COMP 737011 - Memory Safety and Programming Language Design

Lecture 11: Alias Analysis

Hui Xu
xuh@fudan.edu.cn

Outline

- 1. Alias Analysis Problem
- 2. Flow-insensitive Alias Analysis
- 3. Flow-sensitive Alias Analysis

1. Alias Analysis Problem

Alias Analysis

- To determine whether two pointers or references refer to the same memory location at some point during program execution.
- Alias information is useful for:
  - Compiler optimizations.
  - Program verification and bug detection.

Recall The Bug

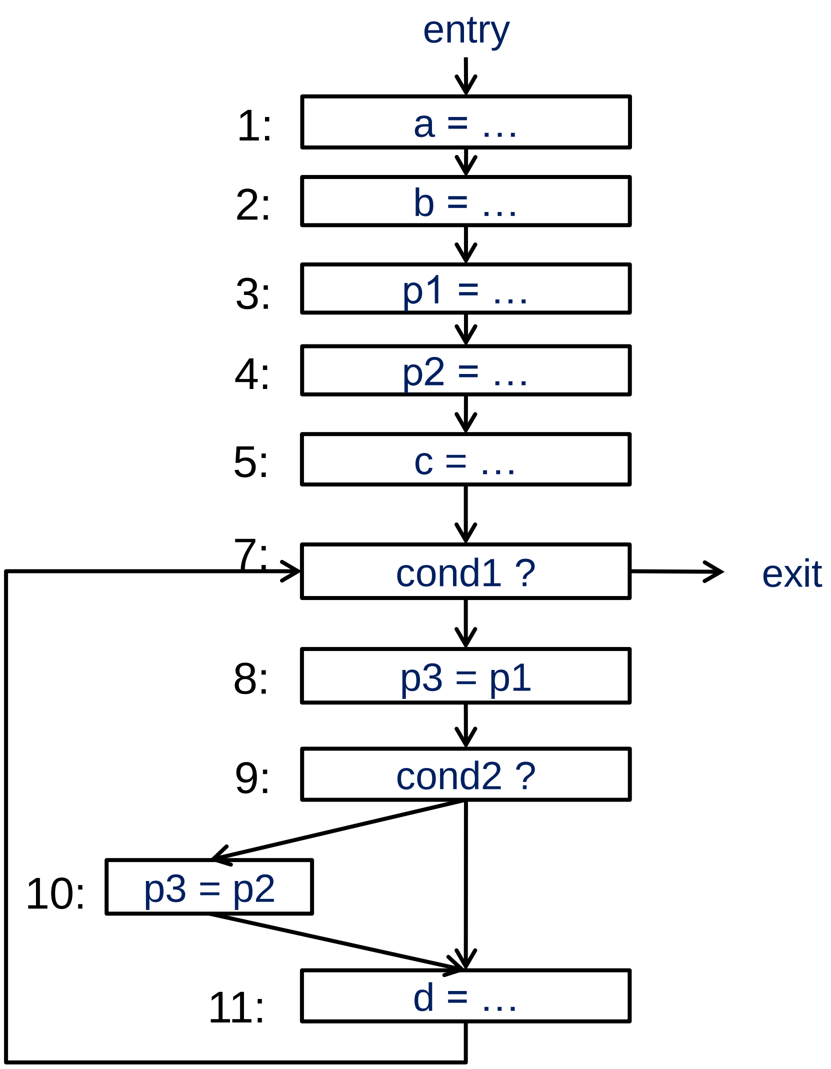
- Whether s and v point to the same memory location?

```
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);
        return v;
    }
}

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

Real-world Programs are More Complicated

