COMP 737011 - Memory Safety and Programming Language Design

Lecture 13: Static Analysis of Rust

徐 辉
xuh@fudan.edu.cn

Outline

- 1. Background of Static Analysis
- 2. Flow-insensitive Alias Analysis
- 3. Lattice-based Dataflow Analysis
- 4. Path-sensitive Analysis

1. Background of Static Analysis

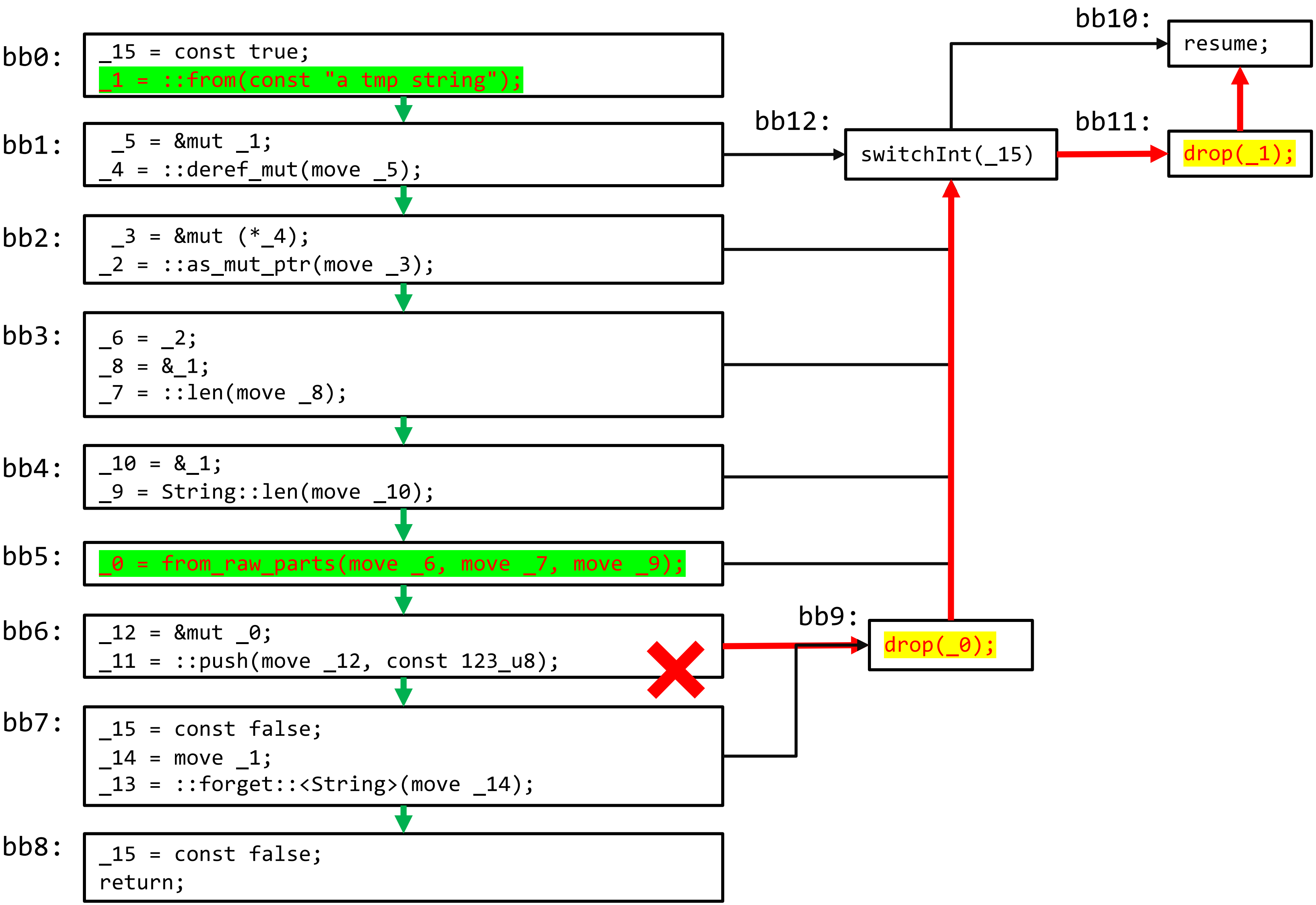
Sample Rust Bug

```
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.push(123);
        std::mem::forget(v)
        return v;

    }
}

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

Bug Analysis with Rust MIR



Dynamic Analysis is Not Enough

- Defects are on uncommon execution paths.
- Executing all paths is infeasible.
- We need static analysis to learn a program's properties without execution.
- Examine the abstraction of a program.
  - a program representation that is simpler to analyze, such as control-flow graph or abstract-syntax tree.

	Dynamic Analysis	Static Analysis
Cost	Proportional to program's execution time	Proportional to program's execution size
Effectiveness	Unsound: may miss errors	Incomplete: may report spurious errors

2. Flow-insensitive Alias Analysis

Andersen-style Analysis

Steensgaard-style Analysis

Common Choices for Alias Analysis

- Flow sensitivity: whether an algorithm considers the order of statements?
- Path sensitivity: whether an algorithm considers the control flow?
- Context sensitivity: whether an algorithm considers how a function is called?

Andersen-style Alias Analysis

- Flow/path/context-insensitive
  - May analysis: it should not miss any alias; false positives are acceptable.
- Represent aliases as equivalence sets
  - e.g., {p, q} {x, y, z} are two alias sets
- How statements affect the alias sets?
  - Subset-based constraints

Form	Constraint	Meaning
a = &b	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
a = b	$a \supseteq b$	$pts(a) \supseteq pts(b)$
a = *b	$a \supseteq * \, b$	$\forall v \in pts(b), pts(a) \supseteq pts(v)$
*a = b	$* \, a \supseteq b$	$\forall v \in pts(a), pts(v) \supseteq pts(b)$

Procedures of Andersen-style Analysis

Step 1. Extract the subset constraints for each statement

Step 2. Init the constraint graph

Step 3. Update the graph with a worklist algorithm

Step 1. Constraint Extraction

Statements

