COMP 737011 - Memory Safety and Programming Language Design

Lecture 10: Rust Compiler

徐 辉

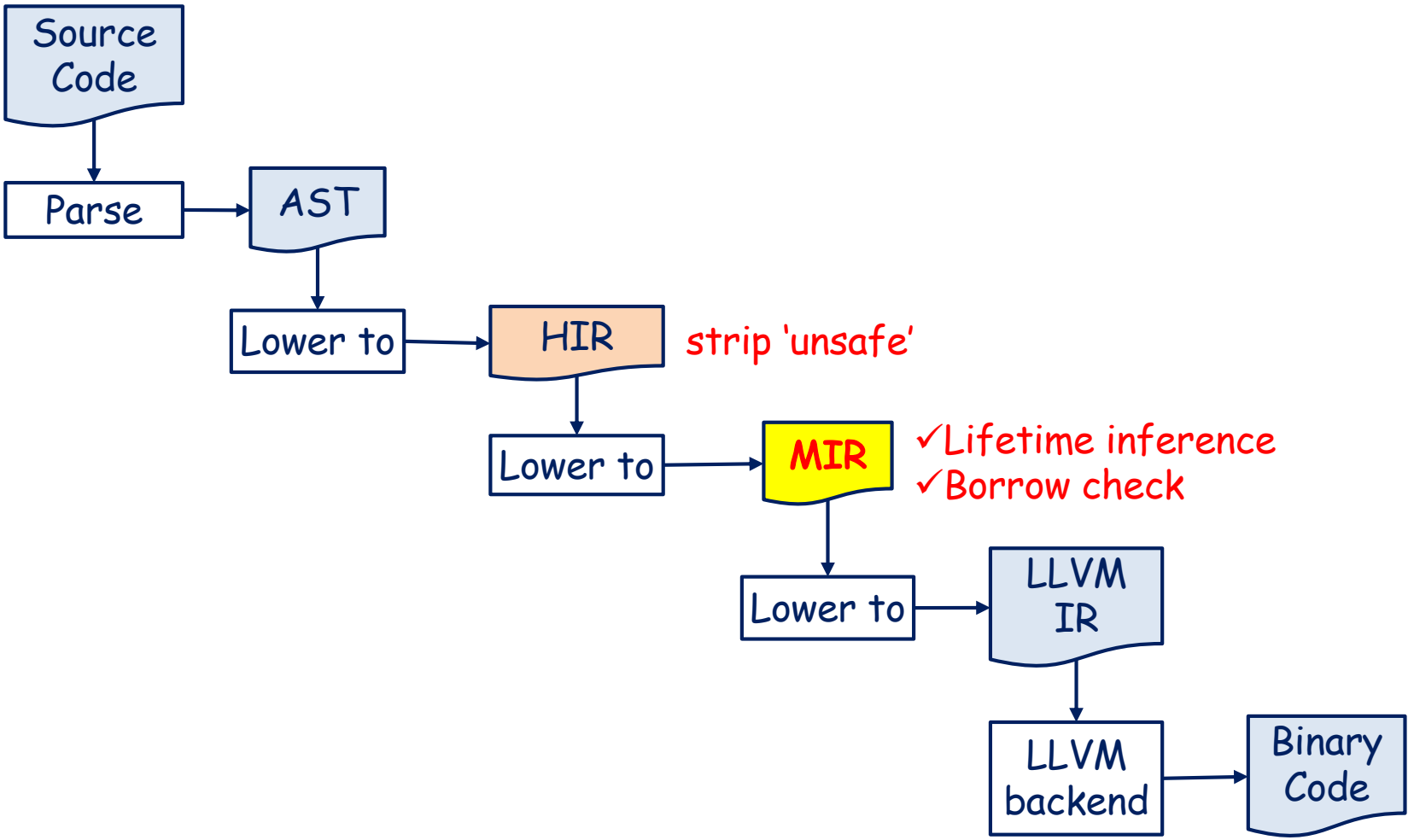
xuh@fudan.edu.cn

Questions

- 1. Overview the compiler
  - HIR
  - MIR
- 2. How's the following mechanisms implemented?
  - 2.1 Lifetime inference
  - 2.2 Borrow check
  - 2.3 Unsafe
- 3. Review the Effectiveness Rust

1. Overview The Compiler

Compilation Stages



https://rustc-dev-guide.rust-lang.org/compiler-src.html

HIR

- HIR is similar to AST (tree-based IR) but more succinct, e.g.,
  - Remove parenthesis
  - Convert "if let" to "match"
- Command to output HIR

```
#: rustc -Z help
...
#: rustc -Z unpretty=hir-tree toy.rs
Crate {
    item: CrateItem {
        module: Mod {
            inner: toy.rs:2:1: 5:2 (#0),
            item_ids: [
                ItemId {
                    id: HirId {
                        owner: DefId(0:1 ~ toy[317d]::{{misc}}[0]),
                        local_id: 0,
                    },
                },
...
```

MIR

- MIR is linear IR

```
fn main() {
    let alice = Box::new(1);
    let bob = &alice;
}
```

```
#: rustc -Z dump-mir=all toy.rs
```

```
fn main() -> () {
    let mut _0: ();
    let _1: std::boxed::Box<i32>;
    scope 1 {
        debug alice => _1;
        let _2: &std::boxed::Box<i32>;
        scope 2 {
            debug bob => _2;
        }
    }
    bb0: {
        StorageLive(_1);
        _1 = const std::boxed::Box::<i32>
             ::new(const 1_i32)
             -> [return: bb2, unwind: bb1];
    }
    bb1 (cleanup): {
        resume;
    }
    bb2: {
        FakeRead(ForLet, _1);
        StorageLive(_2);
        _2 = &_1;
        FakeRead(ForLet, _2);
        _0 = const ();
        StorageDead(_2);
        drop(_1) -> [return: bb3, unwind: bb1];
    }
    bb3: {
        StorageDead(_1);
        return;
    }
}
```

return value

assignment

borrow

Why is Rust Binary Large?

- Rust prefers static linkage for "unknown" targets;
- Specify "prefer-dynamic" for dynamic linkage.

```
#: rustc toy.rs
#: ll
-rwxrwxr-x  1 aisr aisr 307832 4月  18 18:55 toy*
-rw-rw-r--  1 aisr aisr     66 4月  18 17:52 toy.rs
#: rustc -C prefer-dynamic toy.rs
#: ll
-rwxrwxr-x  1 aisr aisr  21040 4月  18 18:56 toy*
#: rustc -C prefer-dynamic -C opt-level=3 toy.rs
#: ll
-rwxrwxr-x  1 aisr aisr  16904 4月  18 18:57 toy*
```

2.1 Lifetime Inference

https://rust-lang.github.io/rfcs/2094-nll.html

Lifetime Inference

- Problem: infer the minimum lifetime of each reference
- Requirement: The lifetime of each reference should not exceeds its referent value.
- Lifetimes are not based on expression rather than lexical scopes or blocks.

```
fn main() { //scope start
    let mut alice = Box::new(1);
    let bob = &alice;
    *alice = 2;
} //scope ends
```

bob lives only in this statement

Constraint-based Lifetime Inference

```
fn main() {
    let mut a: i32 = 1;
    let mut b: i32 = 2;
    let mut p: & T = &a;
    // program point 1
    if condition {
        // program point 2
        print(*p);
        // program point 3
        p = &b;
        // program point 4
    }
    // program point 5
    print(*p);
    // program point 6
}
```

p is alive.

p is alive.