```
let a = Box::new("alice");
let b = Box::new("bob");
let p1 = Box::into_raw(alice);
let p2 = Box::into_raw(bob);
let c = unsafe {
    Box::from_raw(p1)
};
while cond1 {
    let mut p3 = p1;
    if cond2 {
        p3 = p2;
    }
    let d = unsafe {
        Box::from_raw(p3)
    };
}
```



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?

2. Flow-insensitive Alias Analysis

Andersen-style Analysis

Steensgaard-style Analysis

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	$* \, a \supseteq b$

Constraints

p ⊇ {a}

q ⊇ {b}

*p ⊇ q

r ⊇ {c}

s ⊇ p

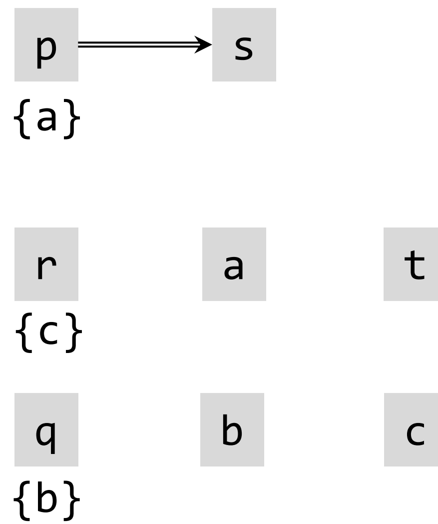
t ⊇ *p

*s ⊇ r

Step 2. Init The Constraint Graph

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

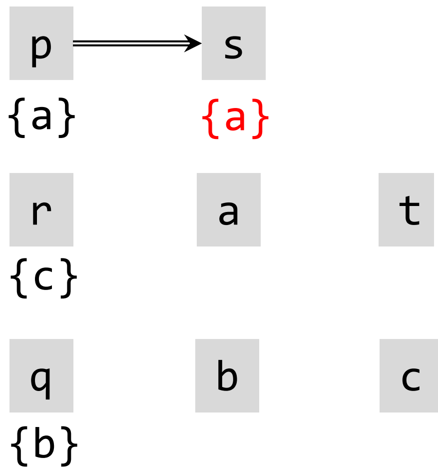
$p \supseteq \{a\}$
$q \supseteq \{b\}$
$*p \supseteq q$
$r \supseteq \{c\}$
$s \supseteq p$
$t \supseteq *p$
$*s \supseteq r$



Step 3. Update the Graph

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

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

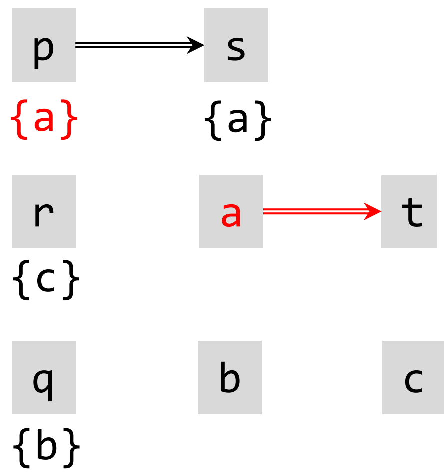



```
Let W = { v | pts(v) ≠∅ }
While W not empty
   v ← select from W
   for each edge v→q do
     pts(q) = pts(q) ∪ pts(v), and add q to W if pts(q) changed
   ...
```

Step 3. Update the Graph

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

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

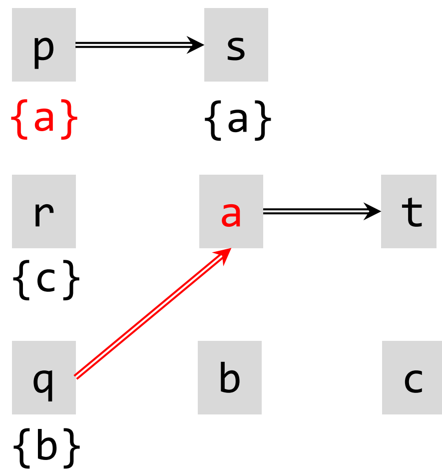



```
Let W = { v | pts(v) ≠∅ }
While W not empty
   v ← select from W
   for each a ∈ pts(v) do
      for each constraint p ⊇*v
         add edge a→p, and add a to W if edge is new
      for each constraint *v ⊇ q
         add edge q→a, and add q to W if edge is new
   for each edge v→q do
      pts(q)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

