement for C/C++

Goal of RustBelt project:
Prove safety of Rust and its standard library.
That requires defining what Rust is!

Rust type system

1-slide summary

Rust type system

1-slide summary

Rust enforces this via ownership & borrowing:

1. Full ownership: `T`
   + mutation, deallocation
   - aliasing
2. Mutable reference: `&mut T`
   (borrowed)
   + mutation
   - aliasing, deallocation
3. Shared reference: `&T`
   (borrowed)
   + aliasing
   - mutation, deallocation

Rust enforces this via ownership & borrowing:

1. Full ownership: `T`
   + mutation, deallocation

Rust's reference types provide strong aliasing information.

The optimizer should exploit that!

3. Shared reference: `&T`
   (borrowed)
   + aliasing
   - mutation, deallocation

Aliasing guarantees: Example #1

```
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  // unknown_function_1 cannot have an alias to x
  unknown_function_1();
  return *x; // must return 42
}
```

Aliasing guarantees: Example #2

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

Aliasing guarantees: Example #3

```
fn test_shared(x: &i32) -> bool {
  let val = *x;
  // unknown_function_2 cannot mutate x
  unknown_function_2(x);
  return *x == val; // must return true
}
```

Rust enforces this via ownership & borrowing:

1. Full ownership: T

Rust's reference types provide strong aliasing information.

The optimizer should exploit that.

+ aliasing

- mutation, deallocation

Rust enforces this via ownership & borrowing:

1. Full ownership: T

Rust's reference types provide strong aliasing information.

The optimizer should exploit that.

But there is a problem:

+ aliasing

- mutation, deallocation

Rust enforces this via ownership & borrowing:

1. Full ownership: T

Rust's reference types provide strong aliasing information.

The optimizer should exploit that.

But there is a problem:

UNSAFE CODE!

+ aliasing

- mutation, deallocation

Unsafe code can access hazardous operations that are banned in safe code:

```
unsafe fn hazardous(x: usize) -> i32 {
  // *const T is the type of raw (unsafe) pointers
  let x_ptr = x as *const i32;
  return *x_ptr; // dereferencing an arbitrary integer
}
```

- Used for better performance, FFI, implementing many standard library types
- Generally encapsulated by safe APIs

```
11: fn test_unique(x: &mut i32) -> i32 {
12:   *x = 42;
13:   unknown_function_1();
14:   return *x;
15: }
```

```
 1: static mut ALIAS: *mut i32 = std::ptr::null_mut();
 2: fn main() {
 3:   let l = &mut 0;
 4:   unsafe { ALIAS = l as *mut i32; }
 5:   println!("The answer is {}", test_unique(&mut *l));
 6:   // prints: The answer is 7
 7: }
 8: fn unknown_function_1() {
 9:   unsafe { *ALIAS = 7; }
10: }
11: fn test_unique(x: &mut i32) -> i32 {
12:   *x = 42;
13:   unknown_function_1();
14:   return *x;
15: }
```

```
 1: static mut ALIAS: *mut i32 = std::ptr::null_mut();
 2: fn main() {
 3:   let l = &mut 0;
 4:   unsafe { ALIAS = l as *mut i32; }
 5:   println!("The answer is {}", test_unique(&mut *l));
 6:   // prints: The answer is 7
 7: }
 8: fn unknown_function_1() {
 9:   unsafe { *ALIAS = 7; }
10: }
11: fn test_unique(x: &mut i32) -> i32 {
12:   *x = 42;
13:   unknown_function_1();
14:   return *x;
15: }
```

ALIAS is a raw pointer (*mut T)

ALIAS and x point to the same location

Overwrites *x with 7

```
1: static mut ALIAS: *mut i32 = std::ptr::null_mut();
2: fn main() {
3:   let l = &mut 0;
4:   unsafe { ALIAS = l as *mut i32; }
5:   println!("The answer is {}", test_unique(&mut *l));
6:   /
7: }
8: fn
9:   u
10: }
11: fn
12:   *x = 42;
13:   unknown_function_1();
14:   return *x;
15: }
```

ALIAS is a raw pointer (*mut T)

int

Overwrites *x with 7

Mutating *x inside unknown_function_1 must be UB to justify the optimization.