p is dead

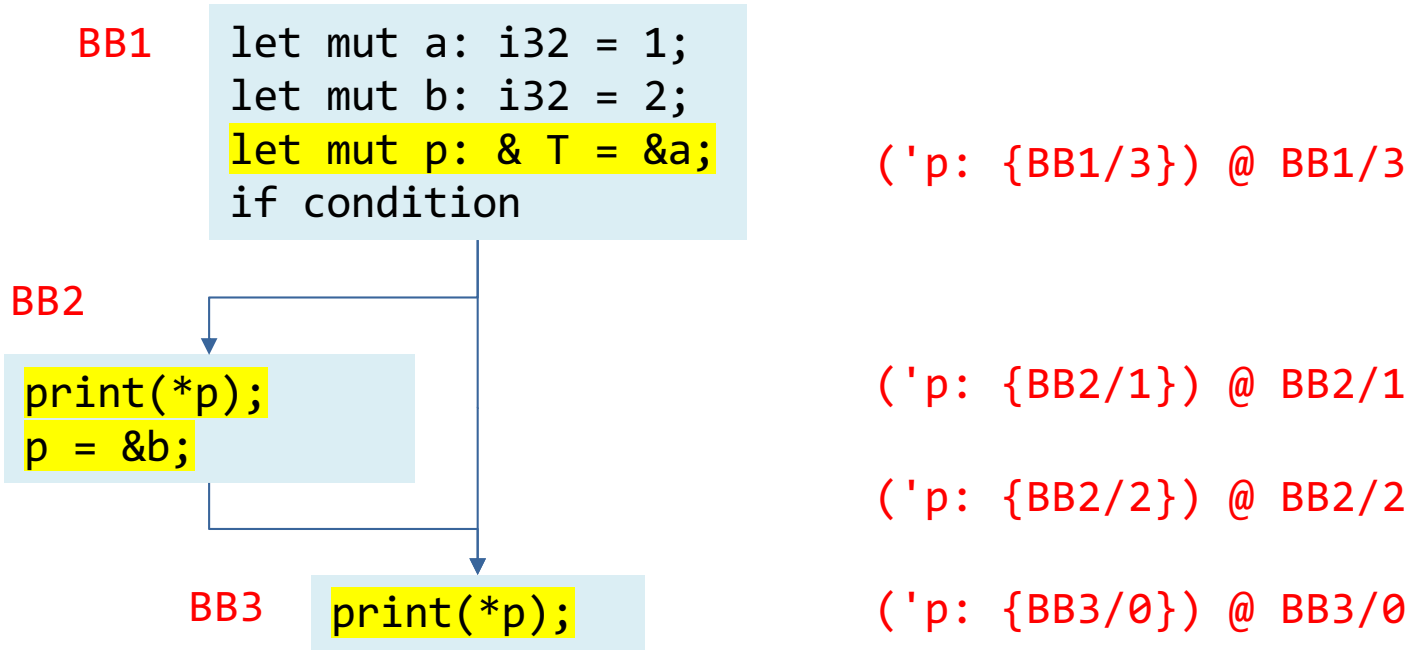
p is alive

p is alive

p is dead

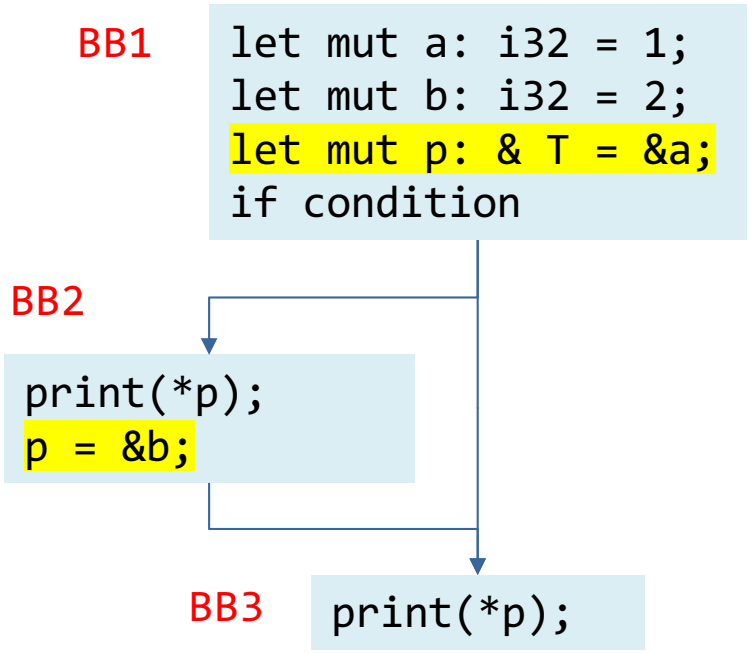
Constraint Representation: Liveness

- (L: {P}) @ P: lifetime L is alive at the point P



Constraint Representation: Subtyping

- (L1: L2) @ P: lifetime L1 outlives lifetime L2 at point P



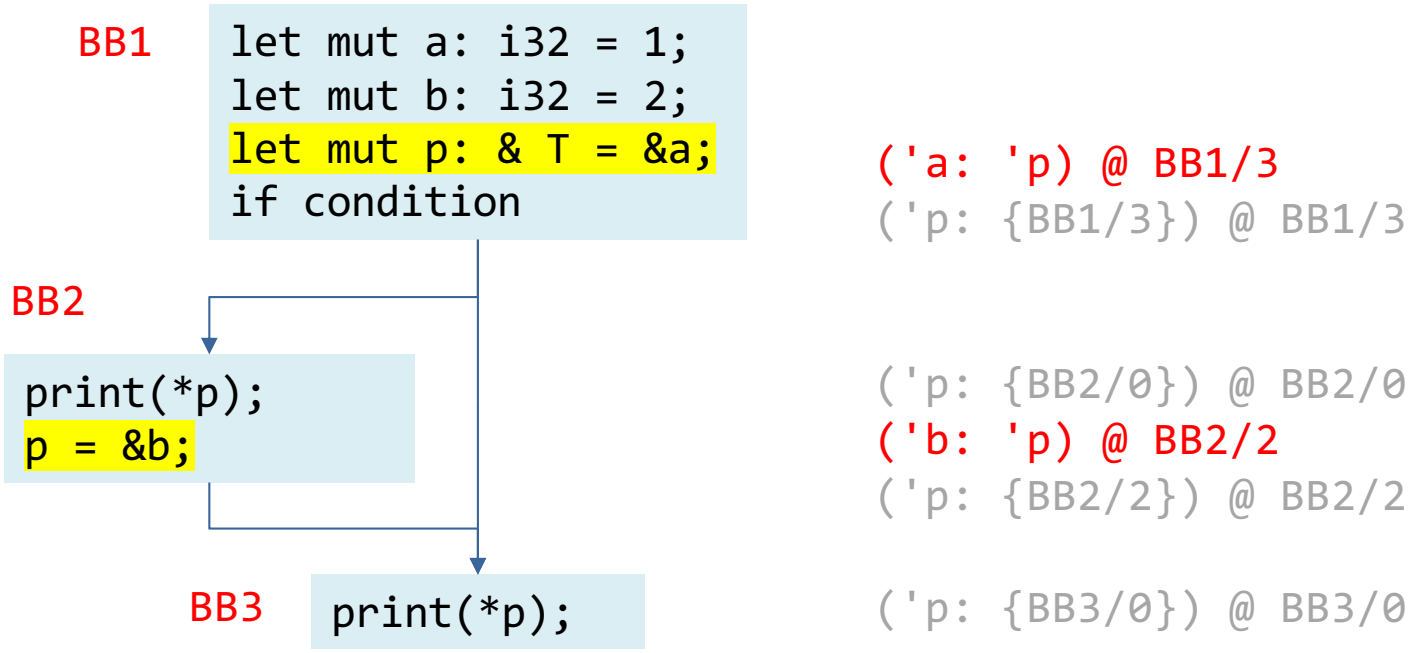
('a: 'p) @ BB1/3
('p: {BB1/3}) @ BB1/3

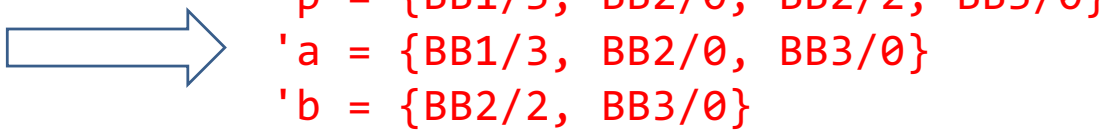
('p: {BB2/1}) @ BB2/1
('b: 'p) @ BB2/2
('p: {BB2/2}) @ BB2/2