Step 3. Update the Graph

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

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

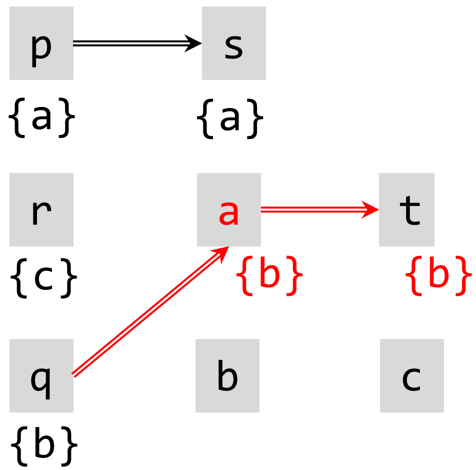



```
Let W = { v | pts(v) ≠∅ }
While W not empty
   v ← select from W
   for each a ∈ pts(v) do
      for each constraint p ⊇*v
         add edge a→p, and add a to W if edge is new
      for each constraint *v ⊇ q
         add edge q→a, and add q to W if edge is new
   for each edge v→q do
      pts(q)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

Step 3. Update the Graph

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

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

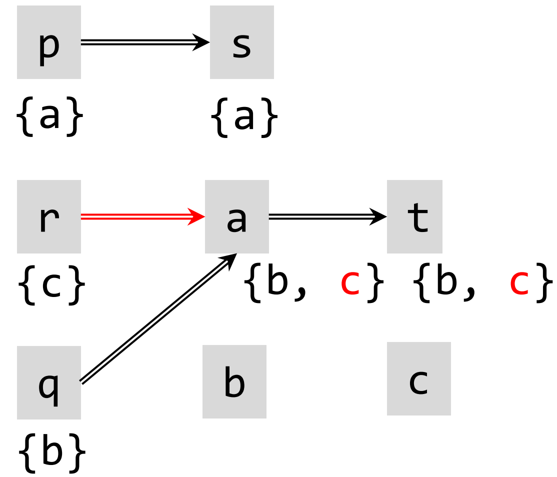



```
Let W = { v | pts(v) ≠∅ }
While W not empty
   v ← select from W
   for each a ∈ pts(v) do
      for each constraint p ⊇*v
         add edge a→p, and add a to W if edge is new
      for each constraint *v ⊇ q
         add edge q→a, and add q to W if edge is new
   for each edge v→q do
      pts(q)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

Step 3. Update the Graph

2: Worklist: : {~~p,~~ r, q, s , a, t} ⟹ 4: Worklist: : {~~p, r, q~~, s , a, t}

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

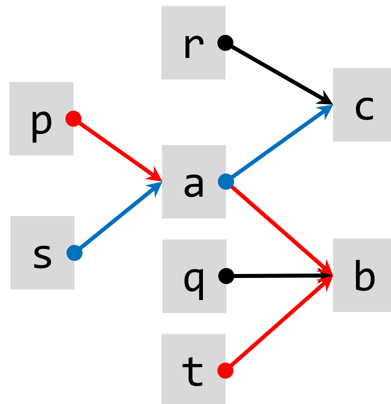



```
Let W = { v | pts(v) ≠∅ }
While W not empty
   v ← select from W
   for each a ∈ pts(v) do
      for each constraint p ⊇*v
         add edge a→p, and add a to W if edge is new
      for each constraint *v ⊇ q
         add edge q→a, and add q to W if edge is new
   for each edge v→q do
      pts(q)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

Precision

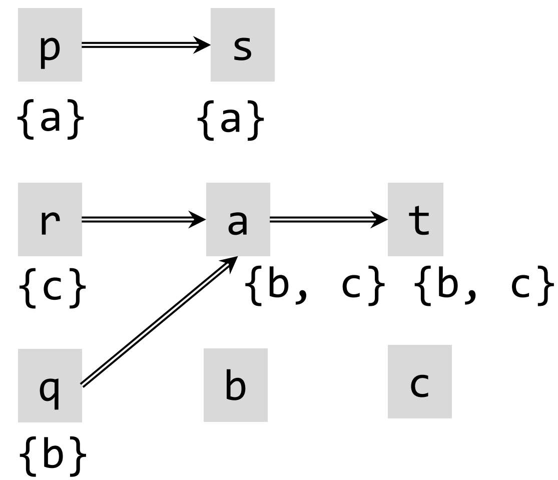
```
p = &a
q = &b
*p = q;
r = &c;
s = p;
t = *p;
*s = r;
```

Flow-sensitive point-to analysis




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

Andersen analysis



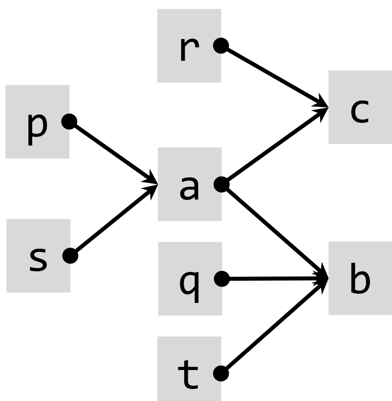