```
p = &a
q = &b
*p = q;
r = &c;
s = p;
t = *p;
*s = r;
```

Form	Constraint
a = &b	$a \supseteq \{b\}$
a = b	$a \supseteq b$
a = *b	$a \supseteq * \, b$
*a = b	$* \, a \supseteq b$

$\Rightarrow$

Constraints

$p \supseteq \{a\}$

$q \supseteq \{b\}$

$*p \supseteq q$

$r \supseteq \{c\}$

$s \supseteq p$

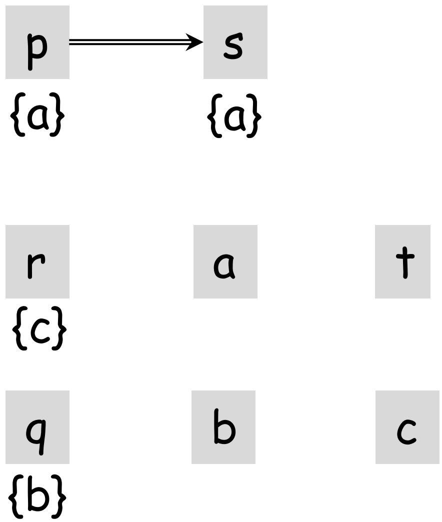
$t \supseteq *p$

$*s \supseteq r$

Step 2. Init The Constraint Graph

- Each node represents a variable and its point-to relationship.
- Each edge represents a subset relationship.

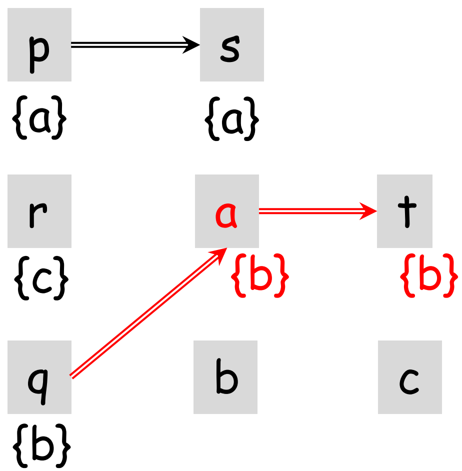
$p \supseteq \{a\}$
$q \supseteq \{b\}$
$*p \supseteq q$
$r \supseteq \{c\}$
$s \supseteq p$
$t \supseteq *p$
$*s \supseteq r$



Step 3. Update the Graph

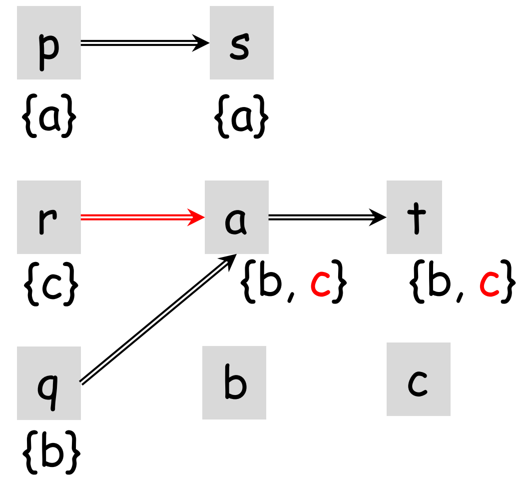
p ⊇ {a}
q ⊇ {b}
*p ⊇ q
r ⊇ {c}
s ⊇ p
t ⊇ *p
*s ⊇ r

1: Worklist: {p, s, r, q}



Result Worklist: {~~p,~~ s, r, q, a, t}

2: Worklist: {s, r, q, a, t}



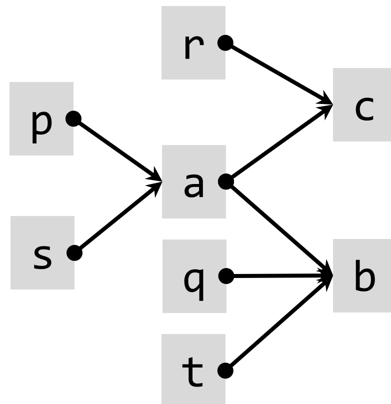
Result Worklist: {~~s~~, r, q, a, t}

```
Let W = { v | pts(v) ≠Ø }
While W not empty
   v ← select from W
   for each a ∈ pts(v) do
      for each constraint p ⊇*v
         add edge a→p, and add a to W if edge is new
      for each constraint *v ⊇ q
         add edge q→a, and add q to W if edge is new
   for each edge v→q do
      pts(q) = pts(q) ∪ pts(v), and add q to W if pts(q) changed
```

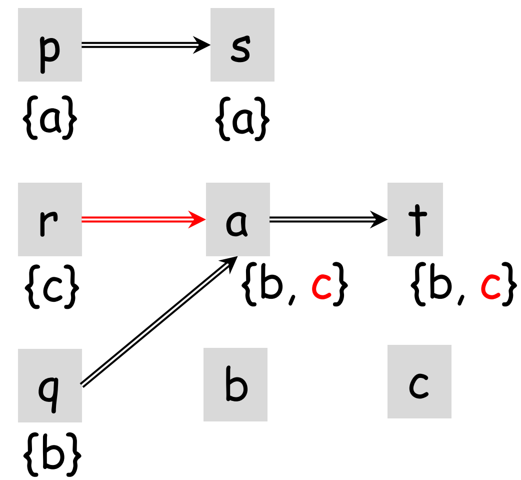
Precision

```
p = &a
q = &b
*p = q;
r = &c;
s = p;
t = *p;
*s = r;
```

Flow-sensitive point-to analysis



Andersen analysis



Flow-sensitive:

```
pts(p) = {a}
pts(q) = {b}
pts(r) = {c}
pts(s) = {a}
pts(t) = {b}
pts(a) = {b, c}
pts(b) = ∅
pts(c) = ∅
```

Andersen:

```
pts(p) = {a}
pts(q) = {b}
pts(r) = {c}
pts(s) = {a}
pts(t) = {b, c}
pts(a) = {b, c}
pts(b) = ∅
pts(c) = ∅
```

*t and c should not be alias

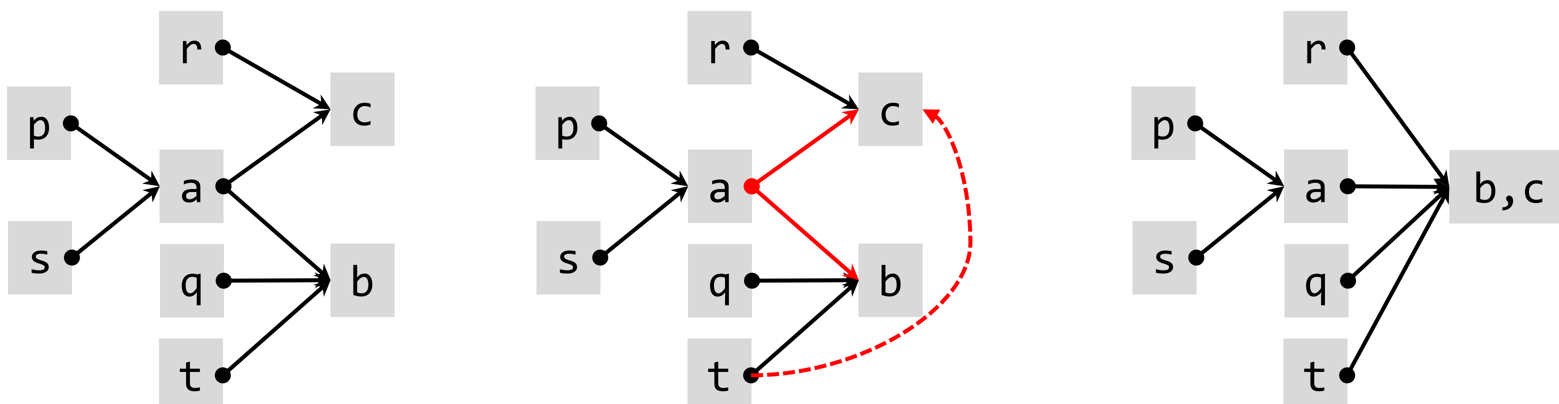
Computational Complexity?

- $O(n^3)$, n is the number of nodes
- In the result, each variable may point to multiple memory units
- We may further restrict each node points to only one abstract location (Steensgaard-style), e.g., x and y point to the same location if *x and *y are alias.

Flow-sensitive point-to analysis

Result of Andersen-style analysis

Result of Steensgaard-style Analysis



Due to flow-insensitivity, if a = &b and a = &c, b and c are recorded as alias.

Steensgaard-Style Analysis

- Use equality constraints instead of subset
- Based on union-find algorithm

Form	Constraint	Meaning	Notes
a = &b	$a \subseteq \{b\}$	$loc(b) \subseteq pts(a)$	Steensgaard
	$a = \{b\}$	$loc(b) = pts(a)$	Simplified Version
a = b	$a = b$	$pts(a) = pts(b)$	
a = *b	$a =* b$	$\forall v \in pts(b), pts(a) = pts(v)$	
*a = b	$* a = b$	$\forall v \in pts(a), pts(v) = pts(b)$	

Problem of simplified version, e.g.,
If a = &c, b = &c, then a and b should not be alias

Union-Find

- Maintain disjoint alias sets
  - Find(x): return the set with x
  - Union(x, y): merge the sets that contains x or y.
- Almost linear complexity: $O(n * \alpha(n))$

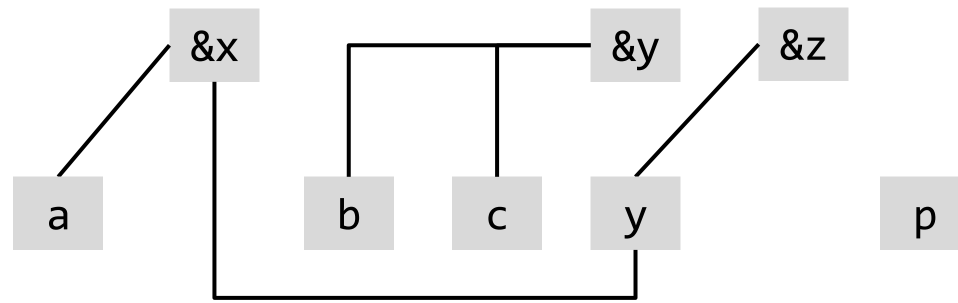
```
while(getPair()!=NULL){
  [p,q] = readPair(p,q);
  pset = find(p);
  qset = find(q);
  if(pset == qset)
      continue;
  else union(p,q);
}
```

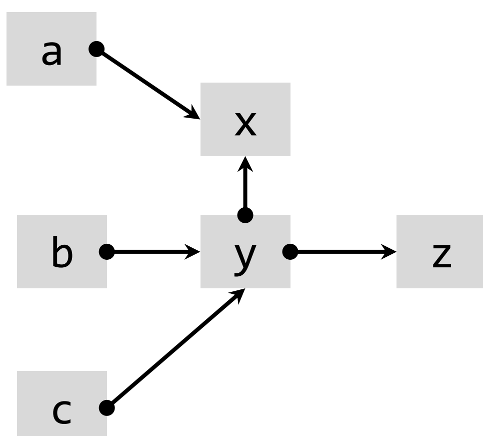
Sample

```
a = &x
b = &y;
if p
  y = &z;
else
  y = &x;
c = &y;
```

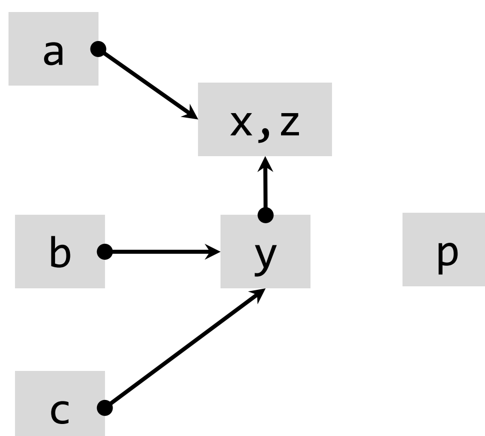
```
{a, &x}
{b, &y}
{p}
{y, &z}