('p: {BB3/0}) @ BB3/0

Solving Constraints via Fixed-Point Iteration

- Init each lifetime variable with an empty set
- Iterate over the constraints via depth-first search
- Stop until all constraints are satisfied





More Rules

- We should define the constraint extraction rule for each particular type of statement.
- Reborrow constraint is complicated…

```
let mut a: i32 = 1;
let mut b = & a;
let c = &*b;
```

```
('a: BB0/1) @ BB0/1
('b: BB0/2) @ BB0/2 ('a: 'b) @ BB0/2
('c: BB0/3) @ BB0/3 ('b: 'c) @ BB0/3
=> also implies ('a: 'c)
```

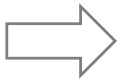
```
let mut a: i32 = 22;
let mut b = & a;
let c = & b;
let d: = & **c;
```

```
('a: BB0/1) @ BB0/1
('b: BB0/2) @ BB0/1 ('a: 'b) @ BB0/2
('c: BB0/3) @ BB0/3 ('b: 'c) @ BB0/3
(‘d: BB0/3) @ BB0/4 ('c: 'd) @ BB0/4
=> also implies ('b: 'd), ('a: 'd)
=> The code would be falsely rejected
```

Case Study

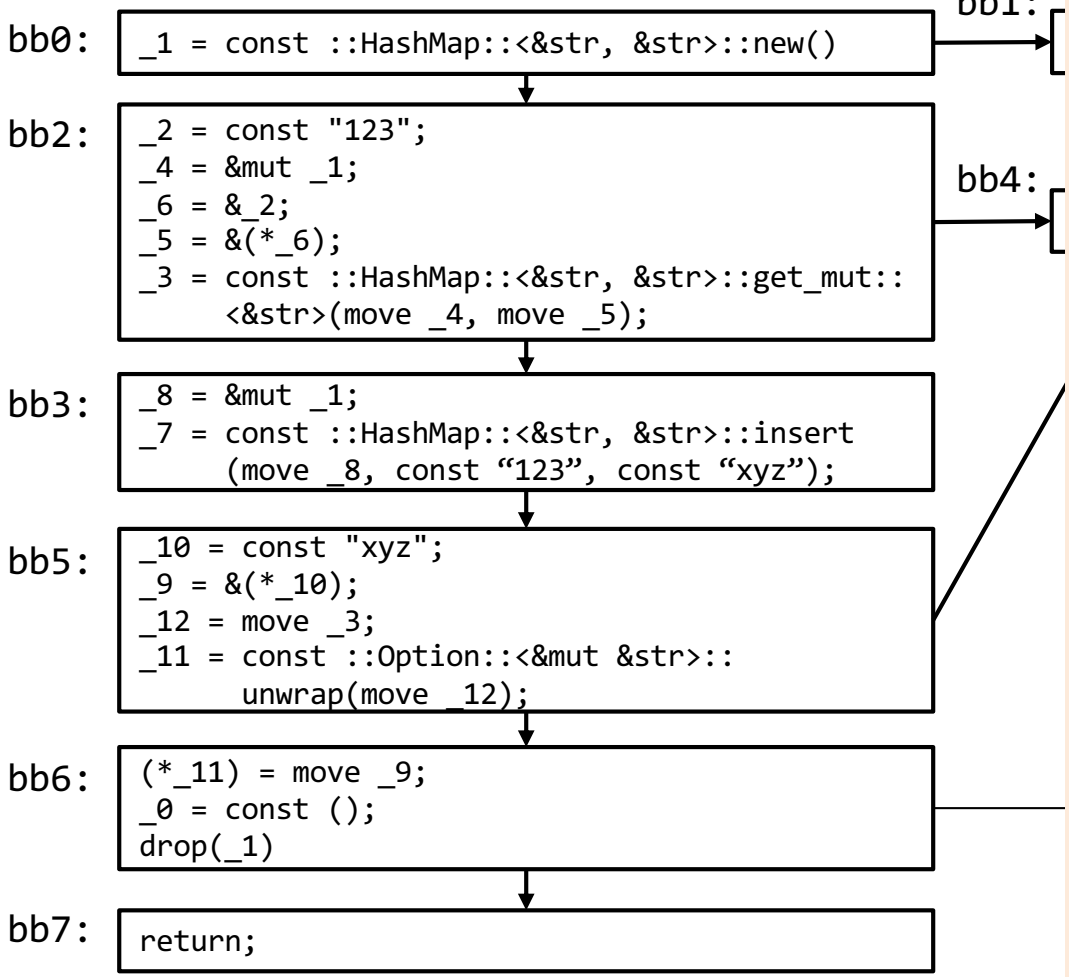
- Do NLL analysis manually

```
use std::collections::HashMap;

fn test1() {
    let mut map = HashMap::new();
    let key = "123";
    let v =  map.get_mut(&key);
    map.insert("123","xyz");
    *(v.unwrap()) = "xyz";
}
```



```
#: rustc -Z dump-mir=all test1.rs
#: cat mir_dump/rustc.test1.-------.nll.0.mir
```

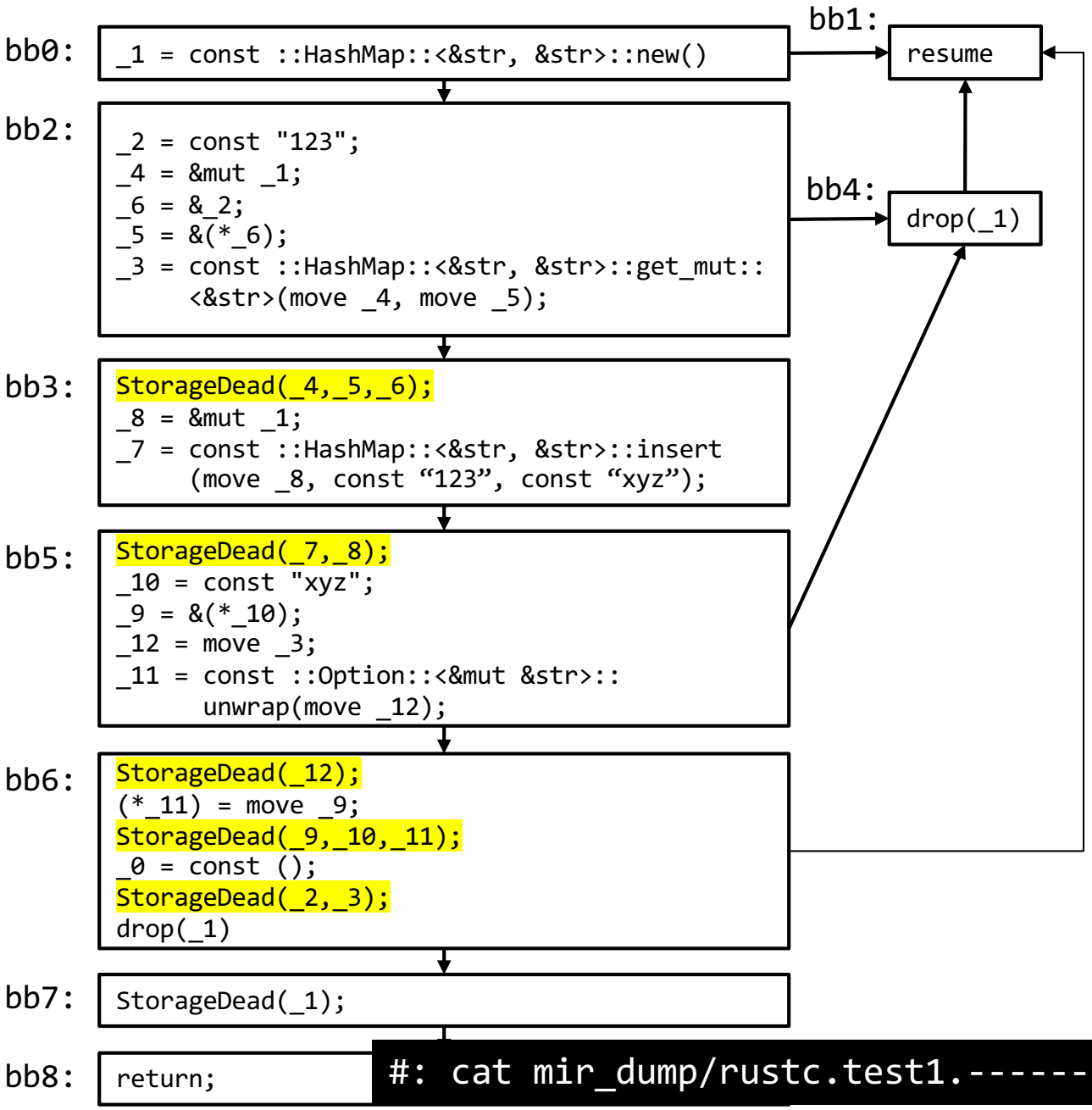
MIR of test1()




```
('1: {BB0/1}) @ BB0/1
('1: {BB2/2}) @ BB2/2
('1: {BB3/1}) @ BB3/1
('1: {BB4/1}) @ BB4/1
('1: {BB6/1}) @ BB6/1

('2: {BB2/1}) @ BB2/1
('2: {BB2/3}) @ BB2/3

('3: {BB2/5}) @ BB2/5
('3: {BB5/3}) @ BB5/3

('4: {BB2/2}) @ BB2/2
('4: {BB2/5}) @ BB2/5

('5: {BB2/4}) @ BB2/4
('4: {BB2/5}) @ BB2/5

('6: {BB2/3}) @ BB2/3
('6: {BB2/4}) @ BB2/4

('7: {BB3/2}) @ BB3/2

('8: {BB3/1}) @ BB3/1
('8: {BB3/2}) @ BB3/2

('9: {BB5/2}) @ BB5/2
('9: {BB6/1}) @ BB6/1

('10: {BB5/1}) @ BB5/1
('10: {BB5/2}) @ BB5/2

('11: {BB5/3}) @ BB5/4
('11: {BB6/1}) @ BB6/1

('12: {BB5/3}) @ BB5/3
('12: {BB5/3}) @ BB5/4
```

```
('1: '4) @ BB2/2
('1: '8) @ BB3/1

('6: '2) @ BB2/3

('6: '5) @ BB2/4
('10: '9) @ BB2/4
```