Stacked Borrows

Work-in-progress aliasing model defining which pointers may be used to access memory, ensuring:

- Uniqueness of mutable references
- Immutability of shared references

Stacked Borrows

Implemented in Miri, an experimental interpreter for Rust's MIR

`https://github.com/rust-lang/miri`

Already uncovered 9 bugs in Rust's standard library!

Stacked Borrows

Work-in-progress aliasing model defining which pointers may be used to access memory, ensuring:

- Uniqueness of mutable references
- Immutability of shared references

Stacked Borrows

Work-in-progress aliasing model defining which pointers may be used to access memory, ensuring:

- Uniqueness of mutable references
- Immutability of shared references

```
1: let mut l = 0;
2: let a = &mut l;
```

```
1: let mut l = 0;
2: let a = &mut l;
3: let b = &mut *a;
```

```
1: let mut l = 0;
2: let a = &mut l;
3: let b = &mut *a;
4: *b = 3;
```

```
1: let mut l = 0;
2: let a = &mut l;
3: let b = &mut *a;
4: *b = 3;
5: *a = 4;
```

```
1: let mut l = 0;
2: let a = &mut l;
3: let b = &mut *a;
4: *b = 3;
5: *a = 4;
6: *b = 4; // ERROR: lifetime of ‘b‘ has ended
```

```
1: let mut l = 0;
2: let a = &mut l;
3: let b = &mut *a;
4: *b = 3;
5: *a = 4;
6: *b = 4; // ERROR: lifetime of ‘b‘ has ended
```

```
1: let mut l = 0;
2: let a = &mut l;
3: let b = &mut *a;
4: *b = 3;
5: *a = 4;
6: *b = 4; // ERROR: lifetime of ‘b‘ has ended
```

- Chain of borrows:
  `l` borrowed to `a` reborrowed to `b`
- Well-bracketed: no ABAB

```
1: let mut l = 0;
2: let a = &mut l;
3: let b = &mut *a;
4: *b = 3;
5:
6:
```

(Re)borrows are organized in a stack.

- Chain of borrows:
  `l` borrowed to `a` reborrowed to `b`
- Well-bracketed: no ABAB

Stacked Borrows ingredients

Pointer values carry a tag: $(\mathit{PtrVal} := \mathit{Loc} \times \mathbb{N}_{\perp})$

`0x40[1], 0x41[⊥]`

references (`&mut T`) are identified by a tag

raw pointers (`*mut T`) fall back to untagged

Stacked Borrows ingredients

Pointer values carry a tag: ($\mathit{PtrVal} := \mathit{Loc} \times \mathbb{N}_{\perp}$)

`0x40[1], 0x41[⊥]`

Every location in memory comes with an associated stack: ($\mathit{Mem} := \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Byte} \times \mathit{Stack}$)

⋮

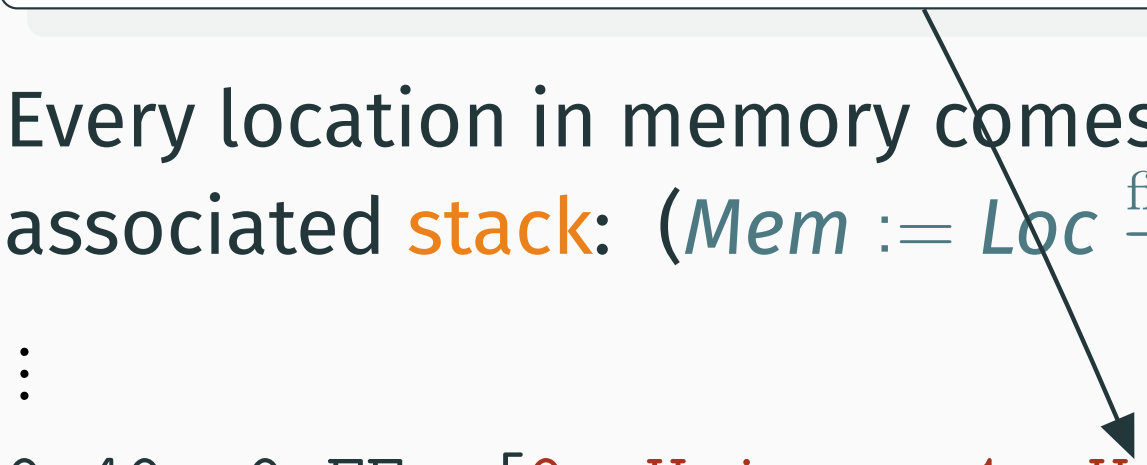
```
0x40: 0xFE, [0: Unique, 1: Unique]
0x41: 0xFE, [0: Unique, ⊥: SharedRW]
```

