COMP 737011 - Memory Safety and Programming Language Design

Lecture 12: Rust Compiler and Enhancement

徐 辉

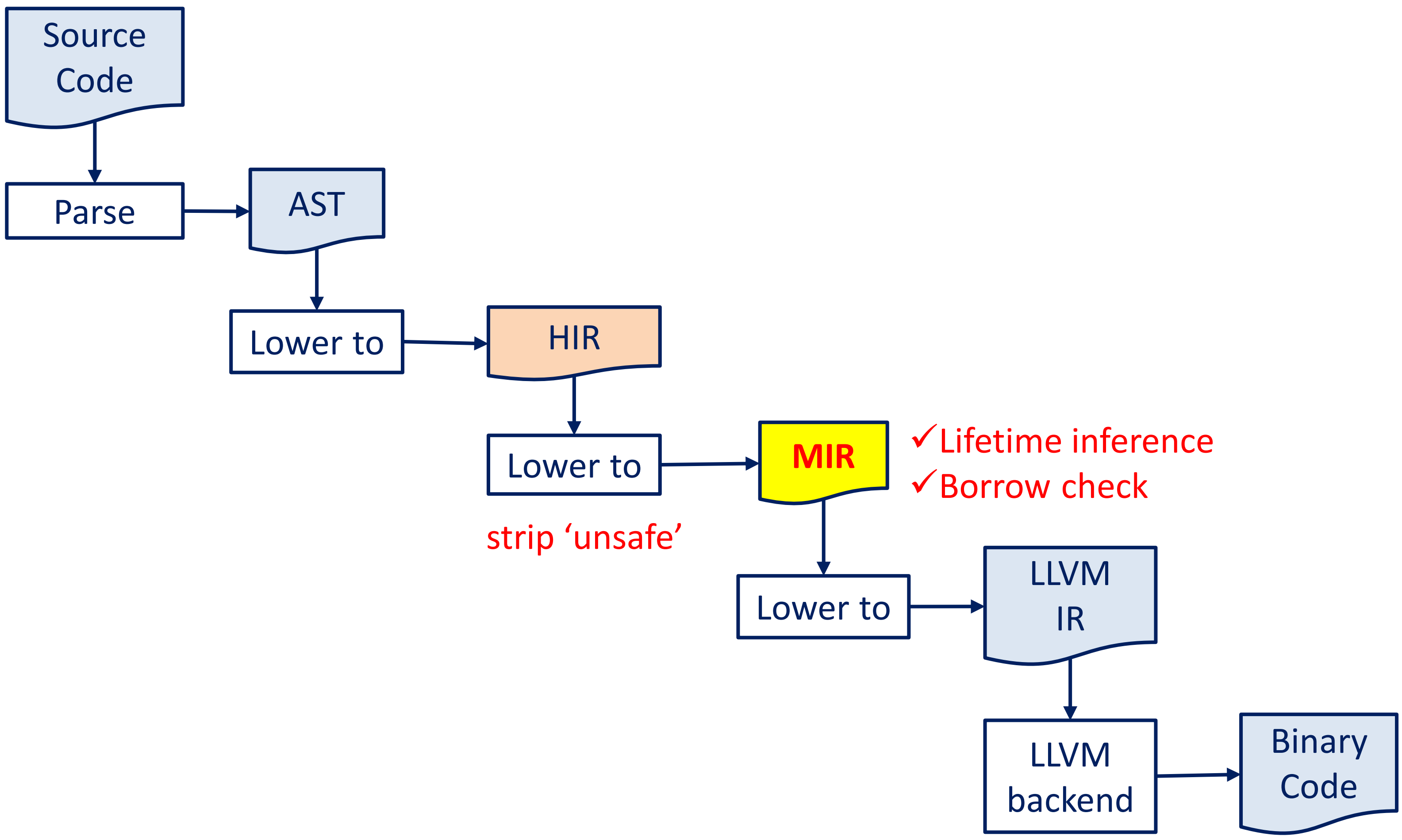
xuh@fudan.edu.cn

Questions

1. Compiler Overview
2. Mechanism Implementation
   - Unsafe
   - Lifetime inference
   - Borrow check
3. Security Enhancement