{y, &x}
{c, &y}
```

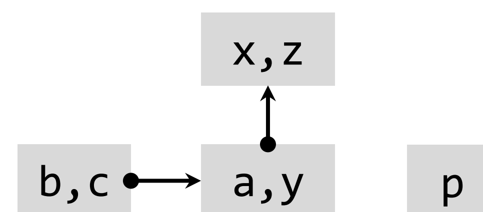




Andersen-style



Steensgaard-style



Simplified Version
(union-find)

Review Flow-sensitivity

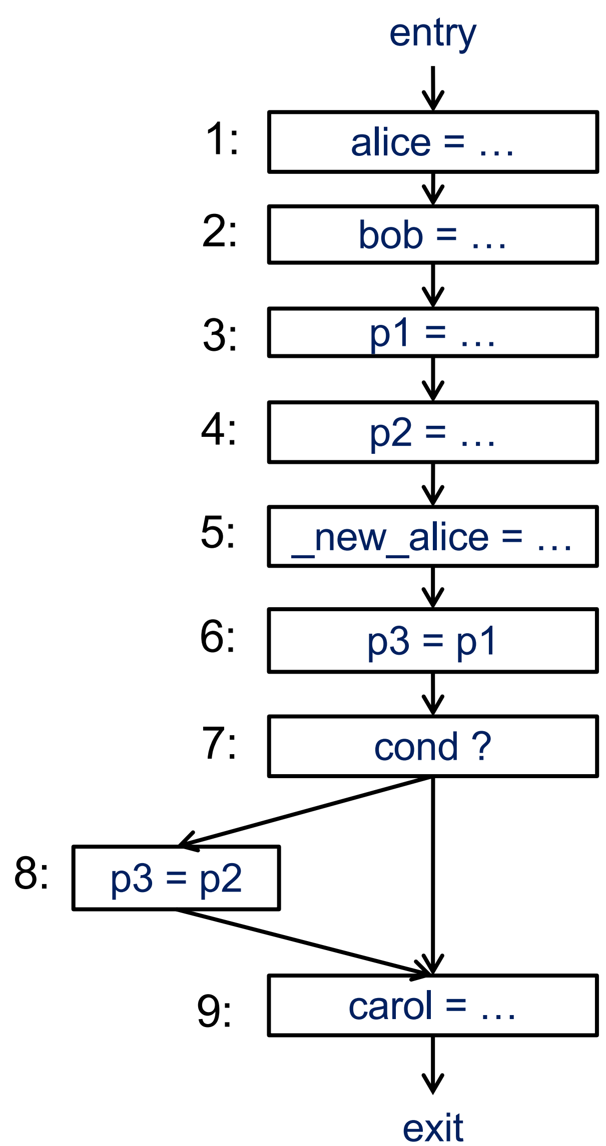
- Flow-insensitive analysis is sound but could incur many false alarms.
- Flow-sensitive analysis maintains the abstraction states for each program point.
- How to handle conditional flows or branches?
  - Merge/split

`let alice = Box::new("alice");`	
`let bob = Box::new("bob");`	
`let p1 = Box::into_raw(alice);`	{p1,alice}, {b}
`let p2 = Box::into_raw(bob);`	{p1,alice}, {p2,bob}
`let _new_alice = unsafe { Box::from_raw(p1)};`	{p1,alice,_new_alice}, {p2,bob}
`let mut p3 = p1;`	{p3,p1,alice,_new_alice}, {p2,bob}
`p3 = p2;`	
`let carol = unsafe { Box::from_raw(p3)};`	{p1,alice,_new_alice}, {carol, p3,p2,bob}
`Drop(carol);`	
`Drop(_new_alice);`	no

3. Lattice-based Dataflow Analysis

Sample Program

```
let alice = Box::new("alice");
let bob = Box::new("bob");
let p1 = Box::into_raw(alice);
let p2 = Box::into_raw(bob);
let _new_alice = unsafe {
    Box::from_raw(p1)};
let mut p3 = p1;
if cond {
    p3 = p2;
}
let carol = unsafe {
    Box::from_raw(p3)};
```

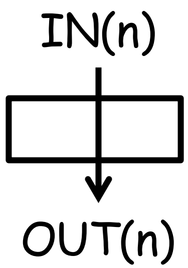


Idea to Reason with a CFG

- Traverse the CFG and update at each program point.
- Transfer function: effect of the statements.
- For each split point:
  - Fork the abstraction states (alias sets)
- For each merge point:
  - Join: combining state from all predecessors.
  - It could also be Meet for other analysis problems, such as must alias analysis (no false positive).
- Traverse the CFG until the state at each program point stops changing.
  - Called "saturated" or "fixed point"

Define the Operations

Transfer Function



$$\text{OUT(n)} = (\text{IN(n)} - \text{KILL(n)}) \cup \text{Gen(n)}$$

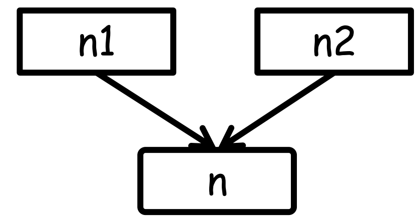
n: $\boxed{\text{x=a}}$

$$\text{KILL(n)} \Rightarrow S_x - x$$
$$\text{Gen(n)} \Rightarrow S_a = S_a \cup x$$

n: $\boxed{\text{cond=?}}$

$$\text{Gen(n)} \Rightarrow \emptyset$$
$$\text{KILL(n)} \Rightarrow \emptyset$$

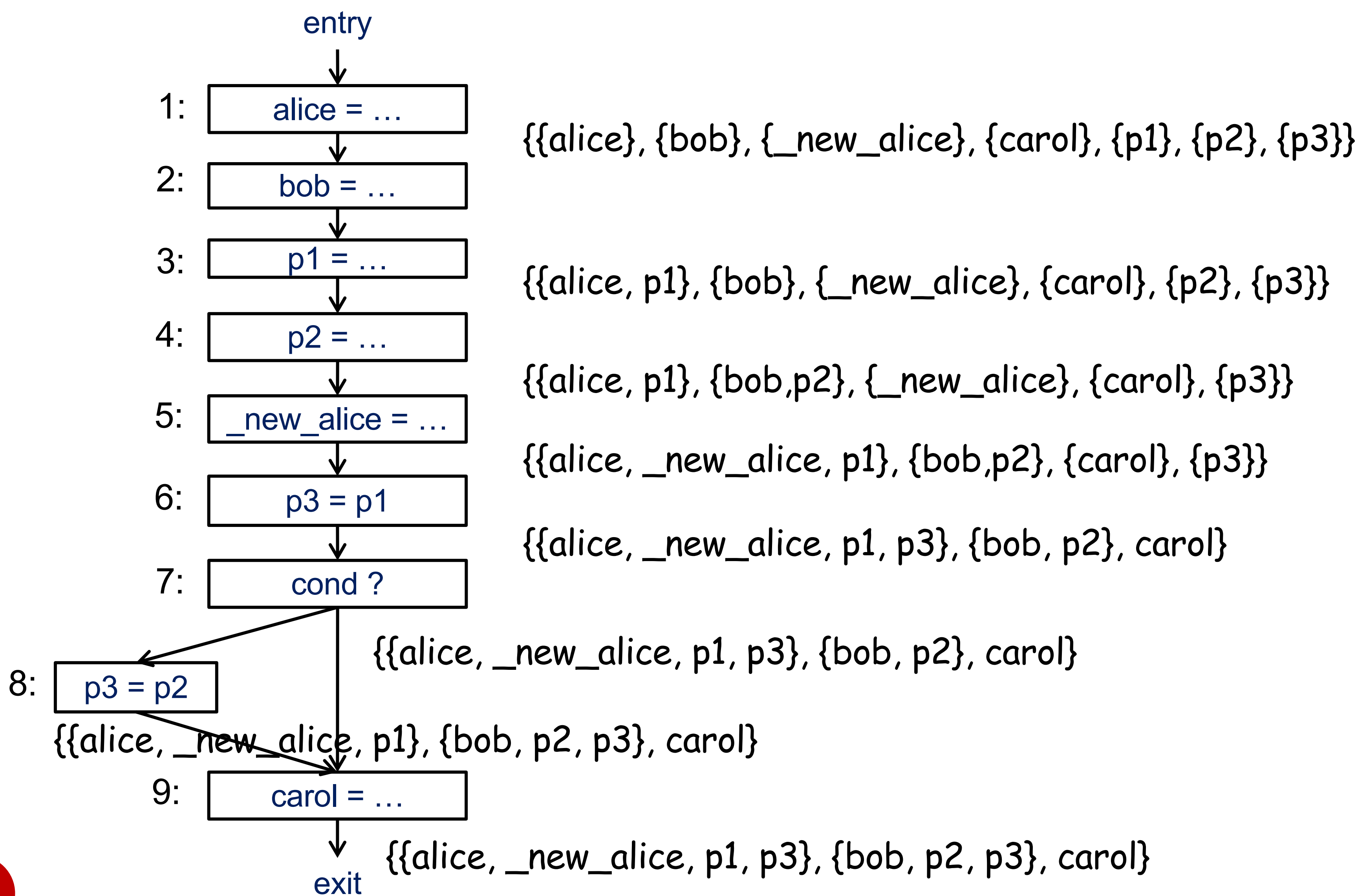
Join



$$\text{IN(n)} = \text{OUT(n1)} \cup \text{OUT(n2)}$$

$$\text{IN}(n) = \bigcup_{n' \in \text{predecessor}(n)} \text{OUT}(n')$$

Analysis


entry
1: alice = …
2: bob = …
3: p1 = …
4: p2 = …
5: _new_alice = …
6: p3 = p1
7: cond ?
8: p3 = p2
9: carol = …
exit
{{alice}, {bob}, {_new_alice}, {carol}, {p1}, {p2}, {p3}}
{{alice, p1}, {bob}, {_new_alice}, {carol}, {p2}, {p3}}
{{alice, p1}, {bob,p2}, {_new_alice}, {carol}, {p3}}
{{alice, _new_alice, p1}, {bob,p2}, {carol}, {p3}}
{{alice, _new_alice, p1, p3}, {bob, p2}, carol}
{{alice, _new_alice, p1, p3}, {bob, p2}, carol}
{{alice, _new_alice, p1}, {bob, p2, p3}, carol}
{{alice, _new_alice, p1, p3}, {bob, p2, p3}, carol}

Overall Algorithm: Chaotic Iteration

