HARVARD
School of Engineering and Applied Sciences

# Pointer Analysis

*CS252r Spring 2011*

# Today: pointer analysis

- What is it? Why? Different dimensions
- Andersen analysis
- Steensgard analysis
- One-level flow
- Pointer analysis for Java

# Pointer analysis

- What memory locations can a pointer expression refer to?
- **Alias analysis**: When do two pointer expressions refer to the same storage location?
- E.g.,

```
int x;
p = &x;
q = p;
```

  - *p and *q alias,
    as do x and *p, and x and *q

# Aliases

- Aliasing can arise due to
  - Pointers
    - e.g., int *p, i;  p = &i;
  - Call-by-reference
    - void m(Object a, Object b) { … }
      m(x,x); // a and b alias in body of m
      m(x,y); // y and b alias in body of m
  - Array indexing
    - int i,j,a[100];
      i = j; // a[i] and a[j] alias

# Why do we want to know?

- Pointer analysis tells us what memory locations code uses or modifies
- Useful in many analyses
- E.g., Available expressions
  - *p = a + b;
    y = a + b;
  - If *p aliases a or b, then second computation of a+b is not redundent
- E.g., Constant propagation
  - x = 3; *p = 4; y = x;
  - Is y constant? If *p and x do not alias each other, then yes. If *p and x always alias each other, then yes. If *p and x sometimes alias each other, then no.

# Some dimensions of pointer analysis

- Intraprocedural / interprocedural
- Flow-sensitive / flow-insensitive
- Context-sensitive / context-insensitive
- Definiteness
  - May versus must
- Heap modeling
- Representation

# Flow-sensitive vs flow-insensitive

- **Flow-sensitive** pointer analysis computes for each program point what memory locations pointer expressions may refer to
- **Flow-insensitive** pointer analysis computes what memory locations pointer expressions may refer to, at any time in program execution
- Flow-sensitive pointer analysis is (traditionally) too expensive to perform for whole program
  - Flow-insensitive pointer analyses typically used for whole program analyses

# Flow-sensitive pointer analysis is hard

| Alias Mechanism | Intraprocedural May Alias | Intraprocedural Must Alias | Interprocedural May Alias | Interprocedural Must Alias |
|---|---|---|---|---|
| Reference Formals, No Pointers, No Structures | – | – | Polynomial[1, 5] | Polynomial[1, 5] |
| Single level pointers, No Reference Formals, No Structures | Polynomial | Polynomial | Polynomial | Polynomial |
| Single level pointers, Reference Formals, No Pointer Reference Formals, No Structures | – | – | Polynomial | Polynomial |
| Multiple level pointers, No Reference Formals, No Structures | $\mathcal{NP}$-hard | Complement is $\mathcal{NP}$-hard | $\mathcal{NP}$-hard | Complement is $\mathcal{NP}$-hard |
| Single level pointers, Pointer Reference Formals, No Structures | – | – | $\mathcal{NP}$-hard | Complement is $\mathcal{NP}$-hard |
| Single level pointers, Structures, No Reference Formals | $\mathcal{NP}$-hard[14] | Complement is $\mathcal{NP}$-hard | $\mathcal{NP}$-hard[14] | Complement is $\mathcal{NP}$-hard |

Table 1: Alias problem decomposition and classification

*Pointer-induced Aliasing: A Problem Classification*, Landi and Ryder, POPL 1990

# Context sensitivity

- Also difficult, but success in scaling up to hundreds of thousands LOC
  - BDDs see Whaley and Lam PLDI 2004
  - Doop, Bravenboer and Smaragdakis OOPSLA 2009 (see Thurs)

# Definiteness

- May analysis: aliasing that may occur during execution
  - (cf. **must-not alias**, although often has different representation)
- Must analysis: aliasing that must occur during execution
- Sometimes both are useful
  - E.g., Consider liveness analysis for *p = *q + 4;
  - If *p must alias x, then x in kill set for statement
  - If *q may alias y, then y in gen set for statement