⋮

Stacked Borrows ingredients

Reference tagged `1` borrows from reference tagged `0`

Every location in memory comes with an associated stack: $(\mathit{Mem} := \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Byte} \times \mathit{Stack})$



⋮

`0x40: 0xFE, [0: Unique, 1: Unique]`

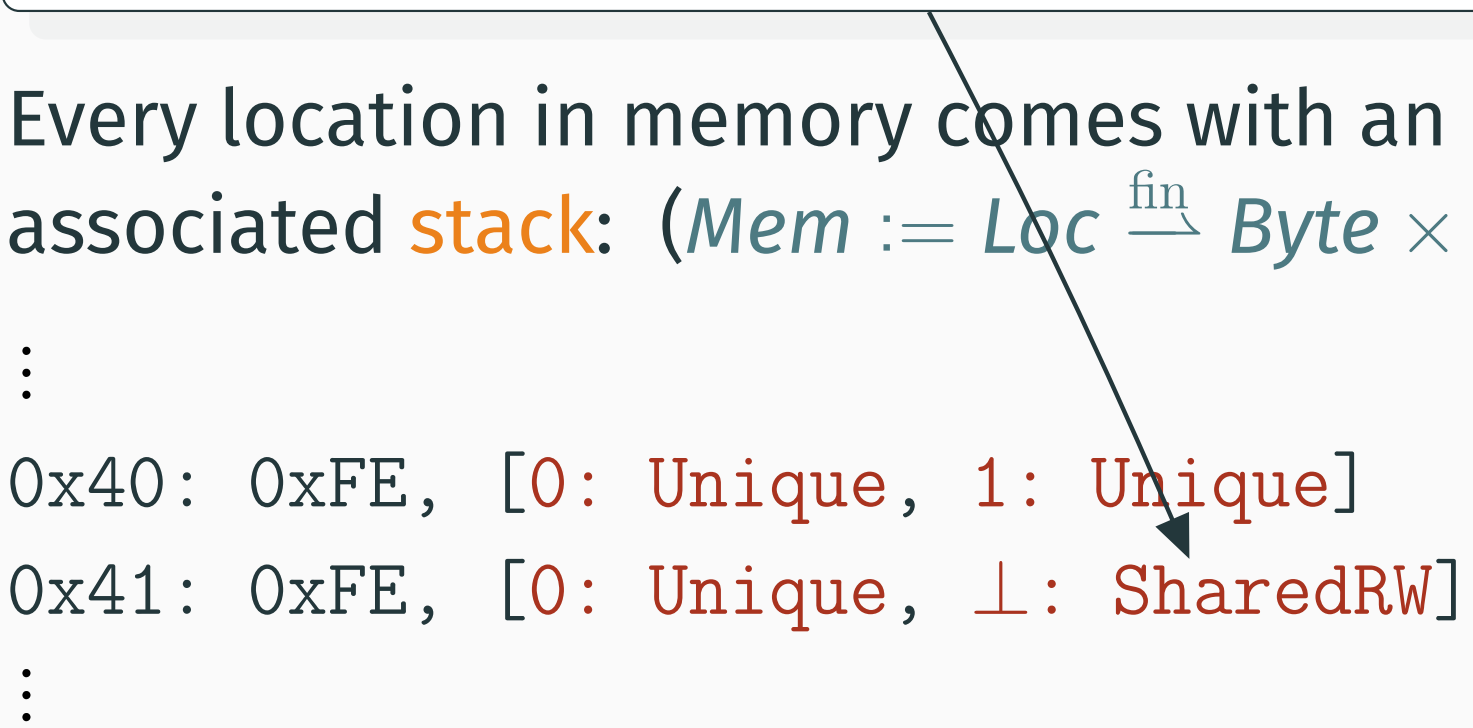
`0x41: 0xFE, [0: Unique, ⊥: SharedRW]`

⋮

Stacked Borrows ingredients

Untagged pointer(s) borrow(s) from reference tagged 0

Every location in memory comes with an associated stack: $(Mem := Loc \xrightarrow{\text{fin}} Byte \times Stack)$



```
 1: static mut ALIAS: *mut i32 = std::ptr::null_mut();
 2: fn main() {
 3:   let l = &mut 0;
 4:   unsafe { ALIAS = l as *mut i32; }
 5:   println!("The answer is {}", test_unique(&mut *l));
 6:   // prints: The answer is 7
 7: }
 8: fn unknown_function_1() {
 9:   unsafe { *ALIAS = 7; }
10: }
11: fn test_unique(x: &mut i32) -> i32 {
12:   *x = 42;
13:   unknown_function_1();
14:   return *x;
15: }
```

```
1: let l = &mut 0;
2: let ALIAS = l as *mut i32;
3: let x = &mut *l;
4: *x = 42;
5: unsafe { *ALIAS = 7; }
6: println!("The answer is {}", *x);
```

```
1: let l = &mut 0; // Tag: 0
```

```
1: let l = &mut 0; // Tag: 0
```

Stack:

[0: Unique]

```
1: let l = &mut 0; // Tag: 0
2: let ALIAS = l as *mut i32; // Tag: ⊥
```

Stack:

`[0: Unique, ⊥: SharedRW]`

Find permission for old tag `0` on stack;
add new permission `⊥: SharedRW` above it

```
1: let l = &mut 0; // Tag: 0
2: let ALIAS = l as *mut i32; // Tag: ⊥
3: let x = &mut *l; // Tag: 1
```

Stack:

`[0: Unique, 1: Unique]`

Find permission for old tag `0` on stack;
remove incompatible `⊥: SharedRW` above;
push new permission `1: Unique`

```
1: let l = &mut 0; // Tag: 0
2: let ALIAS = l as *mut i32; // Tag: ⊥
3: let x = &mut *l; // Tag: 1
4: *x = 42;
```

Stack:

`[0: Unique, 1: Unique]`

Find permission for tag `1` on stack;
remove incompatible items above (none)

```
1: let l = &mut 0; // Tag: 0
2: let ALIAS = l as *mut i32; // Tag: ⊥
3: let x = &mut *l; // Tag: 1
4: *x = 42;
5: unsafe { *ALIAS = 7; }
```

Stack:

```
[0: Unique, 1: Unique]
```