```
pts(p) = {a}
pts(q) = {b}
pts(r) = {c}
pts(s) = {a}
pts(t) = {b, c}
pts(a) = {b, c}
pts(b) = Ø
pts(c) = Ø
```

False positive: *t and c should not be alias

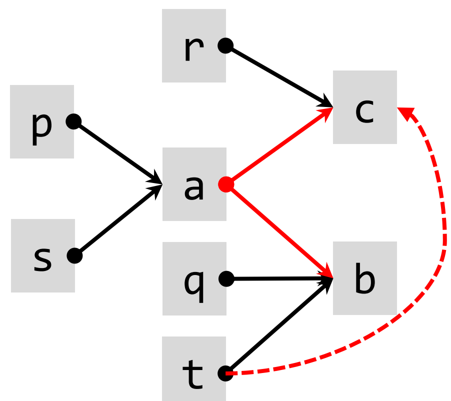
Steensgaard-Style Analysis

- Further restrict each node points to only one abstract location
  - *e.g.,* if *x and *y are alias, x and y point to the same location.

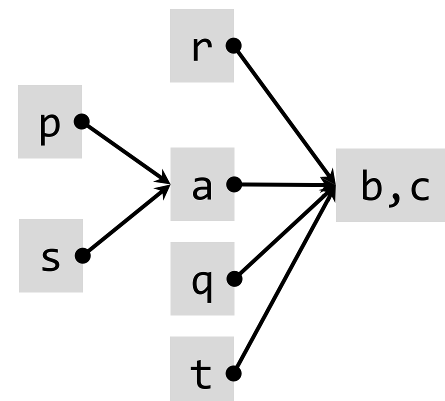
Flow-sensitive point-to analysis



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

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

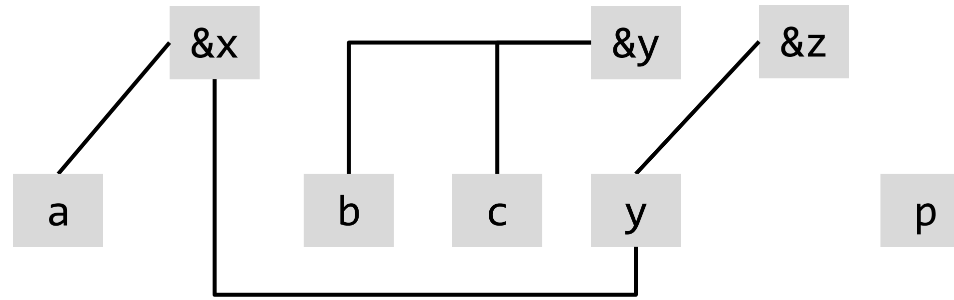
```
while(getPair()!=NULL){
  [p,q] = readPair(p,q);
  pset = find(p);
  qset = find(q);
  if(pset == qset)
      continue;
  else
      union(p,q);
}
```

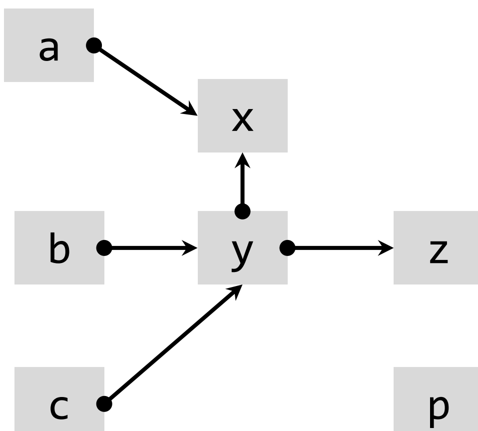
Comparison

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

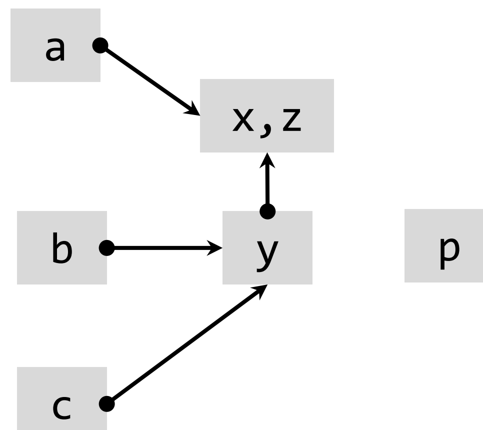
```
{a, &x}
{b, &y}
{p}
{y, &z}