# Representation

- Possible representations
  - Points-to pairs: first element points to the second
    - e.g., (p → b), (q → b)
      *p and b alias, as do *q and b, as do *p and *q
  - Pairs that refer to the same memory
    - e.g., (*p,b), (*q,b), (*p,*q), (**r, b)
    - General, may be less concise than points-to pairs
  - Equivalence sets: sets that are aliases
    - e.g., {*p,*q,b}

# Modeling memory locations

- We want to describe what memory locations a pointer expression may refer to
- How do we model memory locations?
  - For global variables, no trouble, use a single "node"
  - For local variables, use a single "node" per context
    - i.e., just one node if context insensitive
  - For dynamically allocated memory
    - Problem: Potentially unbounded locations created at runtime
    - Need to model locations with some **finite abstraction**

# Modeling dynamic memory locations

- Common solution:
  - For each allocation statement, use one node per context
  - (Note: could choose context-sensitivity for modeling heap locations to be less precise than context-sensitivity for modeling procedure invocation)
- Other solutions:
  - One node for entire heap
  - One node for each type
  - Nodes based on analysis of “shape” of heap
    - More on this in later lecture

# Problem statement

- Let's consider flow-insensitive may pointer analysis
- Assume program consists of statements of form
  - p = &a (address of, includes allocation statements)
  - p = q
  - *p = q
  - p = *q
- Assume pointers p,q∈$P$ and address-taken variables a,b∈$A$ are disjoint
  - Can transform program to make this true
  - For any variable v for which this isn't true, add statement $p_v$ = &$a_v$, and replace v with *$p_v$
- Want to compute relation pts : $P \cup A \rightarrow 2^A$
  - Essentially points to pairs

# Andersen-style pointer analysis

- View pointer assignments as **subset constraints**
- Use constraints to propagate points-to information

| **Constraint type** | **Assignment** | **Constraint** | **Meaning** |
|---|---|---|---|
| Base | a = &b | $a \supseteq \{b\}$ | $loc(b) \in pts(a)$ |
| Simple | a = b | $a \supseteq b$ | $pts(a) \supseteq pts(b)$ |
| Complex | a = *b | $a \supseteq *b$ | $\forall v \in pts(b).\ pts(a) \supseteq pts(v)$ |
| Complex | *a = b | $*a \supseteq b$ | $\forall v \in pts(a).\ pts(v) \supseteq pts(b)$ |

# Andersen-style pointer analysis

- Can solve these constraints directly on sets pts(p)

| | |
|---|---|
| `p = &a;` | $p \supseteq \{a\}$ |
| `q = p;` | $q \supseteq p$ |
| `p = &b;` | $p \supseteq \{b\}$ |
| `r = p;` | $r \supseteq p$ |