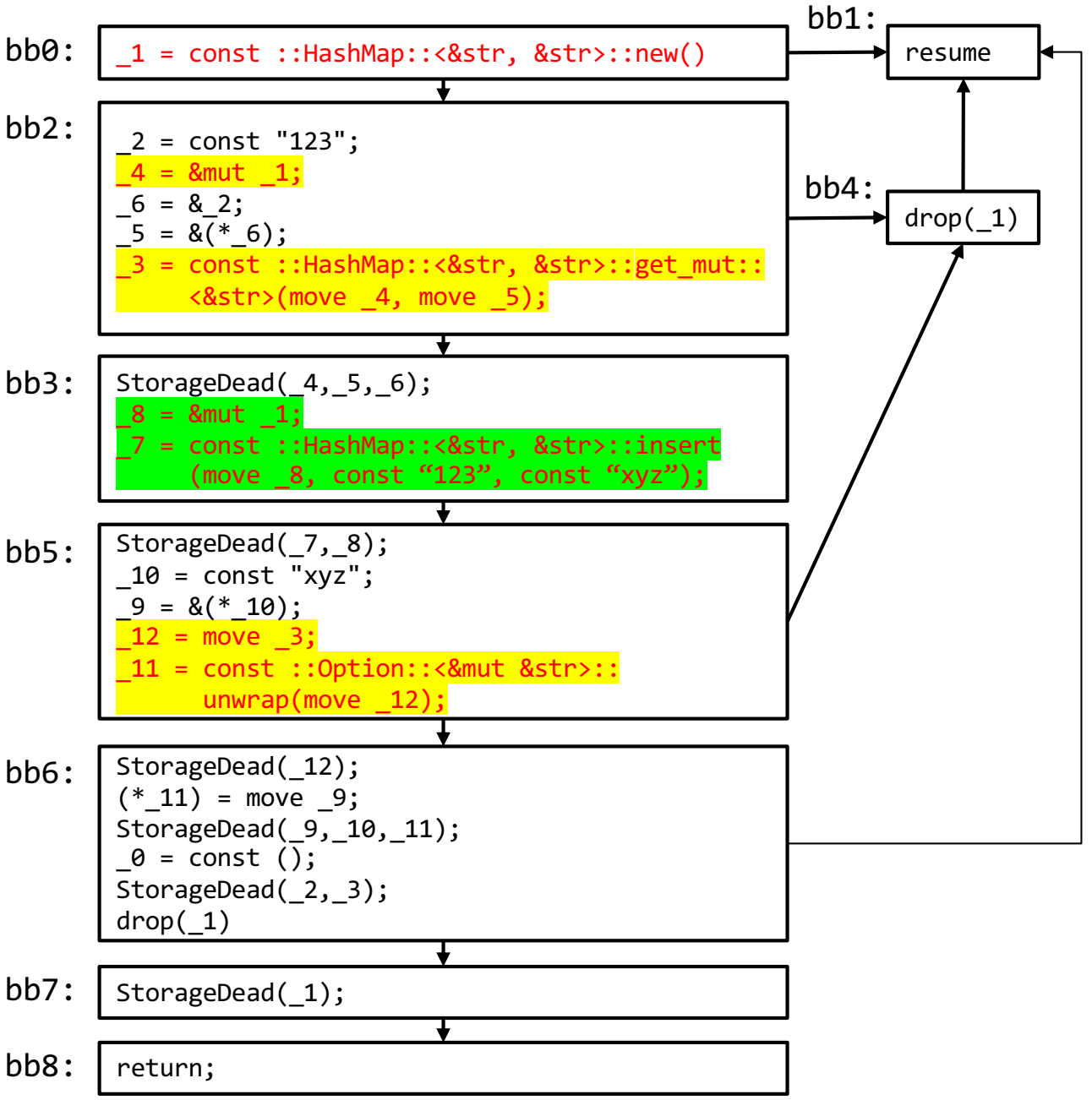
Result



- Use StorageDead() to reclaim variables on the stack;
- Use drop() to run the destructor of a Drop trait

```
#: cat mir_dump/rustc.test1.-------.mir_map.0.mir
```