```
For (each node n):
    IN[n] = OUT[n] = {disjoint sets of all pointers}
Repeat:
    For(each node n):
```
$$\mathrm{IN}(n) = \cup_{n' \in \mathrm{predecessor}(n)} \mathrm{OUT}(n')$$
```
        OUT(n)=(IN(n)-KILL(n))∪Gen(n)
Until IN[n] and OUT[n] stops changing for all n
```

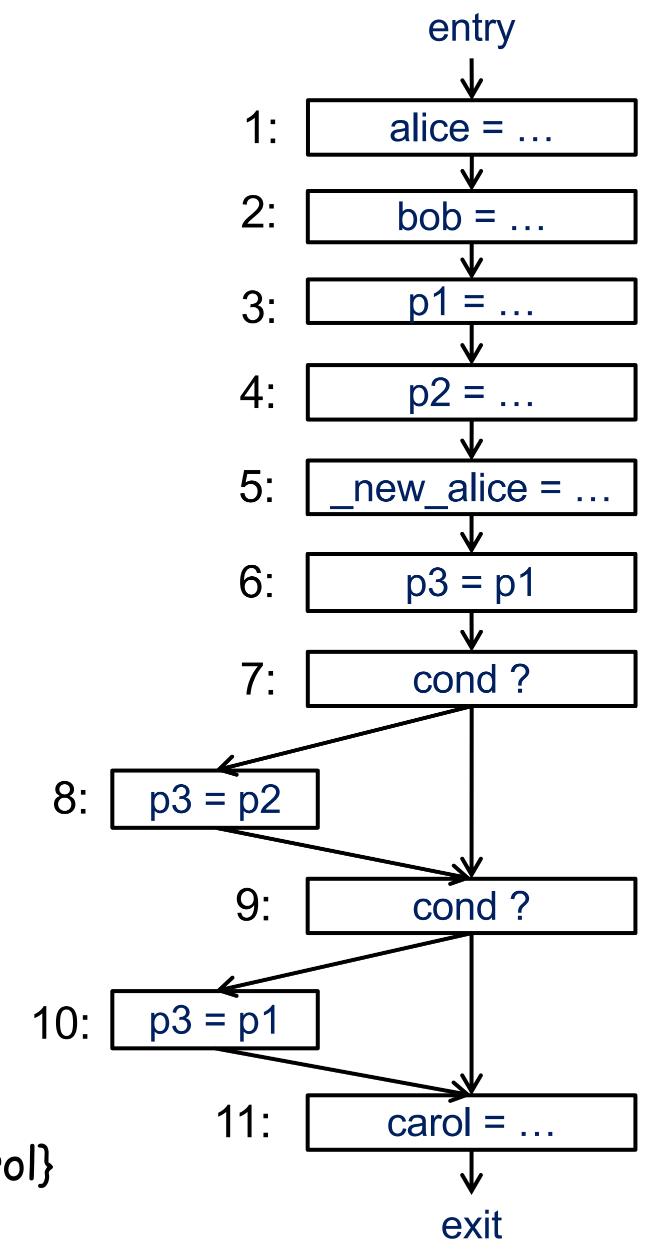
- Does the chaotic iteration algorithm always terminate?
  - Yes, because the number of disjoint alias sets shrinks monotonically
  - In an extreme case, all variables could be alias
  - IN and OUT will stop changing after some iteration

Counter Example