pts(p) = {a, b}

pts(q) = ∅ {a, b}

pts(r) = ∅ {a, b}

pts(a) = ∅

pts(b) = ∅

# Another example

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

$p \supseteq \{a\}$

$q \supseteq \{b\}$

$*p \supseteq q$

$r \supseteq \{c\}$

$s \supseteq p$

$t \supseteq *p$

$*s \supseteq r$

$pts(p) = \{a\}$

$pts(q) = \{b\}$

$pts(r) = \{c\}$

$pts(s) = \{a\}$

$pts(t) = \{b,c\}$

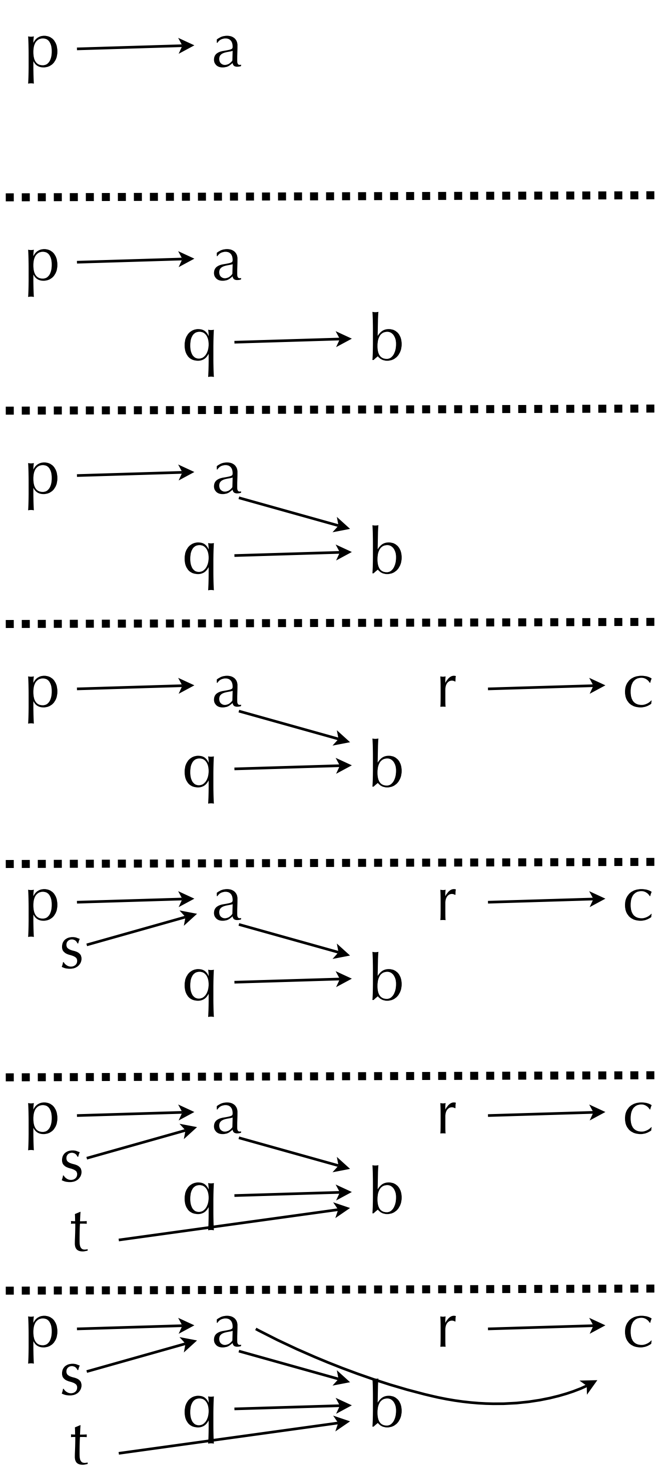
$pts(a) = \{b,c\}$

$pts(b) = \emptyset$

$pts(c) = \emptyset$

# How precise?

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




pts(p) = {a}

pts(q) = {b}

pts(r) = {c}

pts(s) = {a}

pts(t) = {b,c}

pts(a) = {b,c}

pts(b) = ∅

pts(c) = ∅

# Andersen-style as graph closure

- Can be cast as a graph closure problem
- One node for each pts(p), pts(a)

| Assgmt. | Constraint | Meaning | Edge |
|---|---|---|---|
| a = &b | $a \supseteq \{b\}$ | $b \in pts(a)$ | no edge |
| a = b | $a \supseteq b$ | $pts(a) \supseteq pts(b)$ | $b \rightarrow a$ |
| a = *b | $a \supseteq *b$ | $\forall v \in pts(b).\ pts(a) \supseteq pts(v)$ | no edge |
| *a = b | $*a \supseteq b$ | $\forall v \in pts(a).\ pts(v) \supseteq pts(b)$ | no edge |

- Each node has an associated points-to set
- Compute transitive closure of graph, and add edges according to complex constraints

# Workqueue algorithm

- Initialize graph and points to sets using base and simple constraints
- Let $W = \{ v \mid pts(v) \neq \emptyset \}$ (all nodes with non-empty points to sets)
- While W not empty
  - $v \leftarrow$ select from W
  - for each $a \in pts(v)$ do
    - for each constraint $p \supseteq *v$
      - add edge $a \rightarrow p$, and add a to W if edge is new
    - for each constraint $*v \supseteq q$
      - add edge $q \rightarrow a$, and add q to W if edge is new
  - for each edge $v \rightarrow q$ do
    - $pts(q) = pts(q) \cup pts(v)$, and add q to W if pts(q) changed

