Points-to Analysis for Java

- Historical roots in points-to analysis for C
  - Steensgaard's algorithm
  - Andersen's algorithm
  - Flow- and context sensitivity
- Field-sensitive analysis for Java
  - Based on Andersen for C augmented with handling for fields and dynamic dispatch

Flow & Context Sensitivity in Analysis

- Flow sensitivity
  - Analysis calculates a different solution for each program point
  - Analysis captures the sequential order of executions of statements
- Context sensitivity
  - Analyze a method separately for different calling contexts (e.g., call sites)

Points-to Analyses for C

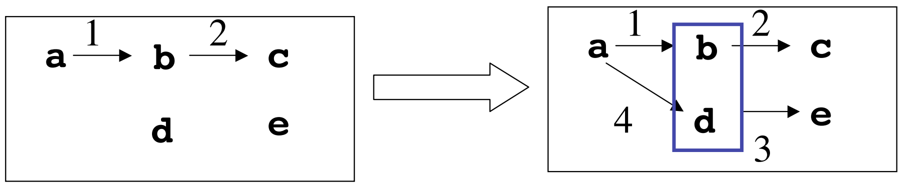
- Popular **flow- and context-insensitive** formulations of points-to analysis
  - Scalable to large codes (MLOC)
  - Good enough for ensuring safety of some optimizations
  - Good for program understanding applications
  - Not great for applications needing precise def-use information (e.g., program slicing, testing)
- Solution procedure utilizes *unification* or *inclusion* constraints
  - P = Q either implies PtsTo(P) = PtsTo(Q) or PtsTo(Q) $\subseteq$ PtsTo(P)
- Extended to points-to analyses for OOPL reference variables

Points-to Analyses for C

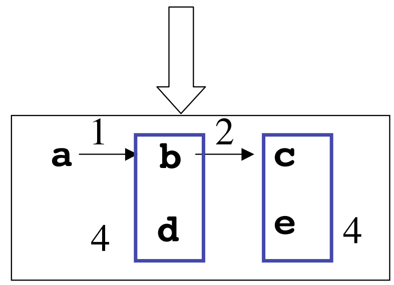
- Steensgaard's algorithm (POPL'96)
  - Uses unification constraints so that for pointer assignments, p = q, algorithm makes PtsTo(p)=PtsTo(q)
    - This union operation is done recursively for multiple-level pointers
  - Reduces the size of the points-to graph (in terms of both nodes and edges)
    - *Almost linear* solution time in terms of program size, O(n) using fast union-find algorithm
    - Imprecision stems from merging points-to sets
  - One points-to set per pointer variable over entire program

cf M Shapiro and S. Horwitz, "Fast and Accurate Flow-insensitive Points-to Analysis" POPL'97

Steensgaard - Example




```
1.a = &b
2.b = &c
3.d = &e
4.a = &d
```



Points-to sets found:

`PtsTo(a)={b,d}`
`PtsTo(b,d)={c,e}`

Steensgaard Solution Procedure - At a glance

- **Find all pointer assignments in program (after conversion to single dereference form)**
- **Form set of points-to graph nodes from pointer variables/fields and variables (in the heap or whose address has been taken)**
  - **Examine each statement, in arbitrary order, and construct points-to edges**
    - **Merge nodes (and edges) where indicated by unification constraints (only 1 out edge labelled * per pointer variable)**
- **After (almost) linear pass over these assignments, points-to graph is complete**

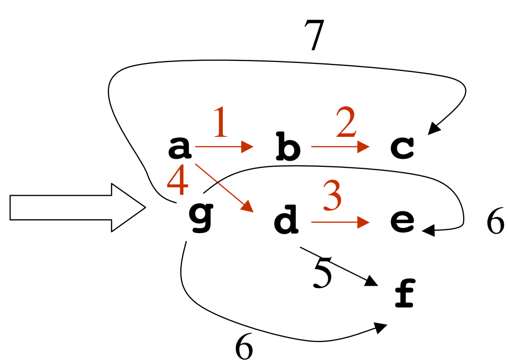
Points-to Analysis for C

- **Andersen's analysis (Thesis 1994)**
  - **Uses inclusion constraints so that for pointer assignments, p = q, algorithm makes**

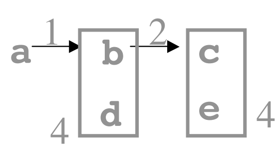
    Pts-to(q) ⊆ Pts-to(p)
  - **Points-to graph is larger (i.e., has more nodes) than Steensgaard's and more precise**
  - **Cubic worst case complexity in program size, $O(n^3)$**
  - **One points-to set per pointer variable over entire program**