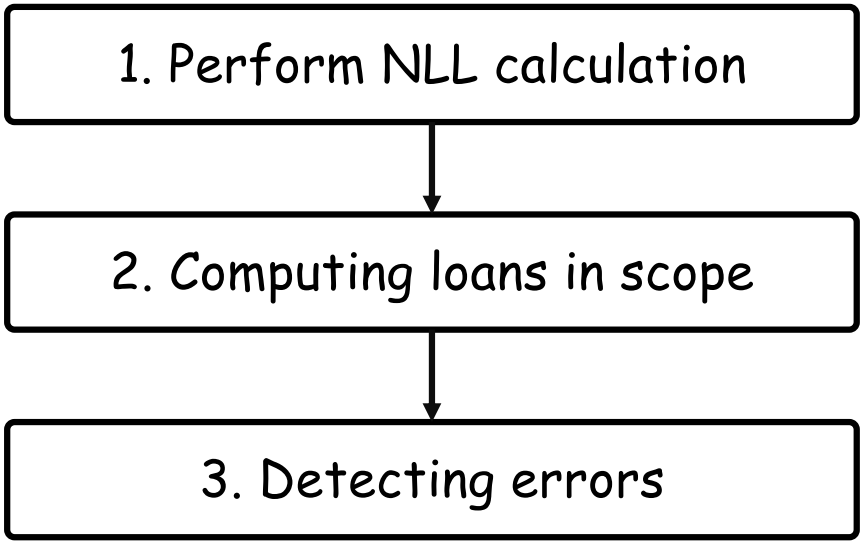
But The Code Should Not Compile…



2.2 Borrow check

Basic Steps

- Operates on the MIR
- Older implementation operated on the HIR



Computing Loans in scope

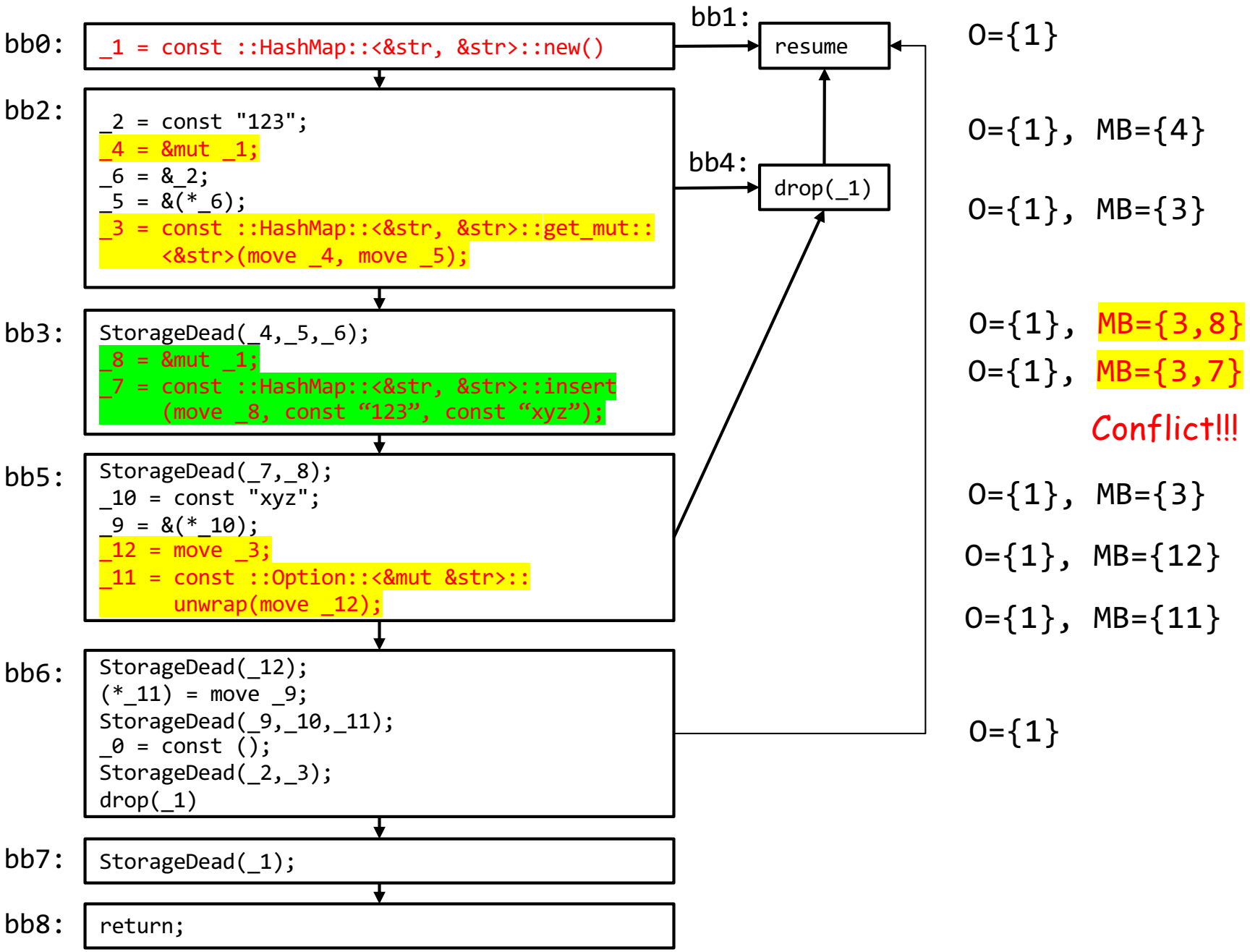
- For a statement at point P in the graph, we define the "transfer function":
  - Any loans whose region does not include P are killed;
  - If this is a borrow statement, the corresponding loan is generated;
  - If this is an assignment lv = <rvalue>, then any loan for some path P of which lv is a prefix is killed.

Detecting Errors

- All variables are initialized before they are used.
- You can't move the same value twice.
- You can't move a value while it is borrowed.
- You can't access a place while it is mutably borrowed
- You can't mutate a place while it is immutably borrowed.

https://rustc-dev-guide.rust-lang.org/borrow_check.html

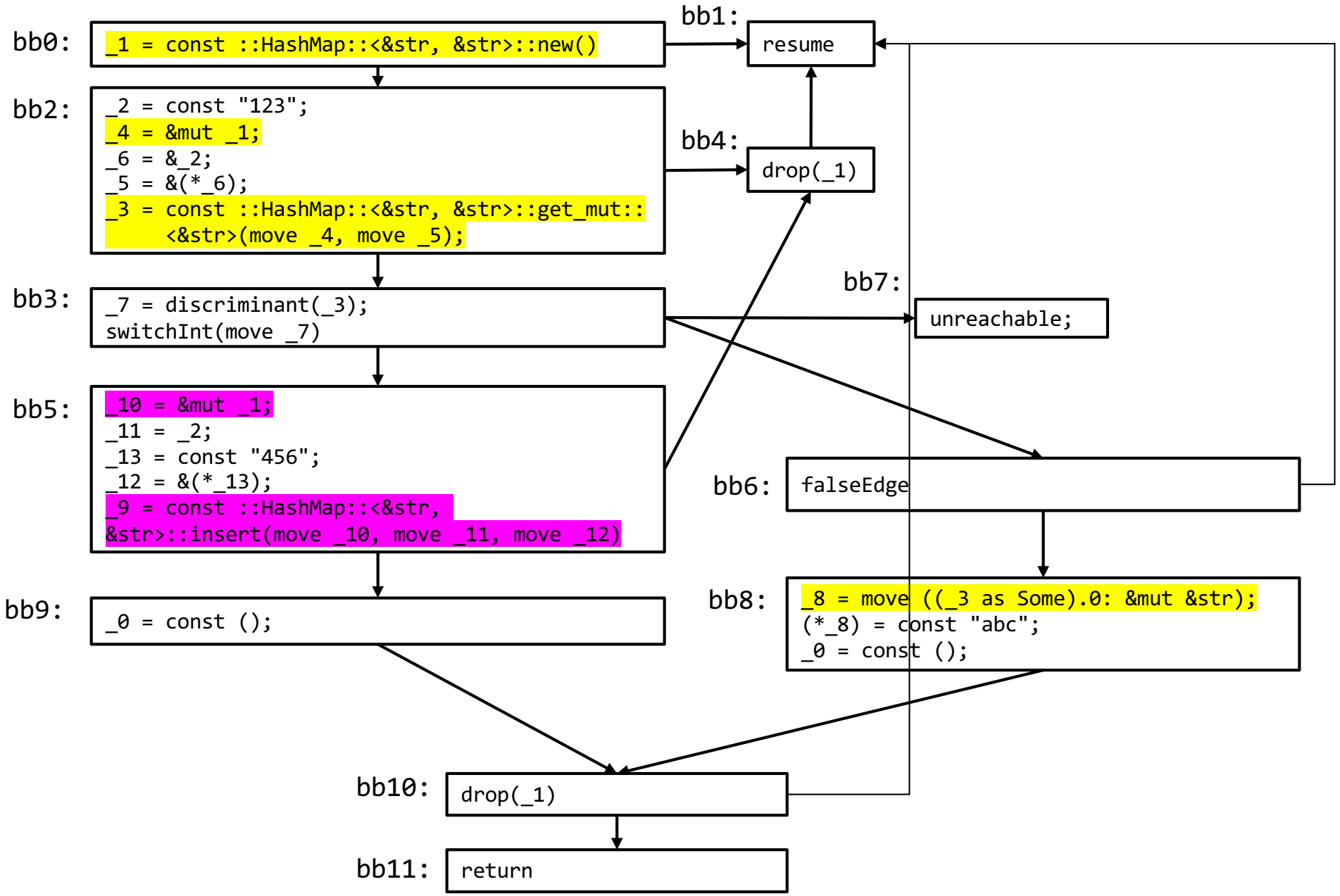
Sample Analysis Approach



Does the following code compile?

```
fn main() {
    let mut map = HashMap::new();
    let key = "123";
    match map.get_mut(&key) {
        Some(value) => *value = "abc",
        None => {
            map.insert(key, "456");
        }
    }
}
```