# Same example, as graph

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

$p \supseteq \{a\}$

$q \supseteq \{b\}$

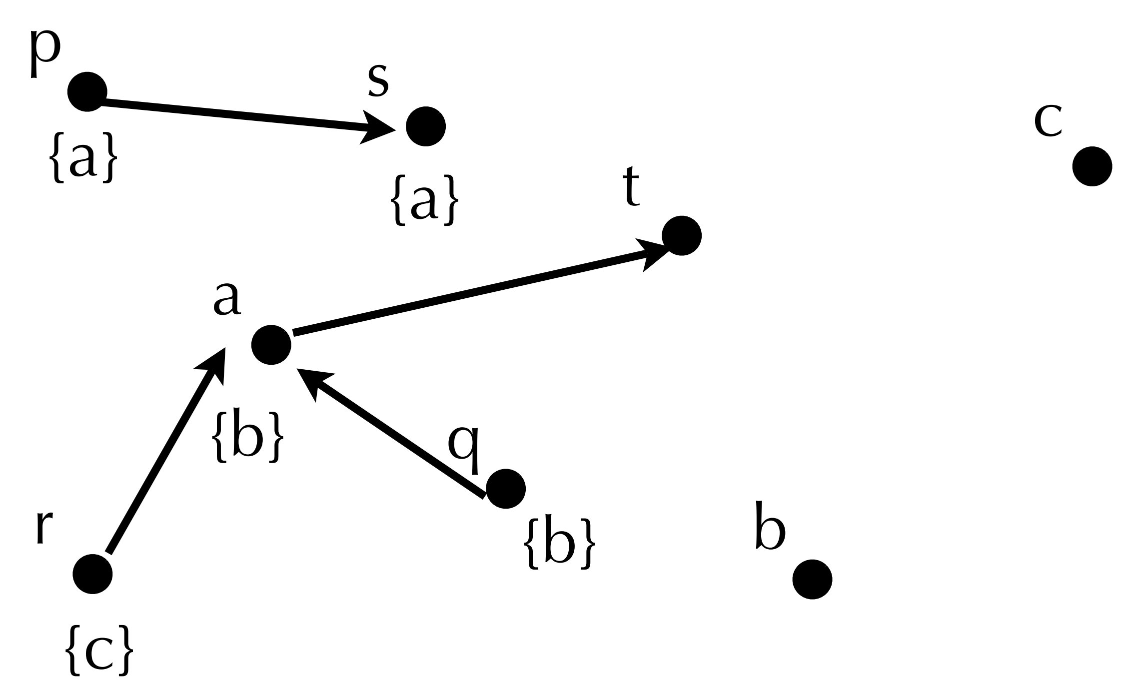
$*p \supseteq q$

$r \supseteq \{c\}$

$s \supseteq p$

$t \supseteq *p$

$*s \supseteq r$




W: p q r s a

# Same example, as graph

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

$p \supseteq \{a\}$

$q \supseteq \{b\}$

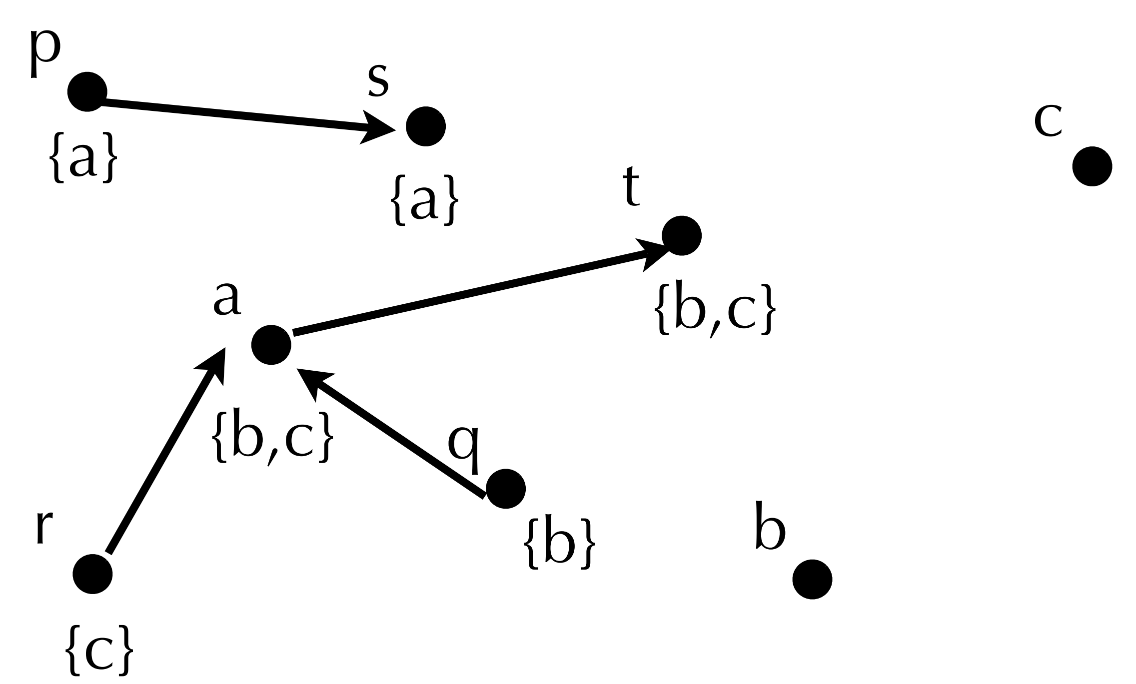
$*p \supseteq q$

$r \supseteq \{c\}$

$s \supseteq p$

$t \supseteq *p$

$*s \supseteq r$



# Cycle elimination

- Andersen-style pointer analysis is $O(n^3)$, for number of nodes in graph (Actually, quadratic in practice [Sridharan and Fink, SAS 09])
  - Improve scalability by reducing n
- Cycle elimination
  - Important optimization for Andersen-style analysis
  - Detect strongly connected components in points-to graph, collapse to single node
    - Why? All nodes in an SCC will have same points-to relation at end of analysis
  - How to detect cycles efficiently?
    - Some reduction can be done statically, some on-the-fly as new edges added
    - See *The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code*, Hardekopf and Lin, PLDI 2007