Find permission for tag ⊥ on stack – there is no such item!

```
1: let l = &mut 0; // Tag: 0
2: let ALIAS = l as *mut i32; // Tag: ⊥
3: let x = &mut *l; // Tag: 1
4: *x = 42;
5: unsafe { *ALIAS = 7; }
```

It is undefined behavior to use a pointer whose tag is not on the stack.

Find permission for tag ⊥ on stack – there is no such item!

Stacked Borrows rules (so far)

- Memory access: find permission for our tag, remove incompatible items above
- Assigning fresh tags:
  - Taking a reference (`&mut term`): fresh tag `n`.
    find old tag; remove incompatible above; push new
  - Reference to raw pointer (`term as *mut T`): tag ⊥.
    find old tag; add new just above

Correctness of optimization (sketch)

```
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  // unknown_function_1 cannot have an alias to x
  unknown_function_1();
  return *x; // must return 42
}
```

Correctness of optimization (sketch)

x's tag with `Unique` permission is at the top of the stack

```
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  // unknown_function_1 cannot have an alias to x
  unknown_function_1();
  return *x; // must return 42
}
```

Correctness of optimization (sketch)

x's tag with `Unique` permission is at the top of the stack

```
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  // unknown_function_1 cannot have an alias to x
  unknown_function_1();
  return *x; // must return 42
}
```

If `unknown_function_1` accesses this memory, it will pop x's permission off the stack

Correctness of optimization (sketch)

x's tag with `Unique` permission is at the top of the stack

```
fn test_unique(x: &mut i32) -> i32 {
  *x = 42;
  // unknown_function_1 cannot have an alias to x
  unknown_function_1();
  return *x; // must return 42
}
```

UB unless x's permission is still in the stack

If `unknown_function_1` accesses this memory, it will pop x's permission off the stack

Correctness of optimization (sketch)

`x`'s tag with `Unique` permission is at the top of the stack

```
fn test_unique(x: &mut i32) -> i32 {
}
```

We assumed that `unknown_function_1` cannot have a reference with `x`'s tag!