Andersen - Example

```
int **a;
int *b,*d,*g;
int c,e,f;
1.a = &b
2.b = &c
3.d = &e
4.a = &d
5.d = &f
6.g = d
7.g = *a
```



Steensgaard solution



Andersen's Solution Procedure - At a glance

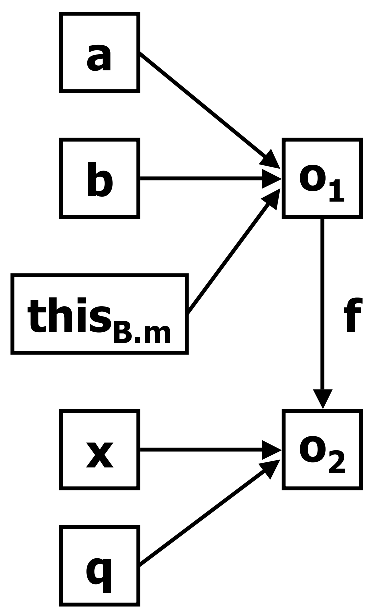
- Find all pointer assignments in program
- Form set of points-to graph nodes from pointer variables/fields and variables on the heap or whose address is taken
  - Examine each statement, in arbitrary order, and construct points-to edges
    - Need to create more edges when see p = q type assignments so that all outgoing points-to edges from q are copied to be outgoing from p (i.e. processing inclusion constraints)
    - If new outgoing edges are added subsequently to q during the algorithm, they must be also copied to p
    - Work results in $O(n^3)$ worst case cost
  - Treat parameter - argument associations like assignment statements

Example of Points-to Analysis

```
class A { void m(X p) {..} }
class B extends A {
    X f;
    void m(X q) { this.f=q;
}
}

B b = new B();
X x = new X();
A a = b;
a.m(x);
```

A.m() not analyzed because it's unreachable.



Constraints Generated

- B b = new B(); PtsTo(b) $\supseteq$ {o_B}
- X x = new X(); PtsTo(x) $\supseteq$ {o_X}
- A a = b; PtsTo(b) $\subseteq$ PtsTo(a)
- a.m(x);
  - Treated like $this_m$ = a; q = x; which generates: PtsTo(a) $\subseteq$ PtsTo($this_m$), PtsTo(x) $\subseteq$ PtsTo(q)
- Then we process the code within m()
  - $this_m$ .f = q
- *A satisfying assignment for these constraints is a points-to solution for this code.*

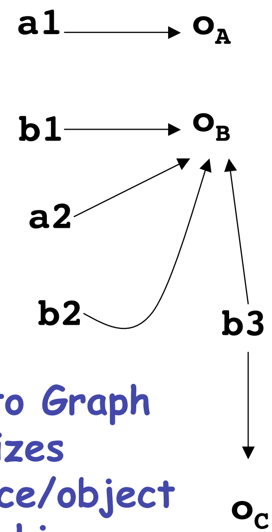
FieldSens Points-to Analysis

- Based on Andersen's points-to analysis but also add object reference fields to points-to graph as suffices for reference variables
  - e.g., class A has fields f,g then p=new A(), means p.f and p.g are in the points-to graph
- Define and solve a system of annotated set-inclusion constraints
  - Handles virtual calls by simulation of run-time method lookup
  - Models the fields of objects
  - Extended BANE (UC Berkeley) constraint solver
- Analyzes only possibly executed code
  - Ignores unreachable code from libraries

Rountev, A. Milnova, B. Ryder, "Points-to Analysis for Java Using Annotated Constraints" OOPSLA'01

FieldSens Example

```
static void main(){
   B b1 = new B();
   A a1 = new A();
   f(b1);
   g(b1);
}
static void f(A a2){
   a2.foo();
}
static void g(B b2){
   B b3 = b2;
   b3 = new C();
   b3.foo();
}
```

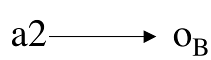


Points-to Graph summarizes reference/object relationships

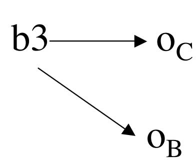
FieldSens Example

cf Frank Tip, OOPSLA'00

```
static void main(){
   B b1 = new B();
   A a1 = new A();
   f(b1);
   g(b1);
}
static void f(A a2){
   a2.foo();
}
static void g(B b2){
   B b3 = b2;
   b3 = new C();
   b3.foo();
}
```




```
class A {
   foo(){..}
}
class B extends A{
   foo()  {…}
}
class C extends B{
   foo()  {…}
}
class D extends B{
   foo(){…}
}
```