# Steensgaard-style analysis

- Also a constraint-based analysis
- Uses **equality constraints** instead of subset constraints
  - Originally phrased as a type-inference problem
- Less precise than Andersen-style, thus more scalable

| **Constraint type** | **Assignment** | **Constraint** | **Meaning** |
|---|---|---|---|
| Base | a = &b | $a \supseteq \{b\}$ | $loc(b) \in pts(a)$ |
| Simple | a = b | a = b | pts(a) = pts(b) |
| Complex | a = *b | a = *b | $\forall v \in pts(b).\ pts(a) = pts(v)$ |
| Complex | *a = b | *a = b | $\forall v \in pts(a).\ pts(v) = pts(b)$ |

# Implementing Steensgaard-style analysis

- Can be efficiently implemented using Union-Find algorithm
  - Nearly linear time: $O(n\alpha(n))$
  - Each statement needs to be processed just once

# One-level flow

- *Unification-based Pointer Analysis with Directional Assignment*, Das, PLDI 2000
- Observation: common use of pointers in C programs is to pass the address of composite objects or updateable arguments; multi-level use of pointers not as common
- Uses unification (like Steensgaard) but avoids unification of **top-level pointers** (pointers that are not themselves pointed to by other pointers)
  - i.e., Use Andersen's rules at top level, Steensgaard's elsewhere