UB unless `x`'s permission is still in the stack

If `unknown_function_1` accesses this memory, it will pop `x`'s permission off the stack

Retagging for unique tags

```
fn test_unique(x: &mut i32) -> i32 {
  retag(x); // think: x = &mut *x;
  *x = 42;
  // unknown_function_1 cannot have an alias to x
  unknown_function_1();
  return *x; // must return 42
}
```

Retagging for unique tags

x gets a fresh tag with `Unique` permission pushed to top of the stack

```
fn test_unique(x: &mut i32) -> i32 {
  retag(x); // think: x = &mut *x;
  *x = 42;
  // unknown_function_1 cannot have an alias to x
  unknown_function_1();
  return *x; // must return 42
}
```

Retagging for unique tags

x gets a fresh tag with `Unique` permission pushed to top of the stack

```
fn test_unique(x: &mut i32) -> i32 {
  retag(x); // think: x = &mut *x;
  *x = 42;
  // unknown_function_1 cannot have an alias to x
  unknown_function_1();
  return *x; // must return 42
}
```

`unknown_function_1` cannot guess or forge our tag: if it accesses this memory, it will pop x's tag off the stack

Retagging for unique tags

x gets a fresh tag with `Unique` permission pushed to top of the stack

```
fn test_unique(x: &mut i32) -> i32 {
  retag(x); // think: x = &mut *x;
  *x = 42;
  // unknown_function_1 cannot have an alias to x
  unknown_function_1();
  return *x; // must return 42
}
```

UB unless x's permission is still in the stack

`unknown_function_1` cannot guess or forge our tag: if it accesses this memory, it will pop x's tag off the stack

Stacked Borrows rules (so far)

- Memory access: find permission for our tag, remove incompatible items above
- Assigning fresh tags:
  - Taking a reference (`&mut term`): fresh tag `n`.
    find old tag; remove incompatible above; push new
  - Reference to raw pointer (`term as *mut T`): tag ⊥.
    find old tag; add new just above

Stacked Borrows rules (so far)

- Memory access: find permission for our tag, remove incompatible items above
- Assigning fresh tags:
  - Taking a reference (`&mut term`): fresh tag `n`.
    find old tag; remove incompatible above; push new
  - Reference to raw pointer (`term as *mut T`): tag ⊥.
    find old tag; add new just above

- Retag when reference “enters” function (argument, load, call returns)

Stacked Borrows

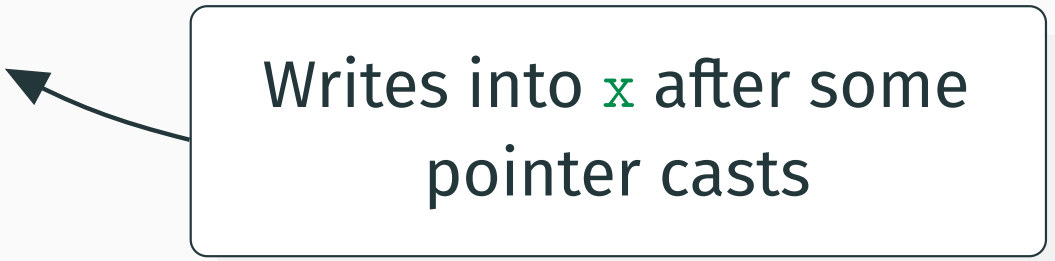
Work-in-progress aliasing model defining which pointers may be used to access memory, ensuring:

- Uniqueness of mutable references
- Immutability of shared references

```
10:
11: fn test_shared(x: &i32) -> bool {
12:   let val = *x;
13:   unknown_function_2(x);
14:   return *x == val;
15: }
```

```
 1: fn main() {
 2:   let l = &mut 0;
 3:   println!("Test result: {}", test_shared(&*l));
 4:   // prints: Test result: false
 5: }
 6: fn unknown_function_2(x: &i32) {
 7:   let ptr = x as *const i32 as *mut i32;
 8:   unsafe { *ptr = 7; }
 9: }
10:
11: fn test_shared(x: &i32) -> bool {
12:   let val = *x;
13:   unknown_function_2(x);
14:   return *x == val;
15: }
```

```
 1: fn main() {
 2:   let l = &mut 0;
 3:   println!("Test result: {}", test_shared(&*l));
 4:   // prints: Test result: false
 5: }
 6: fn unknown_function_2(x: &i32) {
 7:   let ptr = x as *const i32 as *mut i32;
 8:   unsafe { *ptr = 7; }
 9: }
10:
11: fn test_shared(x: &i32) -> bool {
12:   let val = *x;
13:   unknown_function_2(x);
14:   return *x == val;
15: }
```



```
 1: fn main() {
 2:   let l = &mut 0;
 3:   println!("Test result: {}", test_shared(&*l));
 4:   // prints: Test result: false
 5: }
 6: fn unknown_function_2(x: &i32) {
 7:   let ptr = x as *const i32 as *mut i32;
 8:   unsafe { *ptr = 7; }
 9: }
10:
11: fn test_shared(x: &i32) -> bool {
12:   let val = *x;
13:   unknown_function_2(x);
14:   return *x == val;
15: }
```