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

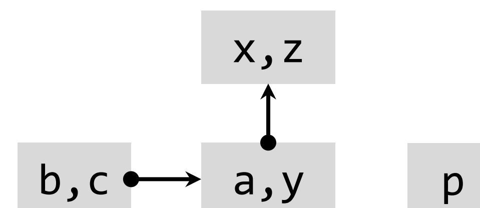




Andersen-style



Steensgaard-style

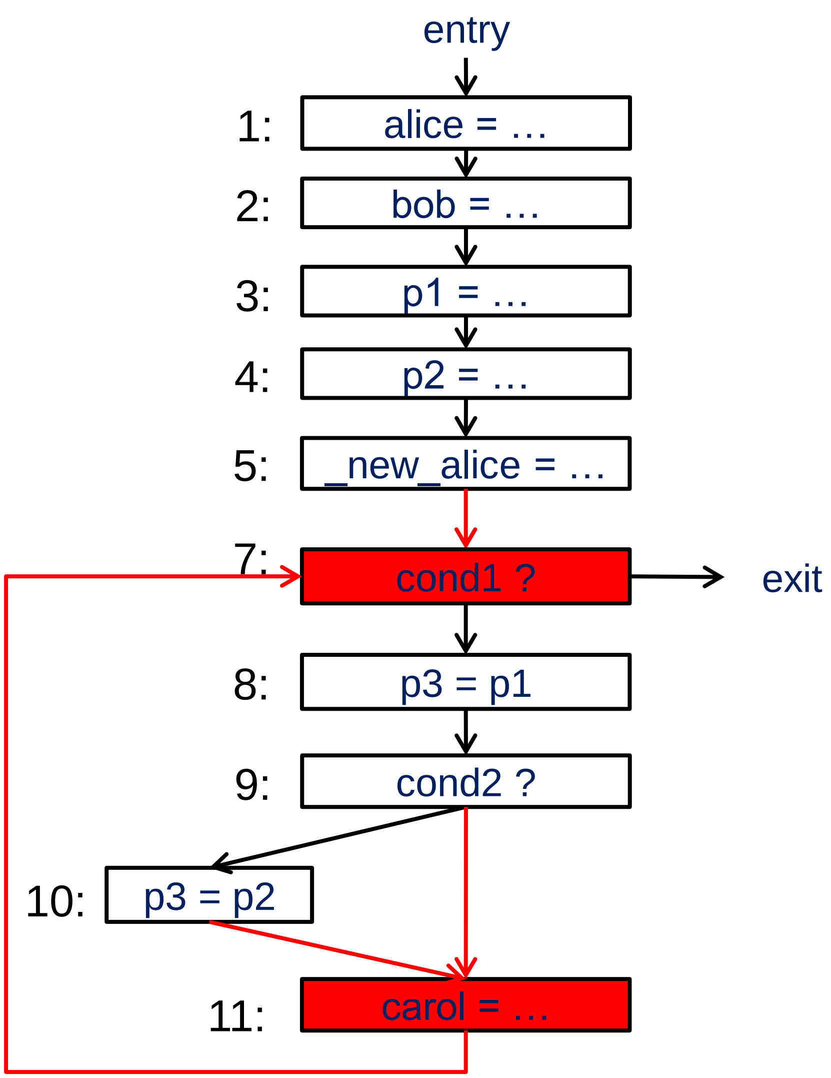


Simplified Version (union-find)

3. Flow-sensitive Alias Analysis

Path Sensitivity: State Duplication or Merging?

```
let a = Box::new("alice");
let b = Box::new("bob");
let p1 = Box::into_raw(alice);
let p2 = Box::into_raw(bob);
let c = unsafe {
    Box::from_raw(p1)
};
while cond1 {
    let mut p3 = p1;
    if cond2 {
        p3 = p2;
    }
    let d = unsafe {
        Box::from_raw(p3)
    };
}
```

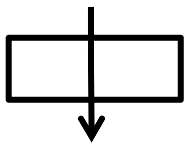


Idea: Lattice-based Approach (Merge)

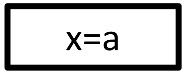
- Traverse the CFG and update at each program point.
- Transfer function: effect of the statements.
- For each split point:
  - Fork the abstraction states (alias sets)
- For each merge point:
  - Join: combining state from all predecessors.
  - It could also be Meet for other analysis problems, such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

Operations

Transfer Function



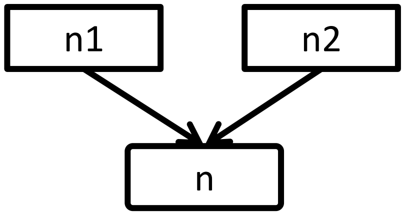
$$\mathrm{OUT(n)} = (\mathrm{IN(n)} - \mathrm{KILL(n)}) \cup \mathrm{Gen(n)}$$

n: 

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

more...