- Not in old Rust compiler
- Now compiles


bb0:
_1 = const ::HashMap::<&str, &str>::new()
bb1:
resume
bb2:
_2 = const "123";
_4 = &mut _1;
_6 = &_2;
_5 = &(*_6);
_3 = const ::HashMap::<&str, &str>::get_mut::
<&str>(move _4, move _5);
bb4:
drop(_1)
bb3:
_7 = discriminant(_3);
switchInt(move _7)
bb7:
unreachable;
bb5:
_10 = &mut _1;
_11 = _2;
_13 = const "456";
_12 = &(*_13);
_9 = const ::HashMap::<&str,
&str>::insert(move _10, move _11, move _12)
bb6:
falseEdge
bb9:
_0 = const ();
bb8:
_8 = move ((_3 as Some).0: &mut &str);
(*_8) = const "abc";
_0 = const ();
bb10:
drop(_1)
bb11:
return

2.3 Unsafe

https://github.com/rust-lang/compiler-team/issues/402

Unsafe Code

- Unsafe marker is stripped away in MIR
- Raw pointers may introduce shared mutable aliases

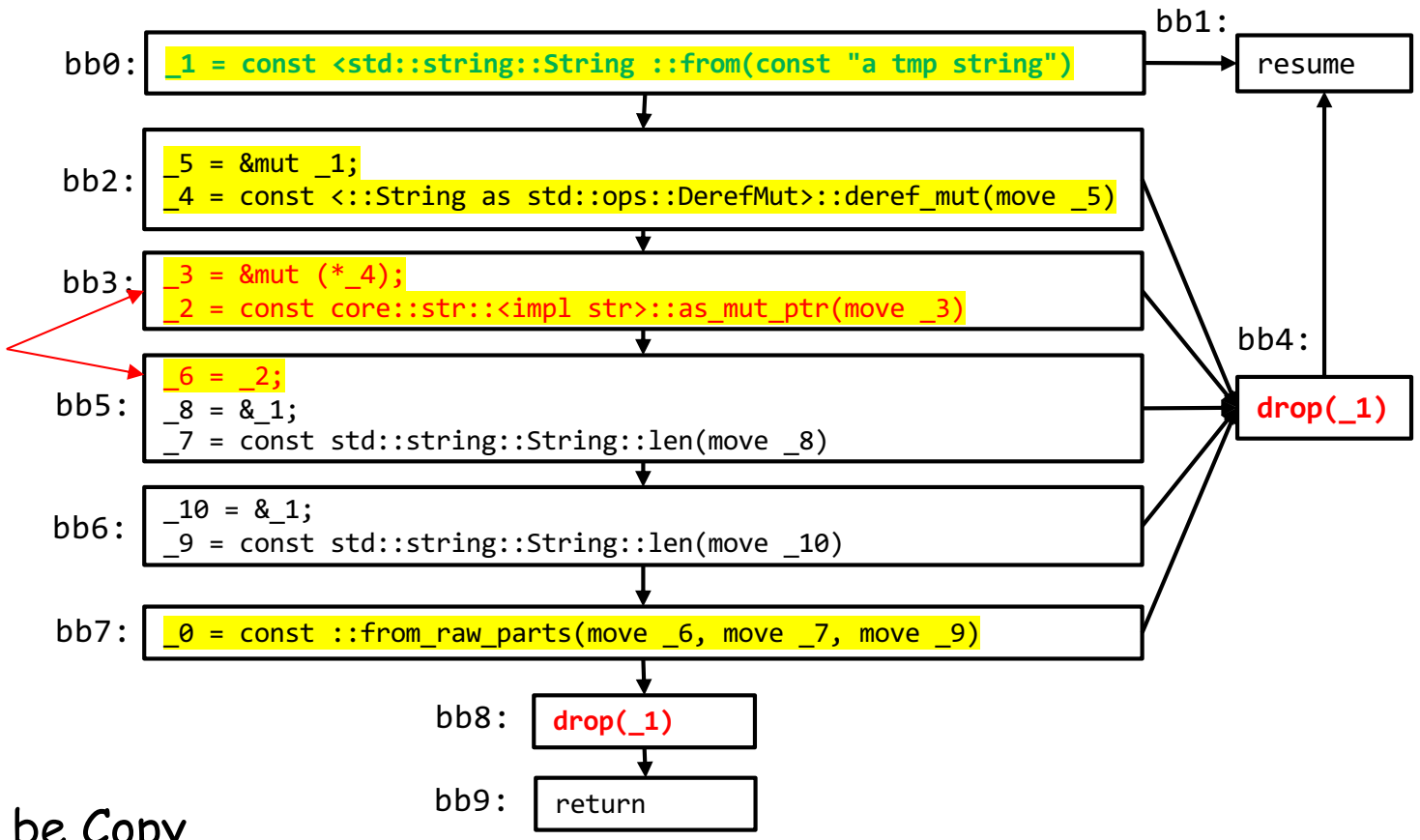
PoC of CVE-2019-16140

```
fn genvec()->Vec<u8>{
    let mut s = String::from("a tmp string");
    let ptr = s.as_mut_ptr();
    unsafe{
        let v = Vec::from_raw_parts(ptr,s.len(),s.len());
        v
    }
}

fn main(){
    let v = genvec(); //v is dangling
    println!("{:?}",v); //illegal memory access
}
```

```
#:./uaf_from_raw_parts
[104, 16, 195, 158, 247, 85, 0, 0, 0, 0, 0, 0]
Segmentation fault (core dumped)
#: rustc -V
rustc 1.44.1 (c7087fe00 2020-06-17)
```

Problem Analysis

```
fn genvec()->Vec<u8>{
    let mut s = String::from("a tmp string");
    let ptr = s.as_mut_ptr();
    unsafe{
        let v = Vec::from_raw_parts(ptr,s.len(),s.len());
        v
    }
}
```

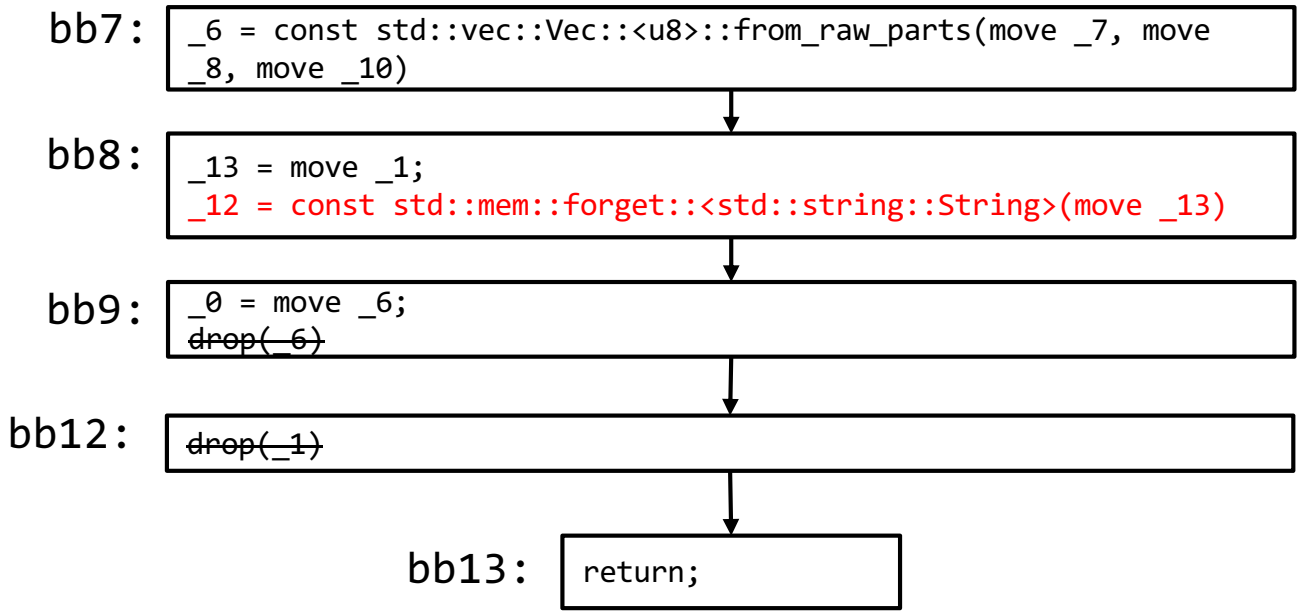


Raw pointer is Copy
Simple alias
Not borrow

Mut Ref should not be Copy

Bug Fix

```
fn genvec2()->Vec<u8>{
    let mut s = String::from("a tmp string");
    let ptr = s.as_mut_ptr();
    unsafe{
        let v = Vec::from_raw_parts(ptr,s.len(),s.len());
+:      std::mem::forget(s);
        v
    }
}
```