# One-level flow

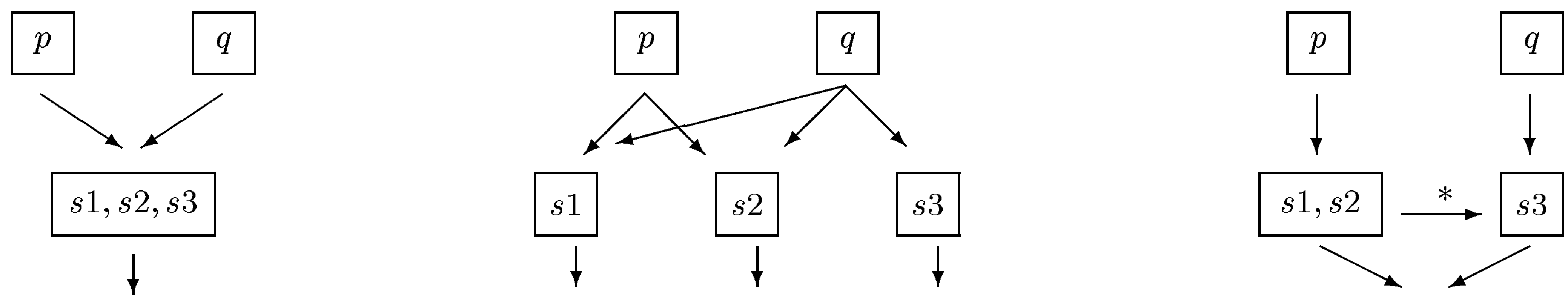
```
foo(&s1);
foo(&s2);
bar(&s3);

foo(struct s *p) { *p.a = 3; bar(p); }
bar(struct s *q) { *q.b = 4; }
```

(a)

```
p = &s1;
p = &s2;
q = &s3;
q = p;
*p.a = 3;
*q.b = 4;
```

(b)




(c) (d) (e)

Figure 1: Two programs that illustrate the difference between the algorithms of Steensgaard and Andersen. The program in (a) above represents the common case in C programs, while the program in (b) above is a variant of the program without procedure calls. Figures (c), (d) and (e) above show the points-to graphs computed by Steensgaard's algorithm, Andersen's algorithm, and our one level flow algorithm, respectively, for the program in (b) above. The edge labeled with * is a flow edge.

- Precision close to Andersen's, scalability close to Steensgaard's
    - At least, for programs where observation holds.
- Doesn't hold in Java, C++, ...

# Pointer analysis in Java

- Different languages use pointers differently
- *Scaling Java Points-To Anlaysis Using SPARK* Lhotak & Hendren CC 2003
  - Most C programs have many more occurrences of the address-of (&) operator than dynamic allocation
    - & creates stack-directed pointers; malloc creates heap-directed pointers
  - Java allows no stack-directed pointers, many more dynamic allocaiton sites than similar-sized C programs
  - Java strongly typed, limits set of objects a pointer can point to
    - Can improve precision
  - Call graph in Java depends on pointer analysis, and vice-versa (in context sensitive pointer analysis)
  - Dereference in Java only through field store and load
  - And more…
    - Larger libraries in Java, more entry points in Java, can't alias fields in Java, ...

# Object-sensitive pointer analysis

- Milanova, Rountev, and Ryder. *Parameterized object sensitivity for points-to analysis for Java*. ACM Trans. Softw. Eng. Methodol., 2005.
  - Context-sensitive interprocedural pointer analysis
  - For context, use stack of receiver objects
  - (More next week?)
- Lhotak and Hendren. *Context-sensitive points-to analysis: is it worth it?* CC 06
  - Object-sensitive pointer analysis more precise than call-stack contexts for Java
  - Likely to scale better

# Closing remarks

- Pointer analysis: important, challenging, active area
  - Many clients, including call-graph construction, live-variable analysis, constant propagation, …
  - Inclusion-based analyses (aka Andersen-style)
  - Equality-based analyses (aka Steensgaard-style)
- Requires a tradeoff between precision and efficiency
  - Ultimately an empirical question. Which clients, which code bases?
- Recent results promising
  - Scalable flow-sensitivity (see Thurs, and Hardekopf and Lin, POPL 09)
  - Context-sensitive Andersen-style analyses seem scalable (See Thurs)
- Other issues/questions (see Hind, PASTE'01)
  - How to measure/compare pointer analyses? Different clients have different needs
  - Demand-driven analyses? May be more precise/scalable…