1. Compiler Overview

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

https://rust-lang.github.io/rfcs/1211-mir.html

2. Mechanism Implementation

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

Unsafe Code

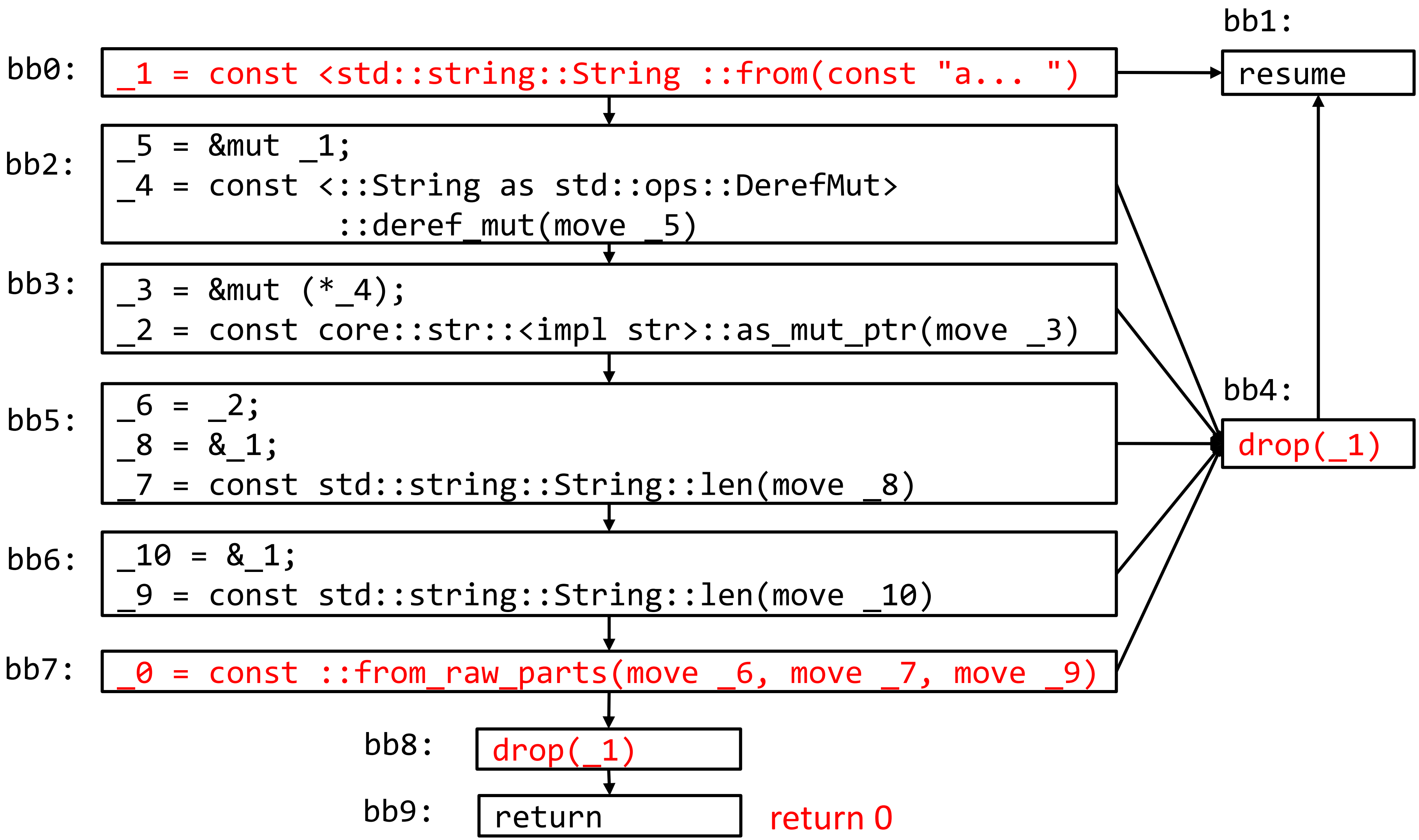
- Unsafe marker is stripped away in MIR
- Raw pointers may introduce shared mutable aliases