Writes into `x` after some pointer casts

Overwrites `*x` with 7

```
1: fn main() {
2:   let l = &mut 0;
3:   println!("Test result: {}", test_shared(&*l));
4:   // prints: Test result: false
5: }
6: fn
7:   l
8:   u
9: }
10:
11: fn test_shared(x: &i32) -> bool {
12:   let val = *x;
13:   unknown_function_2(x);
14:   return *x == val;
15: }
```

Mutating `*x` inside `unknown_function_2` must be UB to justify the optimization.

```
1: let mut l = 0;
2: let a = &mut l;
```

```
1: let mut l = 0;
2: let a = &mut l;
3: let b = &*a;
```

```
1: let mut l = 0;
2: let a = &mut l;
3: let b = &*a;
4: let _val = *b;
```

```
1: let mut l = 0;
2: let a = &mut l;
3: let b = &*a;
4: let _val = *b;
5: let _val = *a;
6: let _val = *b;
```

```
1: let mut l = 0;
2: let a = &mut l;
3: let b = &*a;
4: let _val = *b;
5: let _val = *a;
6: let _val = *b;
7: *a = 1;
8: let _val = *b; // ERROR: lifetime of ‘b‘ has ended
```

```
1: let mut l = 0;
2: let a = &mut l;
3: let b = &*a;
4: let _val = *b;
5: let _val = *a;
6: let _val = *b;
7: *a = 1;
8: let _val = *b; // ERROR: lifetime of ‘b‘ has ended
```

```
1: let mut l = 0;
2: let a = &mut l;
3: let b = &*a;
4: let _val = *b;
5: let _val = *a;
6: let _val = *b;
7: *a = 1;
8: let _val = *b; // ERROR: lifetime of ‘b‘ has ended
```

- Reads allowed through `a` and `b`...
- ...until first write through `a`
- No mutation between creation and use of a shared reference

```
1: let mut l = 0;
2: let a = &mut l;
3: let b = &*a;
4: let _val = *b;
5: let _val = *a;
6:
7:
8:                                                    ended
```

Shared references allow reads but no writes with other pointers.

- Reads allowed through `a` and `b`...
- ...until first write through `a`
- No mutation between creation and use of a shared reference

Stacked Borrows ingredients

Pointers carry a tag: $(\mathit{PtrVal} := \mathit{Loc} \times \mathbb{N}_{\perp})$

`0x40[1], 0x41[⊥]`

references (`&mut T`/`&T`) are identified by a tag

raw pointers (`*mut T`/`*const T`) fall back to untagged

Stacked Borrows ingredients

Pointers carry a tag: ($\mathit{PtrVal} := \mathit{Loc} \times \mathbb{N}_{\perp}$)

`0x40[1], 0x41[⊥]`

Every location in memory comes with an associated stack: ($\mathit{Mem} := \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Byte} \times \mathit{Stack}$)

⋮

```
0x40: 0xFE, [0: Unique, 1: Unique]
0x41: 0xFE, [0: Unique, ⊥: SharedRW]
0x42: 0x00, [0: Unique, 1: SharedRO]
```

⋮

Stacked Borrows ingredients

Pointers carry a tag: ($PtrVal := Loc \times \mathbb{N}_{\perp}$)

`0x40[1], 0x41[⊥]`

Every location in memory comes with an associated stack: ($Mem := Loc \xrightarrow{\text{fin}} Byte \times Stack$)

⋮

```
0x40: 0xFE, [0: Unique, 1: Unique]
0x41: 0xFE, [0: Unique, ⊥: SharedRW]
0x42: 0x00, [0: Unique, 1: SharedRO]
```