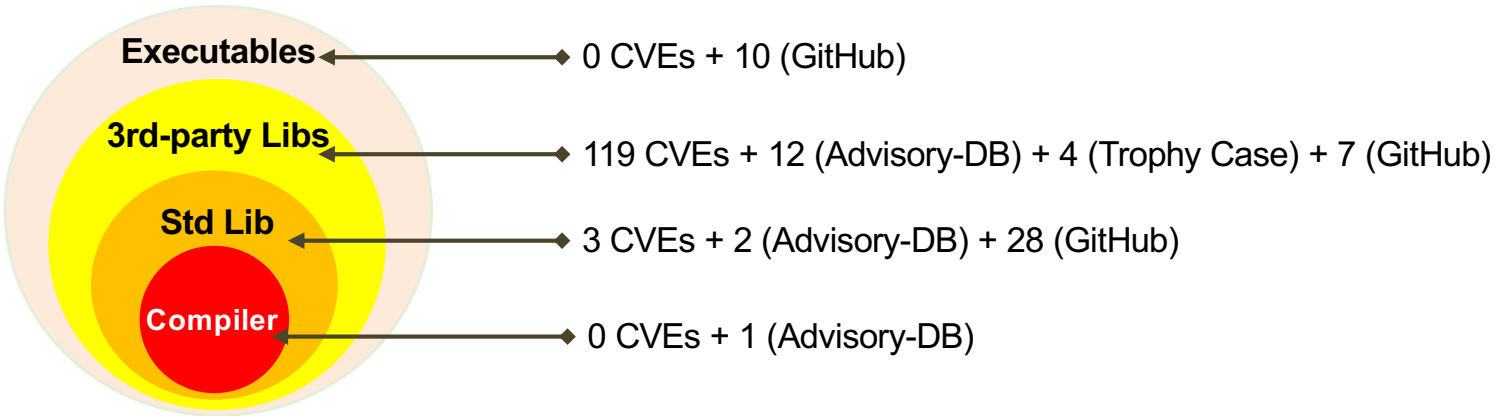


3. Review the Effectiveness Rust

"Memory-safety challenge considered solved? An in-depth study with all Rust CVEs", TOSEM, 2021

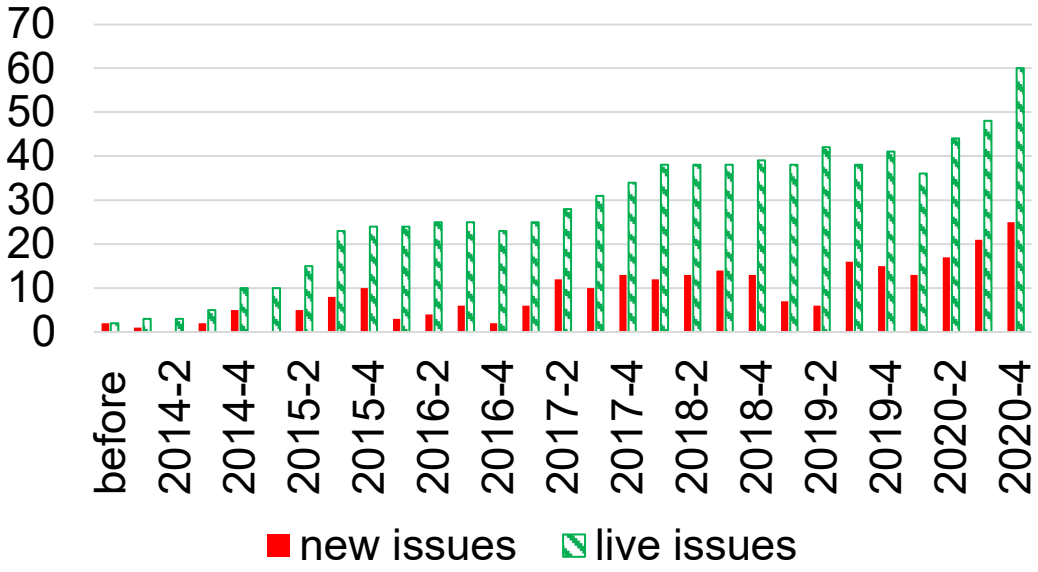
Case Study

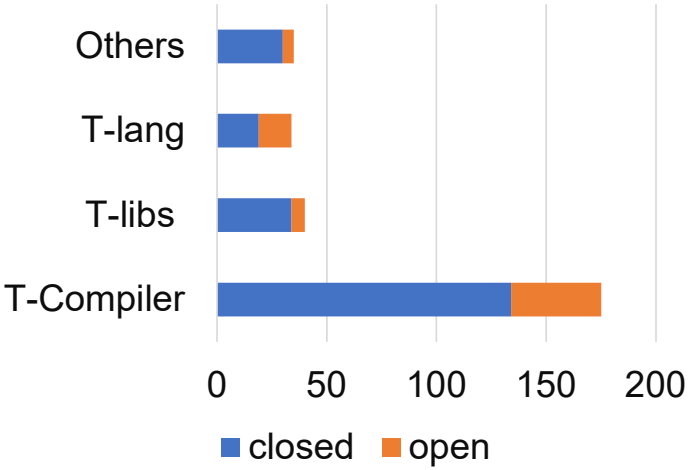
- Research Questions
  - How effective is Rust in preventing memory-safety bugs?
  - What are the characteristics of memory-safety bugs?
  - What lessons can we learn to make Rust more secure?
- Dataset of memory-safety bugs (186 in total)
  - Rust Advisory-DB(CVEs) till 2020-12-31
  - Trophy Case
  - Rust compiler
  - Other GitHub projects



How effective is Rust?

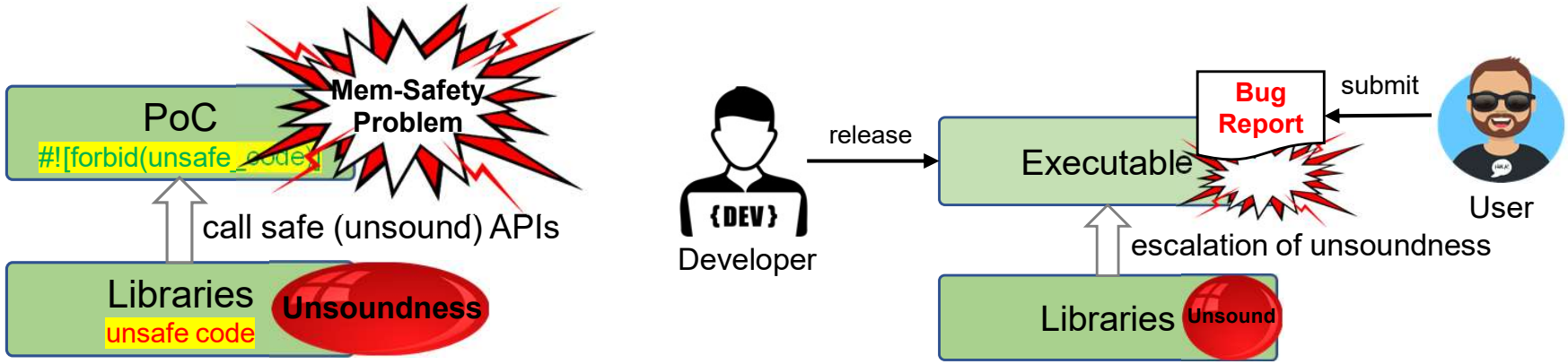
- Do all memory-safety bugs require unsafe code?
  - Yes, except the compiler bug
- How robust is Rust compiler?
  - Trend of unsoundness issues of the Rust compiler





How Severe are These Bugs?

