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

• \textit{p} and \textit{q} \textit{may alias} if it’s possible that \textit{p} and \textit{q} might point to the same address

• If \textit{p} may \textit{not} alias \textit{q}, then a write through \textit{p} does not affect memory pointed to by \textit{q}
  - \ldots \texttt{p.f = 3; x = q.f; // write through \textit{p} doesn’t affect \texttt{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
  - $x = \text{new } C(...);$ // 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
- \( f \) represents 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 $i$)
- $G \mid l = r \rightarrow G \cup \{(l,o_i) \mid o_i \in \text{Pt}(G,r)\}$
  - where $\text{Pt}(G,r) = \{ o_i \mid (r,o_i) \in G \}$
Graph generation rules, cont'd

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

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

\[
\text{resolve}(G,m,o_i,r_0,r_1,...,r_n,l) = \]
\[
\text{let } m_j(p_0,p_1,...,p_n,\text{ret}_j) = \text{dispatch}(o_i,m) \text{ in}
\{(p_0,o_i)\} \cup (G|p_1=r_1) \cup ... \cup (G|l=\text{ret}_j)\]

6
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-insensitive**
  - 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

<table>
<thead>
<tr>
<th>Flow-sensitive:</th>
<th>Flow-insensitive:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p = \text{new } C; )</td>
<td>( (p, o_1) )</td>
</tr>
<tr>
<td>( p = \text{new } D; )</td>
<td>( (p, o_2) )</td>
</tr>
<tr>
<td>( p.f = \text{new } E; )</td>
<td>( ((p, o_2), (o_2, f), o_3) )</td>
</tr>
</tbody>
</table>

Maintains \( G \) per statement

Maintains \( G \) for whole program
Taint analysis from points-to information

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

- 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