FieldSens Characteristics

- **Only analyzes methods *reachable* from main()**
- **Keeps track of individual reference variables and fields**
- **Groups objects by their creation site**
- **Incorporates reference value flow in assignments and method calls**

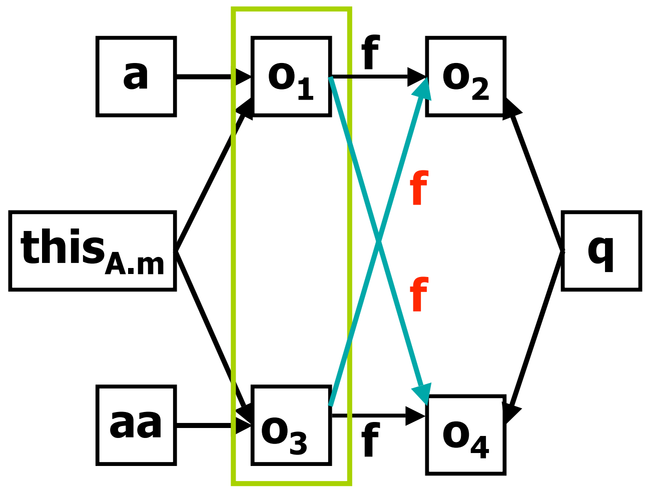
FieldSens Findings

- **Empirical testing found**
  - **Significant precision gains over RTA at call sites found to be polymorphic by CHA**
  - **Could use points-to info in client analysis**
    - **Object read-write information**
    - **Synchronization removal (thread-local)**
    - **Stack allocation (method-local)**

Imprecision of Context Insensitivity

```
class Y extends X { … }

class A {
  X f;
  void m(X q) {
      this.f=q ; }
  }
A a = new A() ;
a.m(new X()) ;
A aa = new A() ;
aa.m(new Y()) ;
```

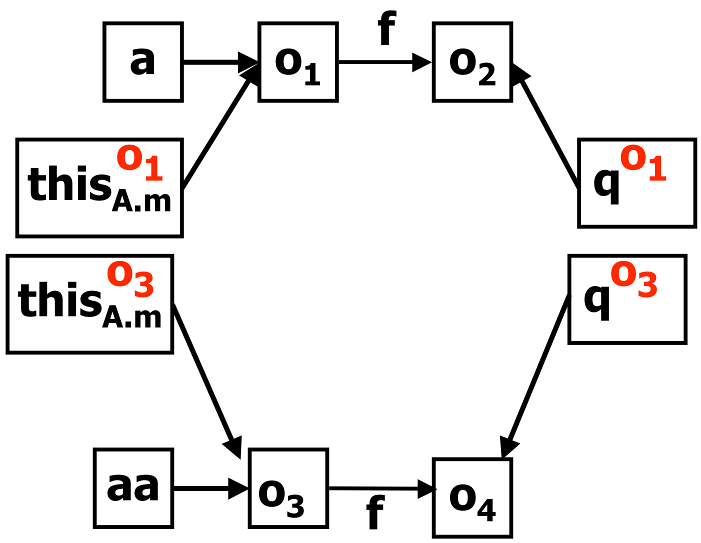


Object-sensitive Analysis

- Form of functional context sensitivity for flow-insensitive analysis of OO languages
- Formulate an object-sensitive Andersen's (points-to) analysis
  - Analysis of instance methods and constructors distinguished for different contexts
  - Receiver objects used to distinguish calling contexts
  - Empirical evaluation vs. context-insensitive FieldSens analysis
    - `this`, formals and return variables (effectively) replicated

Example: Object-sensitive Analysis

```
class A {
  X f;
  void m(X q) {
  this_A.m^o3.f=q^o3; }
  }
A a = new A() ;
a.m(new X()) ;
A aa = new A() ;
aa.m(new Y()) ;
```



ObjSens Findings

- **Precision gains for problems such as def-uses for object fields and side effect analysis (per statement) for practically no additional cost**
- **Clients**
  - **Program test coverage metrics**
  - **Program slicing**
  - **Program understanding tools**

A. Milanova, A. Rountev, and B. Ryder. Parameterized object-sensitivity for points-to and side-effect analyses for Java. In *International Symposium on Software Testing and Analysis*, pages 1–11, 2002.