- All CVEs are library bugs
  - Not involve security issues directly
  - Depend on how developers use the API



Characteristics of Bugs

- Automatic Memory Reclaim
- Unsound Function
- Unsound Generic or Trait
- Other Errors

Table 1. Distribution of memory-safety bugs in Rust std-lib + CVEs + others. For simplicity, we count the CVEs of Rust std-lib into Rust std-lib.

		Consequence					Total
Culprit		Buf. Over-R/W	Use-After-Free	Double Free	Uninit Mem	Other UB	
Auto Memory Reclaim	Bad Drop at Normal Block	0 + 0 + 0	1 + 9 + 6	0 + 2 + 1	0 + 2 + 0	0 + 1 + 0	22
	Bad Drop at Cleanup Block	0 + 0 + 0	0 + 0 + 0	1 + 7 + 0	0 + 5 + 0	0 + 0 + 0	13
Unsound Function	Bad Func. Signature	0 + 2 + 0	1 + 5 + 2	0 + 0 + 0	0 + 0 + 0	1 + 2 + 4	17
	Unsoundness by FFI	0 + 2 + 0	5 + 1 + 0	0 + 0 + 0	0 + 0 + 0	1 + 2 + 1	12
Unsound Generic or Trait	Insuff. Bound of Generic	0 + 0 + 1	0 + 33 + 2	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	36
	Generic Vul. to Spec. Type	3 + 0 + 1	1 + 0 + 0	0 + 0 + 0	1 + 0 + 1	1 + 2 + 0	10
	Unsound Trait	1 + 2 + 1	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	0 + 2 + 0	6
Other Errors	Arithmetic Overflow	3 + 1 + 0	1 + 0 + 0	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	5
	Boundary Check	1 + 9 + 0	1 + 0 + 0	0 + 0 + 0	0 + 0 + 0	1 + 0 + 0	12
	No Spec. Case Handling	2 + 2 + 1	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	2 + 1 + 1	9
	Exception Handling Issue	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	1 + 2 + 1	4
	Wrong API/Args Usage	0 + 3 + 0	1 + 4 + 0	0 + 0 + 0	0 + 1 + 1	0 + 5 + 2	17
	Other Logical Errors	0 + 4 + 1	2 + 3 + 4	0 + 0 + 1	0 + 1 + 0	1 + 4 + 1	22
Total		40	82	12	12	39	185

Case1: Auto Memory Reclaim

Code 1. PoC of use-after-free and double free bugs due to automatic memory reclaim.

```
1  fn genvec() -> Vec<u8> {
2      let mut s = String::from("a tmp string");
3      /*fix2: let mut s = ManuallyDrop::new(String::from("a tmp string"));*/
4      let ptr = s.as_mut_ptr();
5      unsafe {
6          let v = Vec::from_raw_parts(ptr, s.len(), s.len());
7          /*fix1: mem::forget(s);*/
8          return v;
9          /*s is freed when the function returns*/
10     }
11 }
12 fn main() {
13     let v = genvec();
14     assert_eq!('a' as u8,v[0]); /*use-after-free*/
15     /*double free: v is released when the function returns*/
16 }
```

Case2: Drop Uninitialized Memory

Code 4. PoC of dropping uninitialized memory during stack unwinding.

```
1  struct Foo { vec : Vec<i32>, }
2  impl Foo {
3     pub unsafe fn read_from(src: &mut Read) -> Foo {
4        let mut foo = mem::uninitialized::<Foo>();
5        //panic!(); /*panic here would recalim the uninitialized memory of type <Foo>*/
6        let s = slice::from_raw_parts_mut(&mut foo as *mut _ as *mut u8, mem::size_of::<Foo>());
7        src.read_exact(s);
8        foo
9     }
10 }
11 fn main() {
12    let mut v = vec![0,1,2,3,4,5,6];
13    let (p, len, cap) = v.into_raw_parts();
14    let mut u = [p as u64, len as _, cap as _];
15    let bp:*const u8 = &u[0] as *const u64 as *const _;
16    let mut b:&[u8] = unsafe { slice::from_raw_parts(bp, mem::size_of::<u64>()*3) };
17    let mut foo = unsafe{Foo::read_from(&mut b as _)};
18    println!("foo␣=␣{:?}", foo.vec);
19 }
```

Case 3: Insufficient Trait Bound

Code 5. PoC of lacking Send trait bound to generic.

```
1  struct MyStruct<T> {t:T}
2  unsafe impl<T> Send for MyStruct<T> {}
3  //fix: unsafe impl<T:Send> Send for MyStruct<T> {}
4  fn main() {
5      let mut s = MyStruct { t:Rc::new(String::from("untouched_data")) };
6      for _ in 0..99{
7          let mut c = s.clone();
8          std::thread::spawn(move || {
9              if !Rc::get_mut(&mut c.t).is_none(){
10                 (*Rc::get_mut(&mut c.t).unwrap()).clear();
11             }
12             println!("c.t_=_{:?}", c.t);
13         });
14     }
15 }
```

Case 4: Vulnerable Generic

Code 6. PoC of unsound generic that does not respect the memory alignment.

```
#[repr(align(128))]
struct LargeAlign(u8);
struct MyStruct<T> { v:Vec<u8>,  _marker:PhantomData<*const T>, }
impl<T:Sized> MyStruct<T> {
    fn from(mut value:T) -> MyStruct<T> {
        let size = size_of::<T>();
        let mut v = Vec::with_capacity(size_of::<T>());
        let src:*const T = &value;
        unsafe {
            ptr::copy(src, v.as_mut_ptr() as _, 1);
            v.set_len(size)
        }
        MyStruct { v, _marker:PhantomData }
    }
}
impl<T:Sized> ::std::ops::Deref for MyStruct<T> {
    type Target = T;
    fn deref(&self) -> &T{
        let p = self.v.as_ptr() as *const u8 as *const T;
        unsafe { &*p }
    }
}
fn main() {
    let s = MyStruct::from(LargeAlign(123));
    let v = &*s as *const _ as usize;
    assert!(v % std::mem::align_of::<LargeAlign>() == 0);
}
```

Case 5: Unsound Trait

Code 7. PoC of unsound Trait.

```
trait MyTrait {
    fn type_id(&self) -> TypeId where Self: 'static {
        TypeId::of::<Self>()
    }
}
impl dyn MyTrait {
    pub fn is<T:MyTrait + 'static>(&self) -> bool {
        TypeId::of::<T>() == self.type_id()
    }
    pub fn downcast<T:MyTrait + 'static>(self: Box<Self>) -> Result<Box<T>, Box<dyn MyTrait>> {
        if self.is::<T>(){ unsafe {
           let raw:*mut dyn MyTrait = Box::into_raw(self);
           Ok(Box::from_raw(raw as *mut T))
        }} else { Err(self) }
    }
}
impl<T> MyTrait for Box<T> {}
impl MyTrait for u128 {}
impl MyTrait for u8 {
    fn type_id(&self) -> TypeId where Self: 'static {
        TypeId::of::<u128>()
    }
}
fn main(){
    let s = Box::new(10u8);
    let r = MyTrait::downcast::<u128>(s);
}
```

Lessons Learnt?

- Best practice for code suggestion?
  - Generic Bound Declaration
  - Avoiding Bad Drop at Cleanup Block.
- Static analysis with unsafe code?
  - We will discuss more next week

More Reference

- https://rustc-dev-guide.rust-lang.org
- https://rust-lang.github.io/rfcs/1211-mir.html
- https://rust-lang.github.io/rfcs/2094-nll.html