Join



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

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

Overall Algorithm: Chaotic Iteration

```
For (each node n):
    IN[n] = OUT[n] = {disjoint sets of all pointers}
Repeat:
    For(each node n):
```

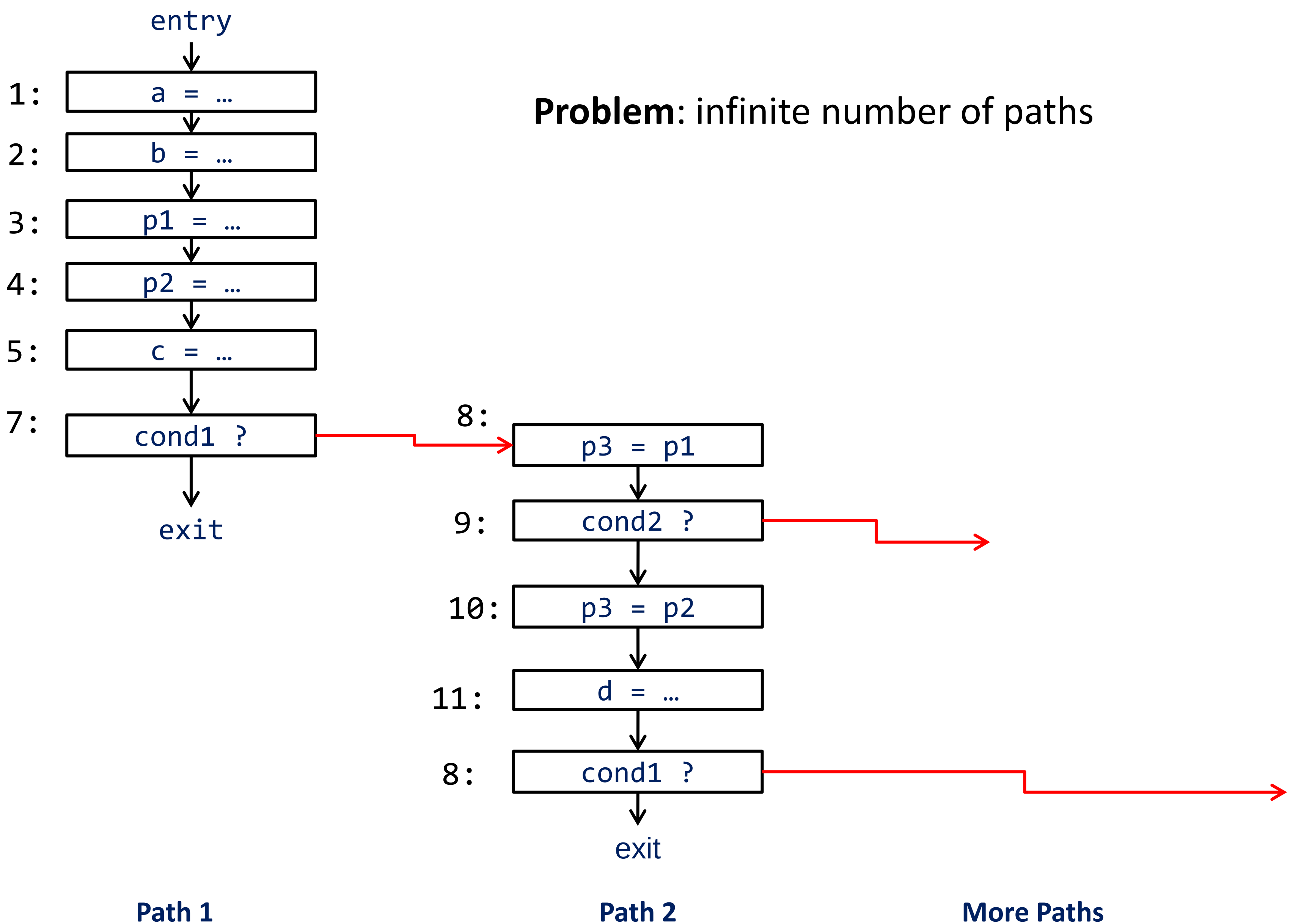
$$\mathrm{IN}(n) = \bigcup_{n' \in \mathrm{predecessor}(n)} \mathrm{OUT}(n')$$

```
        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

Path-sensitive Analysis (Duplicate)

Problem: infinite number of paths



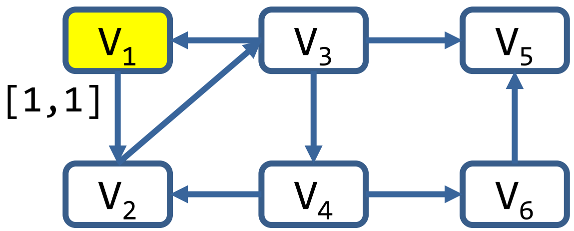
How to Handle Loops?

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

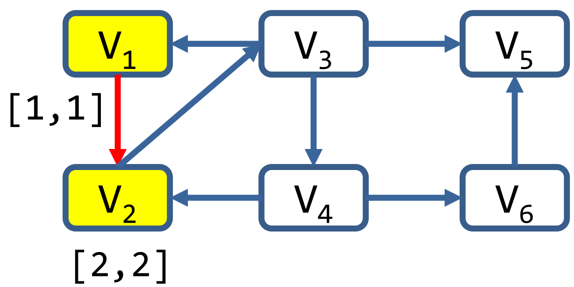
```
DFSVisit(v)
{
    N[v] = c; //first reaching time of node v
    L[v] = c; //first reaching time of the next hop
    c++;
    push v onto the stack;
    for each w in OUT(v) {
        if N[w] == UNDEFINED {
            DFSVisit(w);
            L[v] = min(L[v], L[w]);
        }  else if w is on the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