```
let alice = Box::new("alice");
let bob = Box::new("bob");
let p1 = Box::into_raw(alice);
let p2 = Box::into_raw(bob);
let _new_alice = unsafe {
    Box::from_raw(p1)};
let mut p3 = p1;

if cond {
     p3 = p2;
}
if cond {
    p3 = p1;
}
let carol = unsafe {
    Box::from_raw(p3)};
```

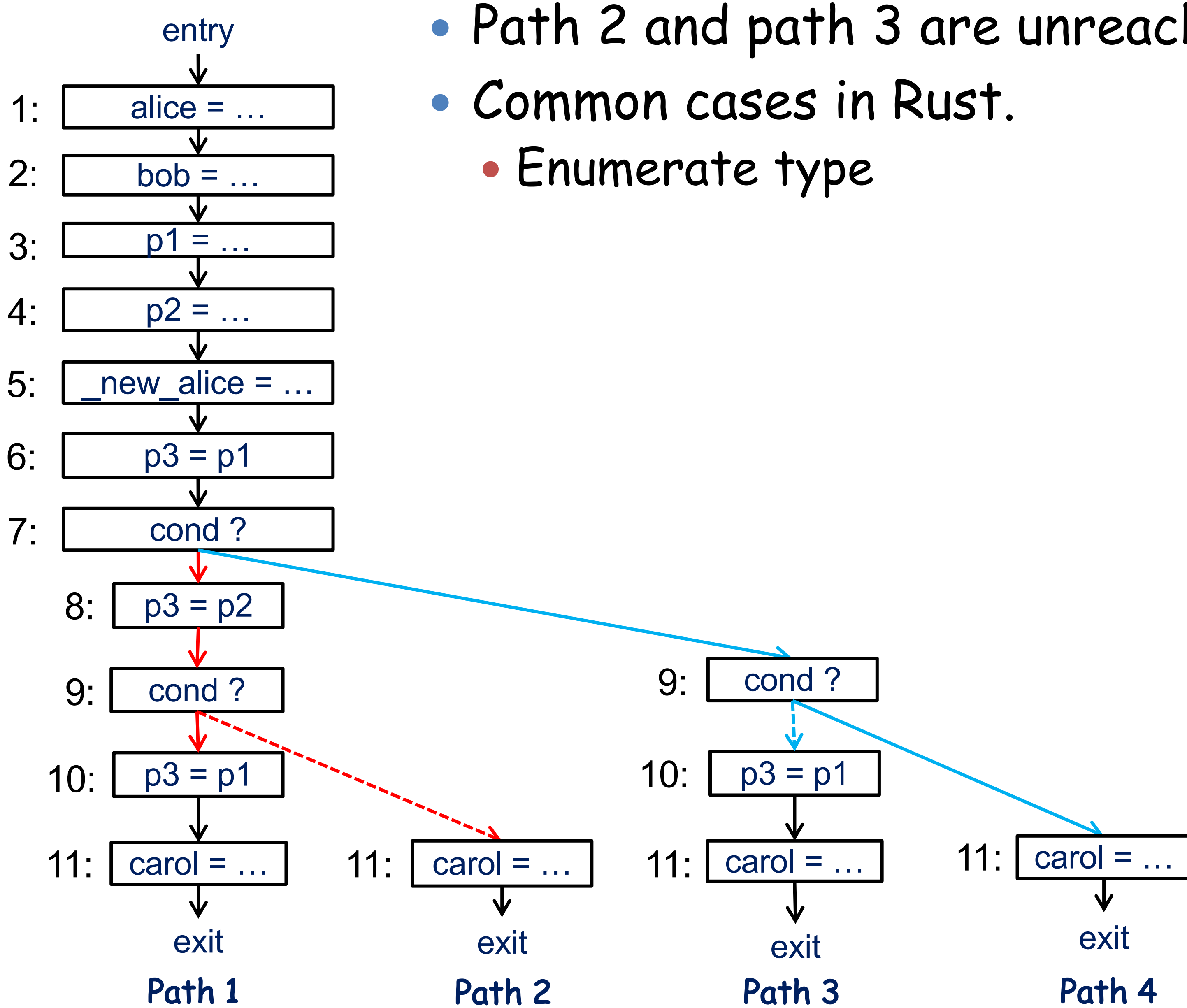
{{alice, _new_alice, p1, p3}, {bob, p2, p3}, carol}



4. Path-Sensitive Analysis

Path-sensitive Analysis

- Path 2 and path 3 are unreachable.
- Common cases in Rust.
  - Enumerate type



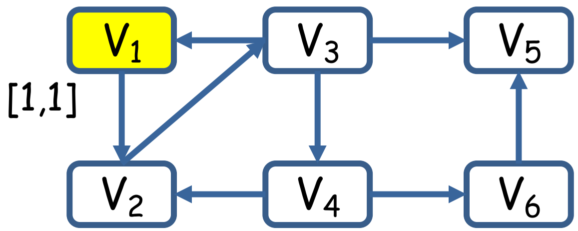
How to Handle Loops?

- 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;
    }
}
```

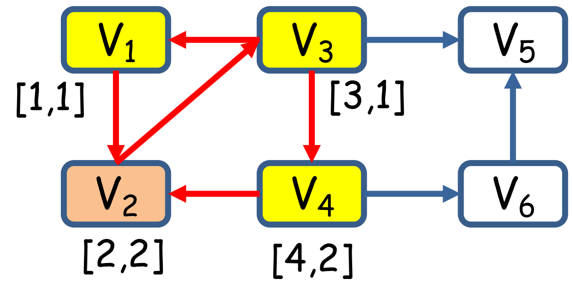
Demonstration of Tarjan

Stack:

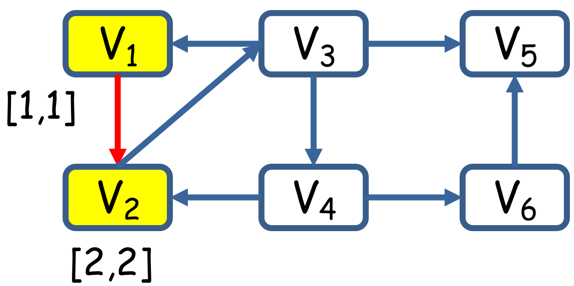


V1[1,1]