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)
```

MIR Analysis



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
    }
}
```

```
_6 = const std::vec::Vec::<u8>::from_raw_parts(move _7, move _8, move _10)
```

```
_13 = move _1;
_12 = const std::mem::forget::<std::string::String>(move _13)
```

```
_0 = move _6;
```

```
return;
```

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 is alive only in this statement

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

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

BB1

```
let mut a: i32 = 1;
let mut b: i32 = 2;
let mut p: & T = &a;
if condition
```

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

BB2

```
print(*p);
p = &b;
```

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

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

BB3

```
print(*p);
```

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

Constraint Representation: Subtyping

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

BB1

```
let mut a: i32 = 1;
let mut b: i32 = 2;
let mut p: & T = &a;
if condition
```

BB2

```
print(*p);
p = &b;
```

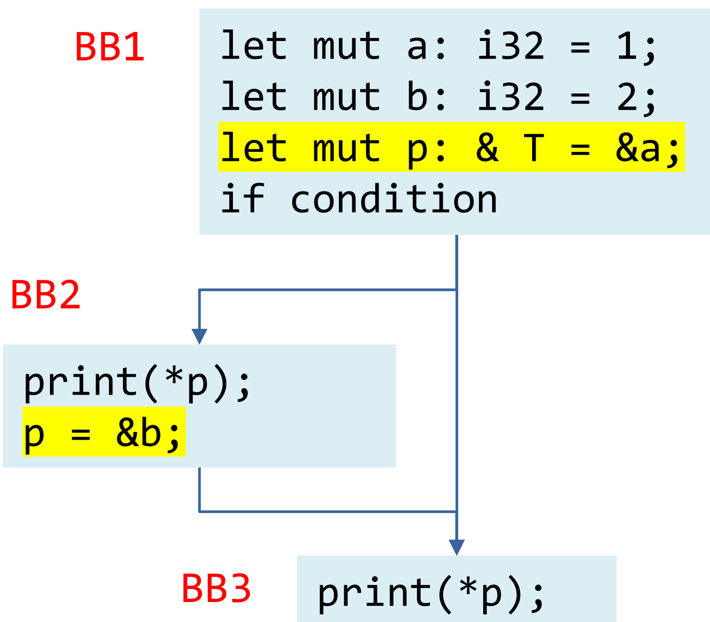
BB3

```
print(*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




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

('p: {BB2/0}) @ BB2/0
('b: 'p) @ BB2/2
('p: {BB2/2}) @ BB2/2

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

⇩

```
'p = {BB1/3, BB2/0, BB2/2, BB3/0}
'a = {BB1/3, BB2/0, BB3/0}
'b = {BB2/2, BB3/0}
```

More Rules

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

```
let mut a: i32 = 1;     ('a: BB0/1) @ BB0/1
let mut b = & a;        ('b: BB0/2) @ BB0/2 ('a: 'b) @ BB0/2
let c = &*b;            ('c: BB0/3) @ BB0/3 ('b: 'c) @ BB0/3
                        => also implies ('a: 'c)
```

```
let mut a: i32 = 22;    ('a: BB0/1) @ BB0/1
let mut b = & a;        ('b: BB0/2) @ BB0/1 ('a: 'b) @ BB0/2
let c = &mut b;         ('c: BB0/3) @ BB0/3 ('b: 'c) @ BB0/3
let d: = &mut **c;      ('d: BB0/4) @ BB0/4 ('c: 'd) @ BB0/4
use(*b);                => also implies ('b: 'd), ('a: 'd)
//use(b); ————————————— => The code would be falsely rejected
use(d);
```

More of False Rejections: Why?