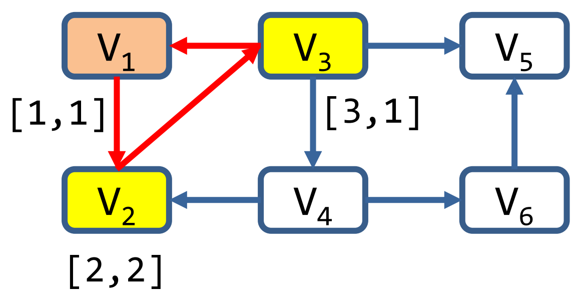
Stack:



V1[1,1]

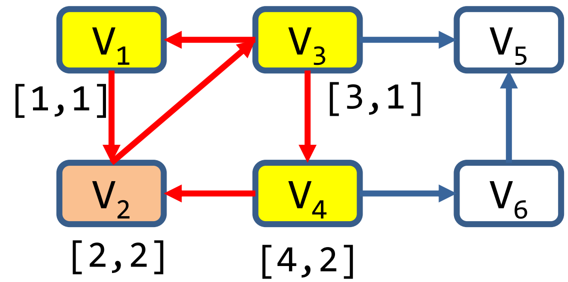


V2[2,2]
V1[1,1]

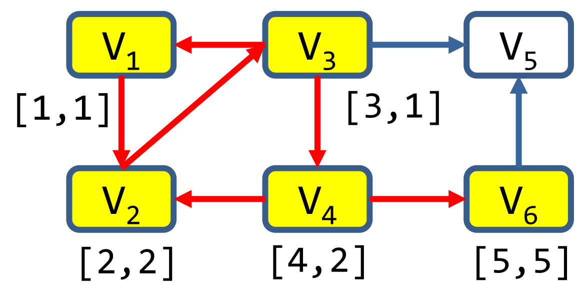


V3[3,1]
V2[2,2]
V1[1,1]

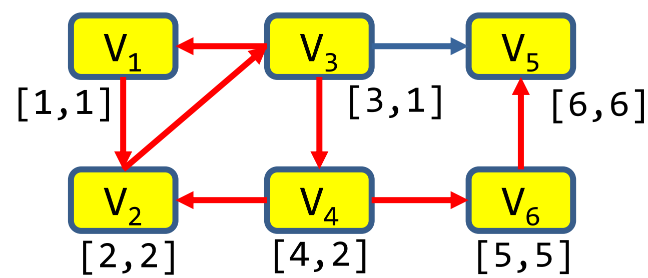
Stack:



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



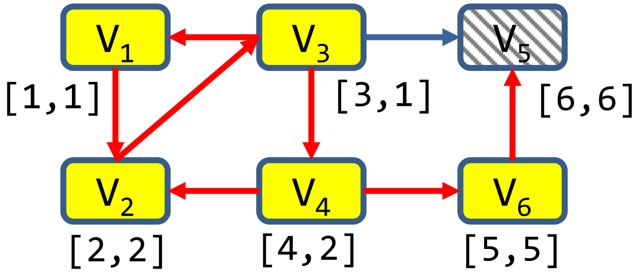
V6[5,5]
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]