Can be read but not written by `1`.

```
 1: fn main() {
 2:   let l = &mut 0;
 3:   println!("Test result: {}", test_shared(&*l));
 4:   // prints: Test result: false
 5: }
 6: fn unknown_function_2(x: &i32) {
 7:   let ptr = x as *const i32 as *mut i32;
 8:   unsafe { *ptr = 7; }
 9: }
10:
11: fn test_shared(x: &i32) -> bool {
12:   let val = *x;
13:   unknown_function_2(x);
14:   return *x == val;
15: }
```

```
1: let l = &mut 0;
2: let x = &*l;
3: let val = *x;
4: let ptr = x as *const i32 as *mut i32;
5: unsafe { *ptr = 7; }
6: let test = *x;
7: println!("Test result: {}", test == val);
```

```
1: let l = &mut 0; // Tag: 0
```

Stack:

[0: Unique]

```
1: let l = &mut 0; // Tag: 0
2: let x = &*l; // Tag: 1
```

Stack:

[0: Unique; 1: SharedRO]

Find read permission for 0 on stack;
remove read-incompatible items above (none);
push new permission 1: SharedRO

```
1: let l = &mut 0; // Tag: 0
2: let x = &*l; // Tag: 1
3: let val = *x;
```

Stack:

[0: Unique; 1: SharedRO]

Find read permission for 1 on stack;
remove read-incompatible items above (none)

```
1: let l = &mut 0; // Tag: 0
2: let x = &*l; // Tag: 1
3: let val = *x;
4: let ptr = x as *const i32 as *mut i32; // Tag: ⊥
```

Stack:

[0: Unique; 1: SharedRO; ⊥: SharedRO]

Find read permission for 1 on stack;
remove read-incompatible items above (none);
push new permission ⊥: SharedRO

```
1: let l = &mut 0; // Tag: 0
2: let x = &*l; // Tag: 1
3: let val = *x;
4: let ptr = x as *const i32 as *mut i32; // Tag: ⊥
5: unsafe { *ptr = 7; }
```

Stack:

[0: Unique; 1: SharedRO; ⊥: SharedRO]

ptr cannot be used for writing:

⊥only has read-only permission!

Stacked Borrows rules

- Memory access: find permission for our tag, remove incompatible items above
- Assigning fresh tags:
  - Taking a reference (`&mut term`): fresh tag `n`.
    find old tag; remove incompatible above; push new
  - Reference to raw pointer (`term as *mut T`): tag $\perp$.
    find old tag; add new just above
- Compatibility:
  - Reads are compatible with `SharedRW`, `SharedRO`
  - Writes to `SharedRW` are compatible with `SharedRW`
- Retag when reference "enters" function (argument, load, call returns)

Correctness of optimization (sketch)

```
fn test_shared(x: &i32) -> bool {
  retag(x); // think: x = &*x;
  let val = *x;
  // unknown_function_2 cannot mutate x
  unknown_function_2(x);
  return *x == val; // must return true
}
```

Correctness of optimization (sketch)

x has fresh tag with permission SharedRO at the top of the stack

```
fn test_shared(x: &i32) -> bool {
  retag(x); // think: x = &*x;
  let val = *x;
  // unknown_function_2 cannot mutate x
  unknown_function_2(x);
  return *x == val; // must return true
}
```

Correctness of optimization (sketch)

x has fresh tag with permission SharedRO at the top of the stack

```
fn test_shared(x: &i32) -> bool {
  retag(x); // think: x = &*x;
  let val = *x;
  // unknown_function_2 cannot mutate x
  unknown_function_2(x);
  return *x == val; // must return true
}
```

If unknown_function_2 writes to this memory, x's tag will be removed from the stack

Correctness of optimization (sketch)

x has fresh tag with permission SharedRO at the top of the stack

```
fn test_shared(x: &i32) -> bool {
  retag(x); // think: x = &*x;
  let val = *x;
  // unknown_function_2 cannot mutate x
  unknown_function_2(x);
  return *x == val; // must return true
}
```

If unknown_function_2 writes to this memory, x's tag will be removed from the stack

UB unless x's permission is still in the stack

What else?

What I didn't talk about:

- Interior mutability (shared references through which mutation is allowed)
- Barriers, two-phase borrows
- Formal model in Coq (proofs of optimizations in progress)

Future work:

- Integrating stacked borrows into RustBelt
- Handling integer-pointer casts
- Proving correctness of compilation to LLVM

For more details,
check out Ralf's blog at:

https://www.ralfj.de/blog/