```rust
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";
}
```

pub fn get_mut<Q: ?Sized>
(&mut self, k: &Q)
-> Option<&mut V>

Before lifetime elision:
&'a self, &'b Q, &'a V

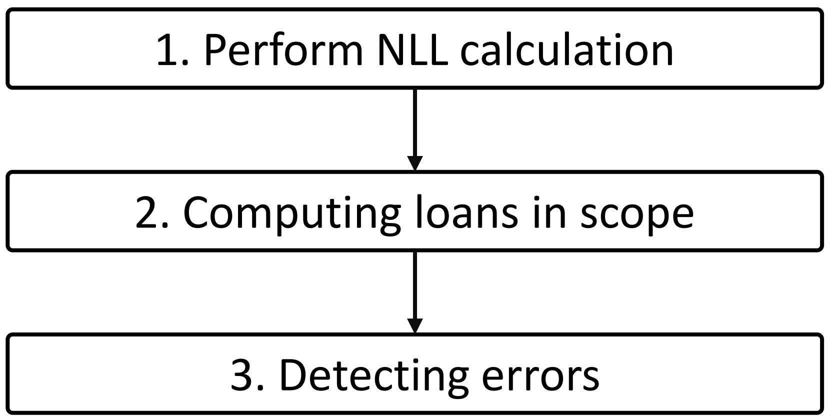
```
error[E0499]: cannot borrow `map` as mutable more than once at a time
 --> nll_case.rs:7:5
6 |     let v =  map.get_mut(&key);
  |              ----------------- first mutable borrow occurs here
7 |     map.insert("123","xyz");
  |     ^^^^^^^^^^^^^^^^^^^^^^^ second mutable borrow occurs here
8 |     *(v.unwrap()) = "xyz";
  |       - first borrow later used here
```

Lifetime Inference Based on MIR

```
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:  _8 = &mut _1;
      _7 = const ::HashMap::<&str, &str>::insert
           (move _8, const “123”, const “xyz”);

bb5:  _10 = const "xyz";
      _9 = &(*_10);
      _12 = move _3;
      _11 = const ::Option::<&mut &str>::unwrap(move _12);

bb6:  (*_11) = move _9;
      _0 = const ();
      drop(_1)

bb7:  return;
```

Borrow Check

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



Computing Loans in Scope: Transfer Function

- Any loans whose region does not include the point are killed
- For a borrow statement, the corresponding loan is generated
- For statement lv = <rvalue>, killed the loan of which lv is a prefix

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

Rules to Detect 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

Sample Analysis Approach

bb0:

```
_1 = const ::HashMap::<&str, &str>::new()
```

O={1}

bb2:

```
_2 = const "123";
_4 = &mut _1;
_6 = &_2;
_5 = &(*_6);
_3 = const ::HashMap::<&str, &str>::get_mut::
     <&str>(move _4, move _5);
```

O={1}, MB={4}

O={1}, MB={3}

bb3:

```
_8 = &mut _1;
_7 = const ::HashMap::<&str, &str>::insert
     (move _8, const “123”, const “xyz”);
```

O={1}, MB={3,8}

O={1}, MB={3,7}

Conflict!!!

bb5:

```
_10 = const "xyz";
_9 = &(*_10);
_12 = move _3;
_11 = const ::Option::<&mut &str>::unwrap(move _12);
```

O={1}, MB={3}

O={1}, MB={12}

O={1}, MB={11}

bb6:

```
(*_11) = move _9;
_0 = const ();
drop(_1)
```

O={1}

bb7:

```
return;
```

Is The Following Code Compilable?

```
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 the earlier versions of Rust compiler
- Now compilable

No Conflict In MIR

```
_1 = const ::HashMap::<&str, &str>::new()
```

```
_2 = const "123";
_4 = &mut _1;
_6 = &_2;
_5 = &(*_6);
_3 = const ::HashMap::<&str, &str>::get_mut::<&str>(move _4, move _5);
```

```
_7 = discriminant(_3);
switchInt(move _7)
```

```
_10 = &mut _1;
_11 = _2;
_13 = const "456";
_12 = &(*_13);
_9 = const ::HashMap::<&str,
&str>::insert(move _10, move _11, move _12)
```

```
_8 = move ((_3 as Some).0: &mut &str);
(*_8) = const "abc";
_0 = const ();
```

3. Security Enhancement

Use-after-free/double free detection

“SafeDrop: Detecting memory deallocation bugs of Rust programs via static data-flow analysis.” TOSEM, 2022.

