Introduction

- *Aliasing* occurs when different names refer to the same thing
  - Typically, we only care for imperative programs
  - The usual culprit: pointers
- A core building block for other analyses
  - …p.f = 3; // What does p point to?
- Useful for many languages
  - We use Java in these slides
    - Recall that Java objects are heap-allocated, so p.f dereferences p to write to its f field

May Alias Analysis

- p and q *may alias* if it's possible that p and q might point to the same address
- If p may *not* alias q, then a write through p does not affect memory pointed to by q
  - …p.f = 3; x = q.f; // write through p doesn't affect x
- Most conservative may alias analysis?
  - Everything may alias everything else

Points-to Analysis (Emami, Ghiya, Hendren)

- Determine set of locations p may point to
  - E.g., (p, o) means p may point to the object o
  - To decide if p and q alias, see if their points-to sets overlap
  - Can view the points-to information as a graph G
    - Heap objects may point to other heap objects
- Need to name locations in the program
  - Pick a finite set of possible location names
    - No problem with cyclic structures
  - x = new C(...); // where does x point to?
    - (x, {C@257}) "the malloc of C at line 257"

Basic analysis: Statements of interest

- $l = r$ variable assignment
- $l.f = r$ instance field write
- $l = r.f$ instance field read
- $l = \text{new } C$ object allocation
- $l = r_0.m(r_1,\ldots,r_n)$ method call
  - l, r represent local variables
  - f represent a field
  - C represents a class name
  - m represents a method name

Points-to graph: Generation rules

Format: $G \mid s \rightarrow G'$

- Says that given a **points-to graph** G and statement s, the new points-to graph is G'

Basic rules

- $G \mid l = \text{new } C \rightarrow G \cup \{ (l, o_i) \}$ *(stmt on line i)*
- $G \mid l = r \rightarrow G \cup \{ (l, o_i) \mid o_i \in Pt(G, r) \}$
  - where $Pt(G, r) = \{ o_i \mid (r, o_i) \in G \}$

Graph generation rules, cont'd

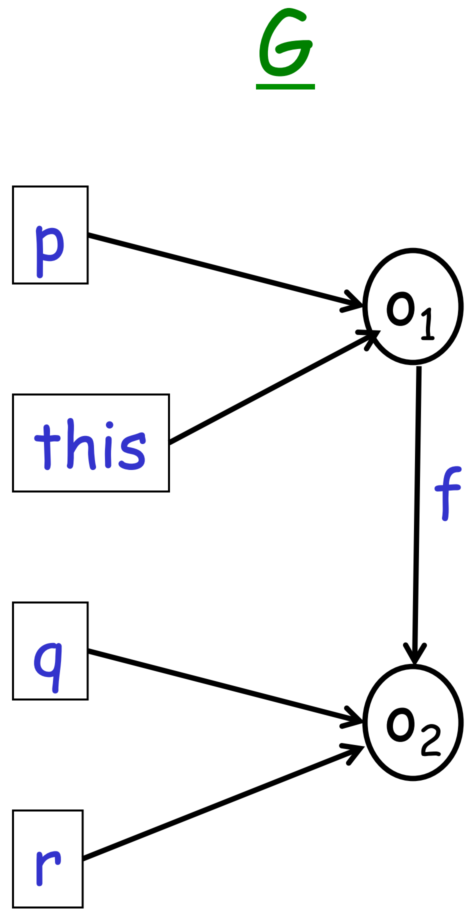
- $G \mid l.f = r \rightarrow G \cup \{(\langle o_i, f\rangle, o_j) \mid$
  $o_i \in Pt(G,l)\ \&\ o_j \in Pt(G,r)\}$

- $G \mid l = r_0.m(r_1,\ldots,r_n) \rightarrow$
  $G \cup \{\ resolve(G,m,o_i,r_0,r_1,\ldots,r_n,l) \mid o_i \in P(G,r_0)\}$

$resolve(G,m,o_i,r_0,r_1,\ldots,r_n,l) =$
let $m_j(p_0,p_1,\ldots,p_n,ret_j) = dispatch(o_i,m)$ in
$\{(p_0,o_i)\} \cup (G \mid p_1=r_1) \cup \ldots \cup (G \mid l=ret_j)$

Example

```
class Y { … }
class X {
  Y f;
  void set(Y r) {
    this.f = r;
  }
  static void main() {
    s1: X p = new X();
    s2: Y q = new Y();
    p.set(q);
  }
}
```



Characterizing this analysis

- This is called Andersen's analysis. It is:
- **Flow-insensitve**
  - The order of the statements is not considered
- **Context-insensitive**
  - The identity of this, and the fact that the statement could be invoked with different call stacks, is not considered
- Flow- and context-sensitivity make the analysis more precise, but more expensive

Example: Flow sensitivity

	Flow-sensitive:	Flow-insensitive:
p = new C;	$\{(p, o_1)\}$	$\{ (p, o_1),$
p = new D;	$\{(p, o_2)\}$	(p, o_2)
p.f = new E;	$\{(p, o_2), (<o_2, f>, o_3)\}$	$(<o_1, f>, o_3)$
		$(<o_2, f>, o_3) \}$

Maintains *G*
per statement

Maintains *G*
for whole program

Taint analysis from points-to information

- Can data from variable x reach variable y?
  - $Pt(G,x) \cap Pt(G,y) \neq \emptyset$

- The answer is conservative, since it does not account for order of statements
  - Will never say 'no' when it could actually happen
  - But might say 'yes' when it cannot