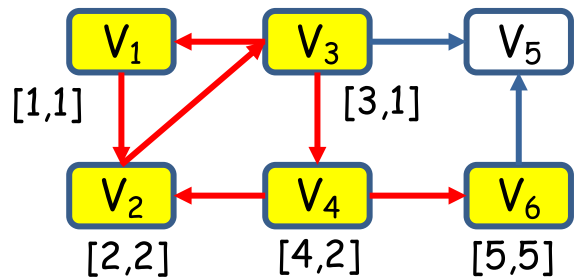
Stack:



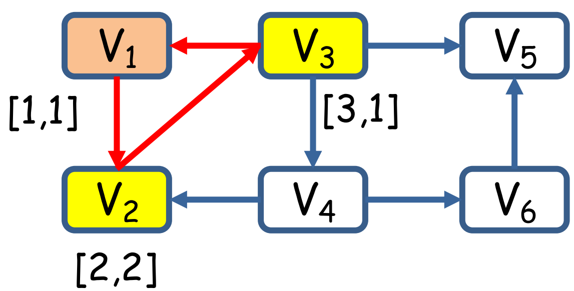
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]



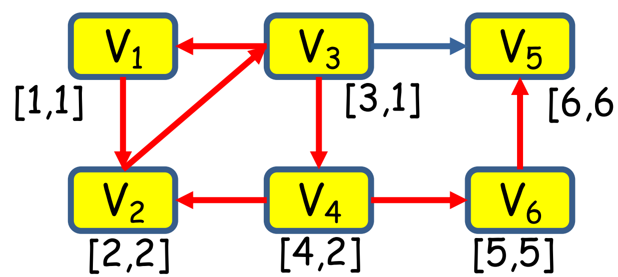
V2[2,2]
V1[1,1]



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



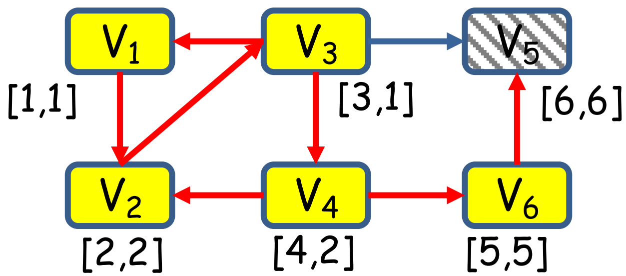
V3[3,1]
V2[2,2]
V1[1,1]



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

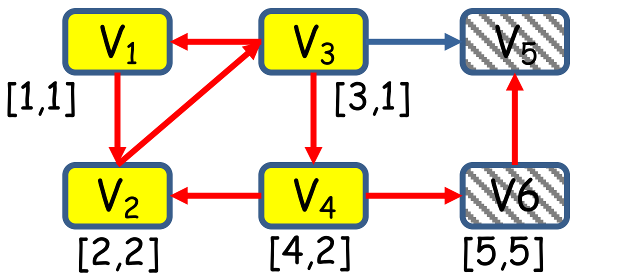
Demonstration of Tarjan

Stack: **SCC:**



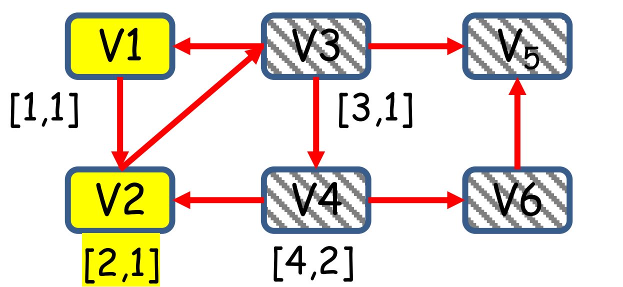
V5[6,6]
V6[5,5]
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]

{V5}



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

{V5}
{V6}



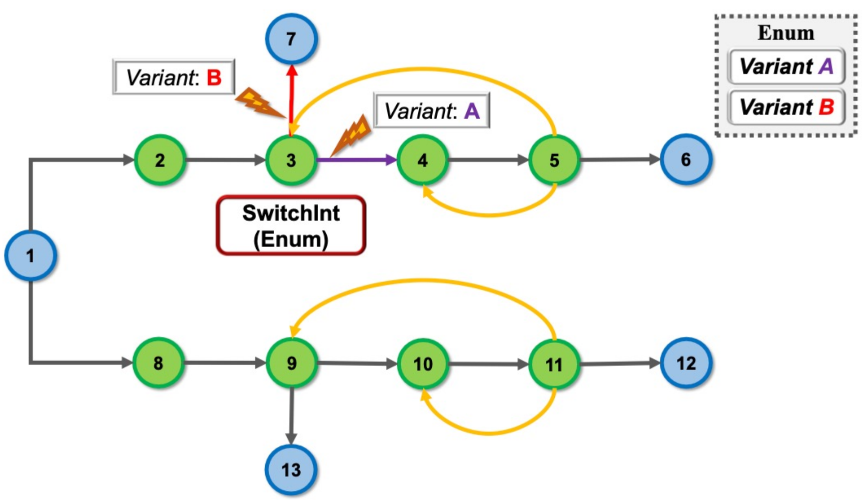
min(L[v], L[w]);