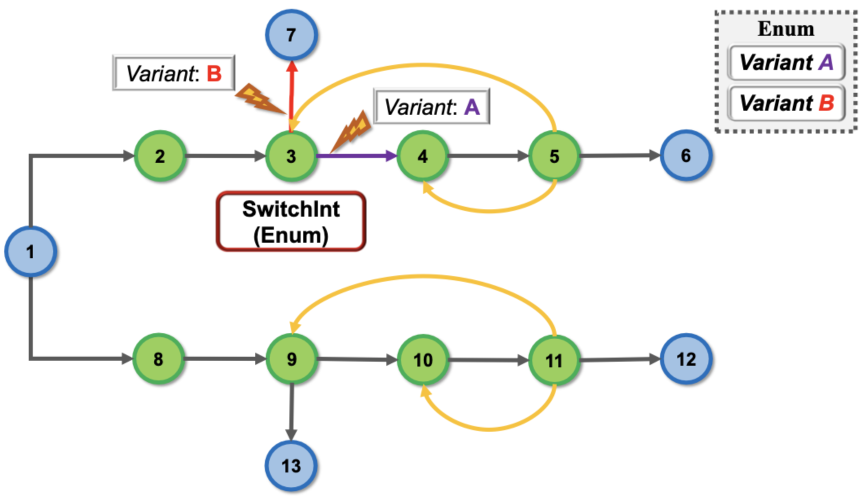
Recall The Bugs Related to Automatic Drop

```
bb0:  _1 = const <std::string::String ::from(const "a... ")
bb1:  resume
bb2:  _5 = &mut _1;
      _4 = const <::String as std::ops::DerefMut>
                  ::deref_mut(move _5)
bb3:  _3 = &mut (*_4);
      _2 = const core::str::<impl str>::as_mut_ptr(move _3)
bb4:  drop(_1)
bb5:  _6 = _2;
      _8 = &_1;
      _7 = const std::string::String::len(move _8)
bb6:  _10 = &_1;
      _9 = const std::string::String::len(move _10)
bb7:  _0 = const ::from_raw_parts(move _6, move _7, move _9)
bb8:  drop(_1)
bb9:  return
```

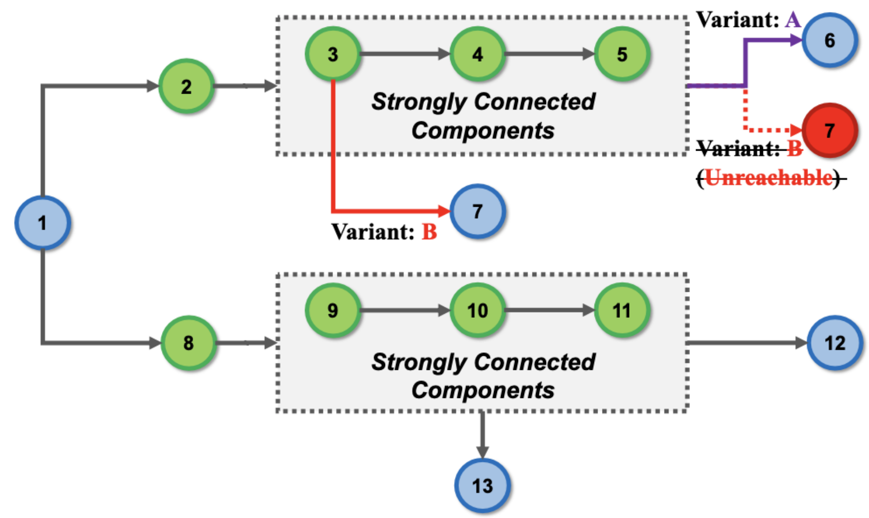
return 0

Path Extraction

- Generate a spanning tree based on the CFG with shrinked SCCs
- Refine the tree to handle corner cases afterwards
  - Enumerate types



Control-flow Graph



Spanning Tree

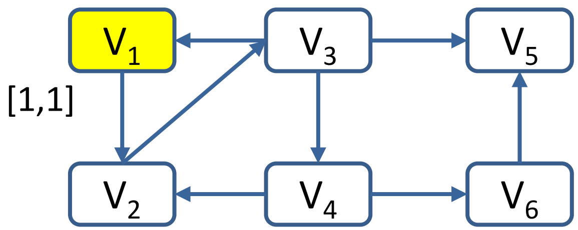
How to Handle Loops?