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

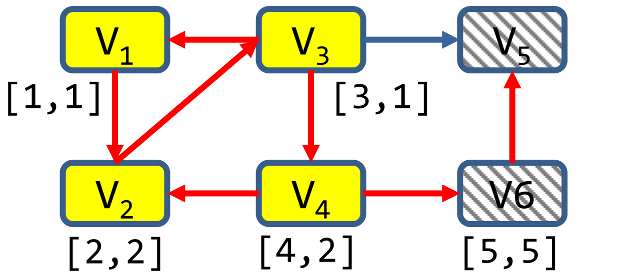
Demonstration of Tarjan

Stack: **SCC:**



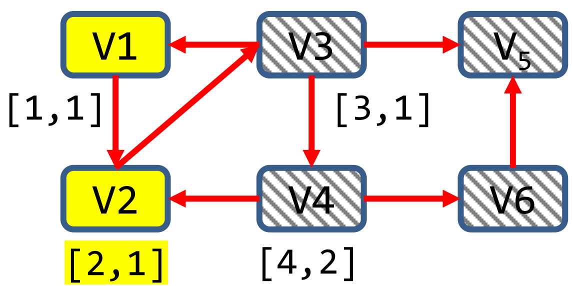
V5[6,6]
V6[5,5]
V4[4,2]
V3[3,1]
V2[2,2]
V1[1,1]

{V5}



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

{V5}
{V6}



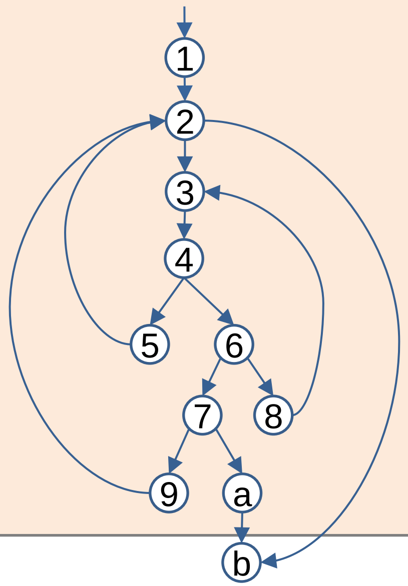

```
min(L[v], L[w]);
```

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

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

Another Example of SCC Analysis

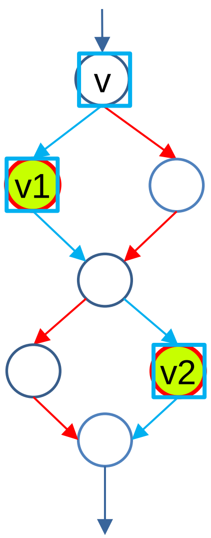
```
for i in 0..2 {
    let x = 1;
    loop {
        match x {
            MyTy::I(v) => {
                let v1 = unsafe {Vec::from_raw_parts(ptr, len, cap)};
                println!{"match MyTy:I(v)..."};
                break; },
            _ => { x+=1; },
        }
        if x == i {
            break;
        }
    }
    if x == i {
        break;
    }
}
```



Control Sensitivity: Condition Satisfiability

```
enum MyTy { I(i32), F(f32), }
fn foo(x:MyTy){
    let mut v =vec!{1,2,3};
    let (ptr, len, cap) = v.into_raw_parts();
    match x {
        MyTy::I(v) => let v1 = unsafe {Vec::from_raw_parts(ptr, len, cap)},
        _ => { },
    }
    match x {
        MyTy::F(v) => let v2 = unsafe {Vec::from_raw_parts(ptr, len, cap)},
        _ => { },
    }
}
```

X cannot be both type MyTy::I and MyTy:: F



The path (v, v1, v2)is unreachable

alias set = { ~~{v, ptr, [x.type!=MyTy::I() AND x.type!=MyTy::F()]~~ }
{v, ptr, v1, [x.type=MyTy::I() AND x.type!=MyTy::F()]}
{v, ptr, v2, [x.type!=MyTy::I() AND x.type=MyTy::F()]}
~~{v, ptr, v1, v2, [x.type=MyTy::I() AND x.type=MyTy::F()]~~}
}