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

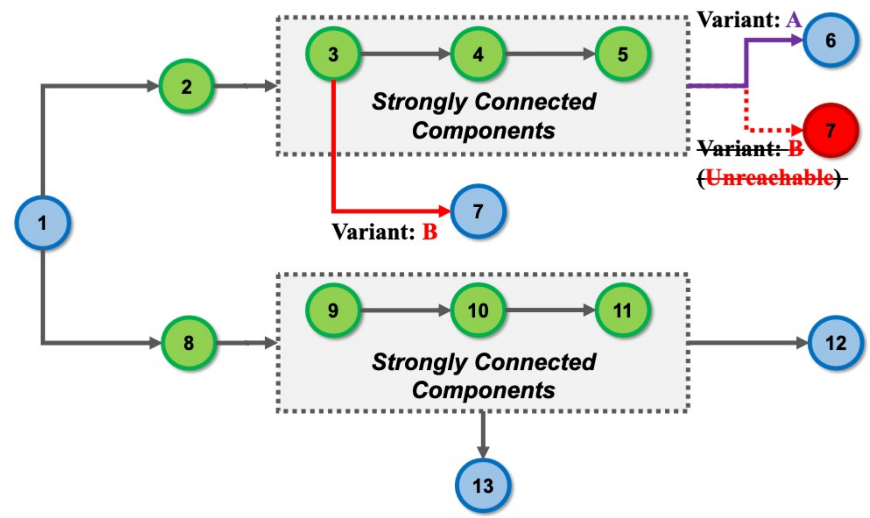
{V5}
{V6}
{4,3,2,1}

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

Alias Analysis

- Similar to the simplified Steensgaard-style analysis

```
LValue := Use::Move(RValue)        : e.g., a = move b      =>
     |  := Use::Copy(RValue)        : e.g., a = b           =>
     |  := Ref/AddressOf(RValue)    : e.g., a = &b          =>
     |  := Deref(RValue)            : e.g., a = *(b)        =>
     |  := Fn(Move(RValue))         : e.g., a = Fn(move b)  =>
     |  := Fn(Copy(RValue))         : e.g., a = Fn(b)       =>
```

$S_a = S_a - a,\ S_b = S_b \cup a$

$S_a = S_a - a,\ S_b = S_b \cup a$

$S_a = S_a - a,\ S_b = S_b \cup a$

$S_a = S_a - a,\ S_b = S_b \cup a$

$Update(S_a, S_b)$

$Update(S_a, S_b)$

- 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}
```

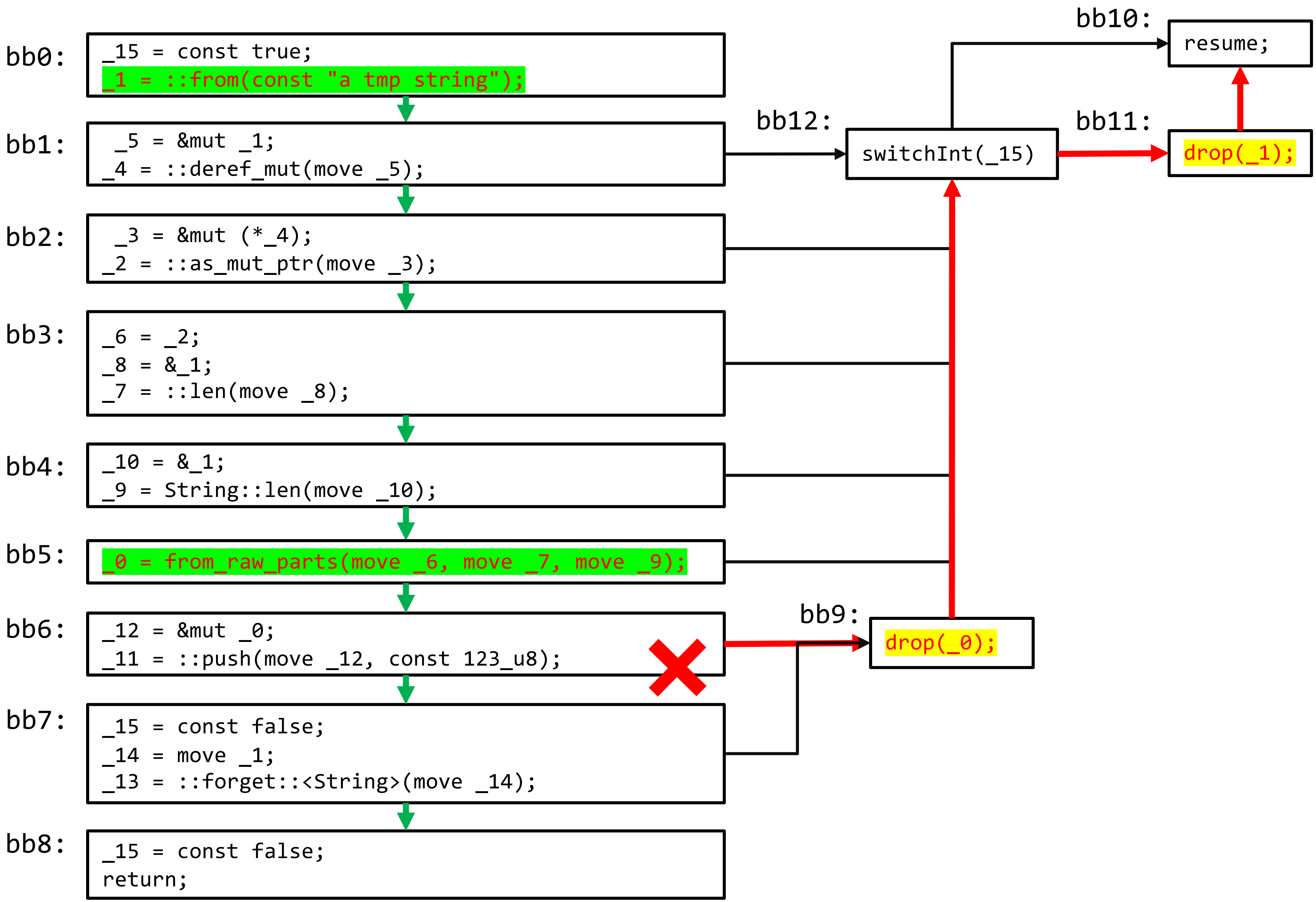
Inter-procedure and Field-sensitive

```
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}
    ...
```

Apply the Approach to Bug Detection



Summary of Static Analysis

- Sound because it considers all possible executions of the program.
- It is incomplete due to Rice's Theorem.
- Many trade-off options in design:
  - Flow-sensitivity
  - Path-sensitivity
  - Context-sensitivity
  - …

More Reference

- [Andersen'94] Andersen, Lars Ole. "Program analysis and specialization for the C programming language." PhD diss., University of Cophenhagen, 1994.
- Steensgaard, Bjarne. "Points-to analysis in almost linear time." *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1996.
- [Hind'01] Michael Hind, Pointer analysis: Haven't we solved this problem yet?. In *Proc. of the 2001 ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering*, 2001.
- http://web-static-aws.seas.harvard.edu/courses/cs252/2011sp/slides/Lec06-PointerAnalysis.pdf.
- https://www.cs.cmu.edu/afs/cs/academic/class/15745-s16/www/lectures/L6-Foundations-of-Dataflow.pdf
- "SafeDrop: Detecting memory deallocation bugs of Rust programs via static data-flow analysis", TOSEM, 2022.