- Detect strongly-connected components
  - e.g., with Tarjan algorithm

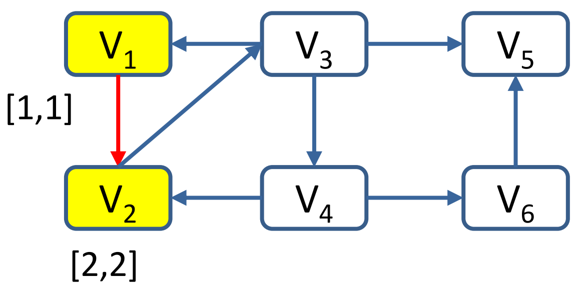
```
DFSVisit(v)
{
    N[v] = c; //first reaching time of node v
    L[v] = c; //first reaching time of the next hop
    c++;
    push v onto the stack;
    for each w in OUT(v) {
        if N[w] == UNDEFINED {
            DFSVisit(w);
            L[v] = min(L[v], L[w]);
        }  else if w is on the stack {
            L[v] = min(L[v], N[w]);
        }
    }
    if L[v] == N[v] { //scc found
        pop vertices off stack down to v;
    }
}
```

Demonstration of Tarjan

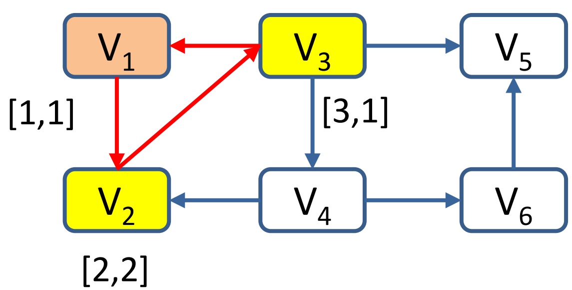
Stack:



V1[1,1]

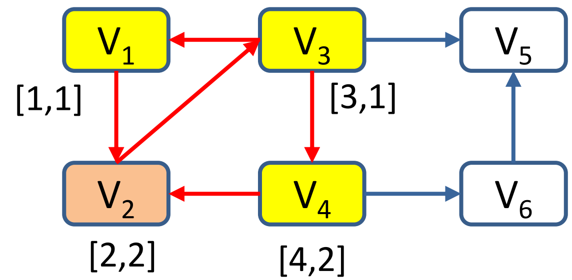


V2[2,2]
V1[1,1]

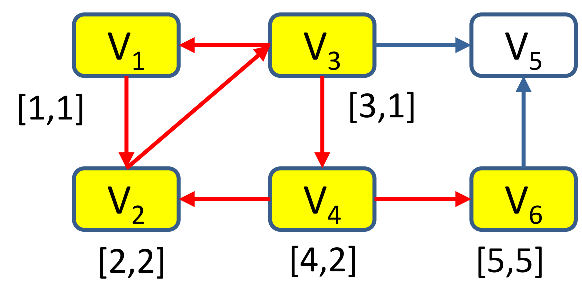


V3[3,1]
V2[2,2]
V1[1,1]

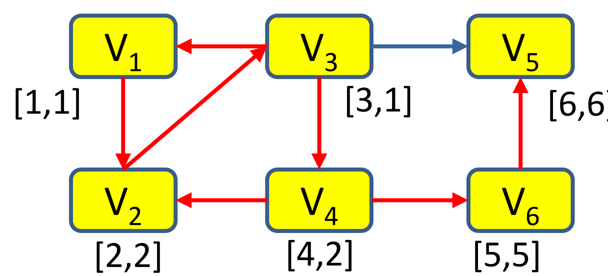
Stack:



V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]

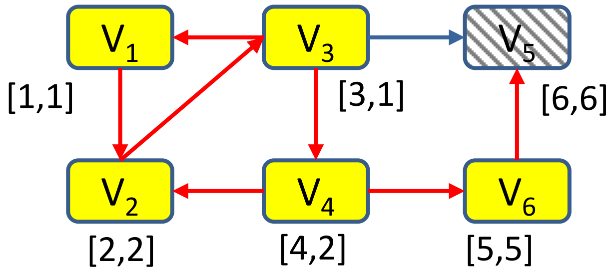


V6[5,5]
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]



V5[6,6]
V6[5,5]
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]

Demonstration of Tarjan

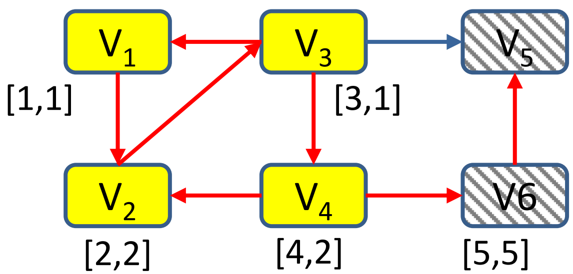


Stack:

V5[6,6]
V6[5,5]
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]

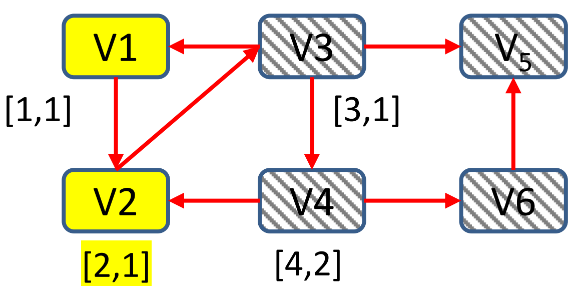
SCC:

{V5}



V6[5,5]
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]

{V5}
{V6}



min(L[v], L[w]);

V2[2,1]
V1[1,1]

{V5}
{V6}
{4,3,2,1}

Alias Analysis

- Similar to the simplified Steensgaard-style analysis

`LValue := Use::Move(RValue)`	`: e.g., a = move b`	`=>`	$S_a = S_a - a,\ \ S_b = S_b \cup a$
`| := Use::Copy(RValue)`	`: e.g., a = b`	`=>`	$S_a = S_a - a,\ \ S_b = S_b \cup a$
`| := Ref/AddressOf(RValue)`	`: e.g., a = &b`	`=>`	$S_a = S_a - a,\ \ S_b = S_b \cup a$
`| := Deref(RValue)`	`: e.g., a = *(b)`	`=>`	$S_a = S_a - a,\ \ S_b = S_b \cup a$
`| := Fn(Move(RValue))`	`: e.g., a = Fn(move b)`	`=>`	$Update(S_a, S_b)$
`| := Fn(Copy(RValue))`	`: e.g., a = Fn(b)`	`=>`	$Update(S_a, S_b)$

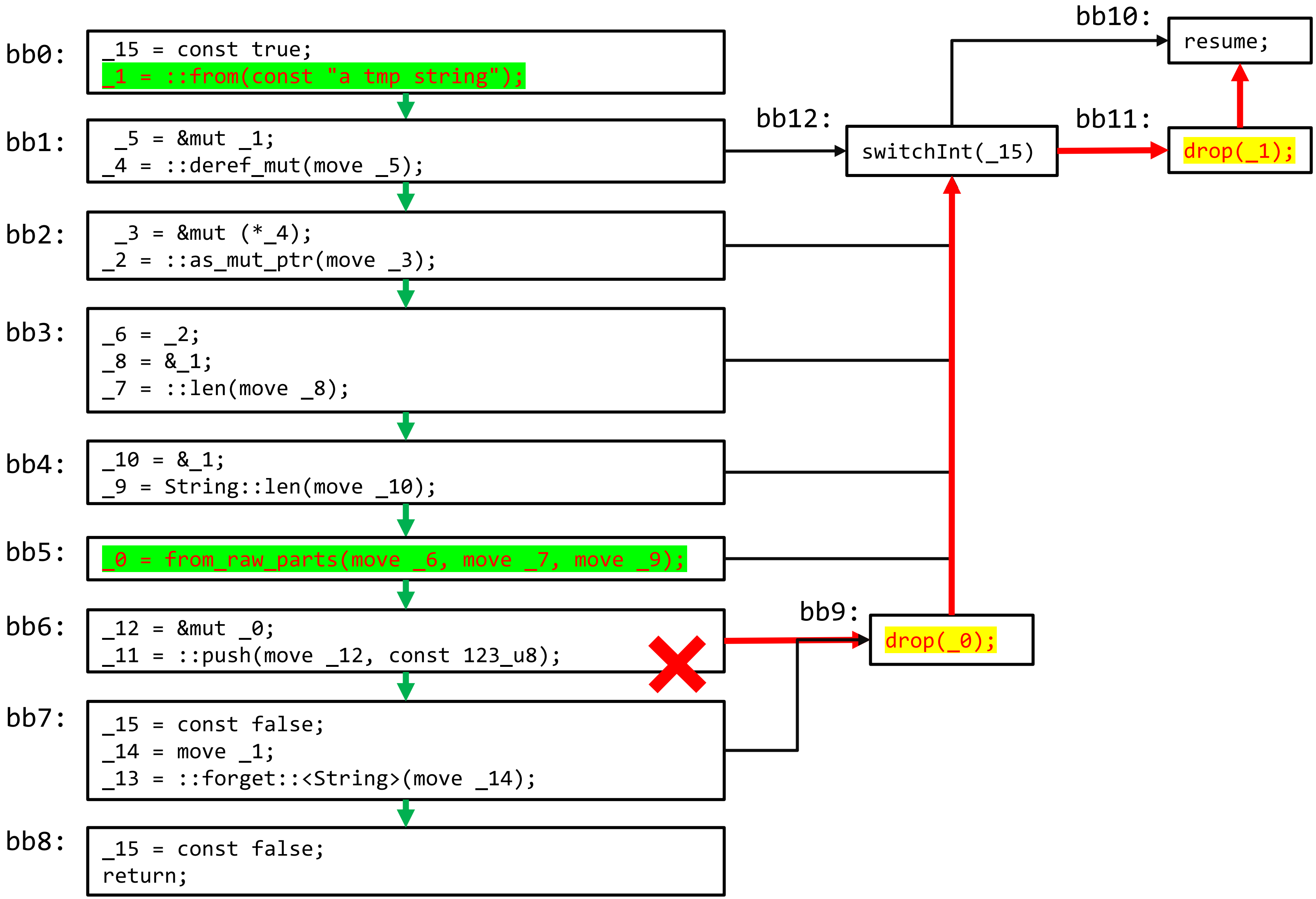
- Example

```
Statement 1:  _2 = &_1;          // alias set:{_1, _2}
Statement 2:  _1 = move _4;      // alias sets:{_1, _4}, {_2}
Statement 3:  _3 = &_1;          // alias sets:{_1, _3, _4}, {_2}
```

Inter-procedure And Field-sensitive

```
enum E { A, B { ptr: *mut u8 } }
struct S { b: E }

fn foo(_1: &mut String) -> S:
    _3 = str::as_mut_ptr(_1);  // alias set: {_3, _1}
    ((_2 as B).0: *mut u8) = move _3;    // alias set: {_2.0, _3, _1}
    discriminant(_2) = 1;  // instantiate the enum type to variant B
    (_0.0: E) = move _2; // alias sets: {_0.0, _2}, {_0.0.0, _2.0, _3, _1}
    return;

fn main():
    _1 = String::from("string"); // alias set: {_1},
    _2 = &mut _1;    // alias set: {_2, _1},
    _3 = foo(move _2); // alias set: {_3.0.0, _2, _1}
    ...
```

Application to Detect Bugs in Unwinding Paths



In-Class Practice

- Write a Rust program with memory leakage bugs
- Emit the MIR code
- Analyze the MIR and discuss how to detect the bug