CS 293S Pointer Analysis

Yufei Ding

Slides adapted from Wei Le, Stephen Chong

Focus of this lecture

- Terms and concepts
- Algorithms: Andersen-Style and Steensgaard-Style
- Advanced topics

What is Pointer/Alias/points-to Analysis?

- Pointer analysis statically determines:
  - the possible runtime values of a pointer
  - what storage locations a pointer can point to
  - there are certain models can represent the storage locations:

- Pointer analysis is hard, but essential for enabling many compiler optimizations.

Note: pointer analysis, alias analysis, points-to analysis often are used interchangeably

May and Must Aliasing

- May aliasing:
  - aliasing that may occur during execution (e.g., if (c) p = &i)
- Must aliasing:
  - aliasing that must occur during execution (e.g., p = &i)
- Easiest alias analysis: nothing must alias, everything may alias

Example Optimizations

- GCSE needs info on what is read/written:
  - Can p point to a or b?

```
*p = a + b;
x = a + b;
```

- Reaching definitions and constant propagation:
  - Can p point to x?

```
x = 5;
*p = 42;
y = x;
```

How Hard Is This Problem?

- Undecidable [Landi1992] [Ramalingan1994]
- Approximation algorithms, worst-case complexity, range from almost linear to doubly exponential [Hind2001]
- Two primary algorithms for point-to analysis
  - Andersen-style Analysis
  - Steensgaard-style Analysis

Andersen-Style Pointer Analysis *[Andersen1994]*

- Flow-insensitive, context-insensitive analysis
  - First for C programs, later for Java
- View pointer assignments as subset constraints:

Constraint type	Assignment	Constraint	Meaning
Base	a = &b	a ⊇ {b}	loc(b) ∈ pts(a)
Simple	a = b	a ⊇ b	pts(a) ⊇ pts(b)
Complex	a = *b	a ⊇ *b	∀v∈pts(b). pts(a) ⊇ pts(v)
Complex	*a = b	*a ⊇ b	∀v∈pts(a). pts(v) ⊇ pts(b)

Andersen-Style Pointer Analysis

- Basic idea:
  - map to subset constraints
  - construct the constraint graphs
  - compute transitive closure to propagate points-to relations along the edges of the constraint graphs

- Constraint graph:
  - one node for each variable representing its points-to set, e.g., pts(p), pts(a)
  - one directed edge for certain constraint

Andersen-Style Pointer Analysis: Constructing Constraint Graphs

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

Andersen-Style Pointer Analysis

- Initialize graph and points to sets using base and simple constraints
- Let W = { v | pts(v) ≠∅ } (all nodes with non-empty points to sets)
- 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

Andersen-style analysis: Algorithm Analysis

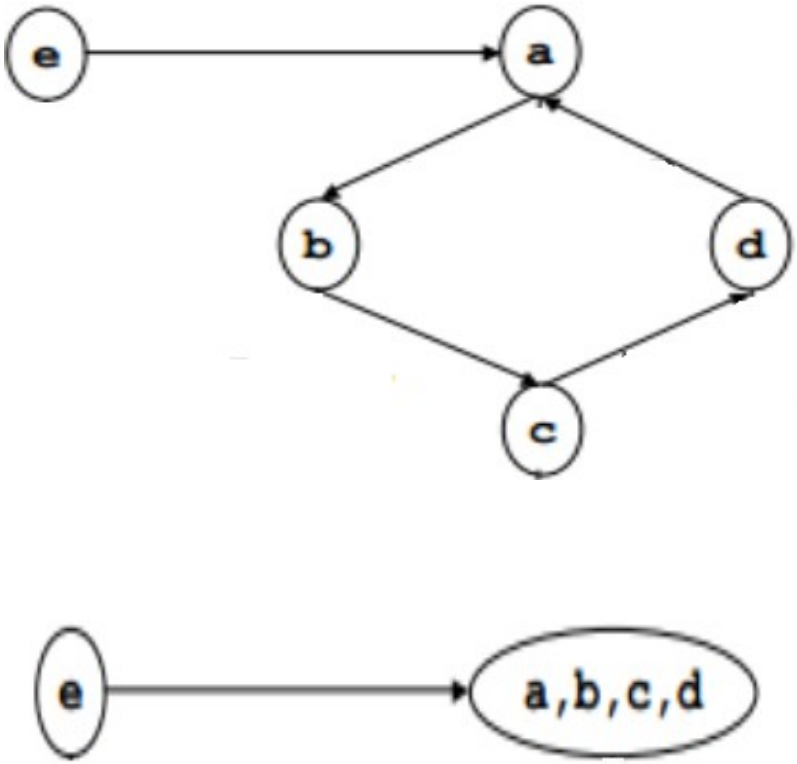
- Can be reduced to computing the transitive closure of a dynamic graph
  - dynamic graph: the graph changes over the analysis of the program
  - the transitive closure of a directed acyclic graph (DAG) is the reachability relation of the DAG. (graph: a set of nodes, and binary relations among the nodes)

- A well-studied problem for which the best known complexity is O(n3) (n is the number of node)

Andersen-Style Pointer Analysis: Cycle Elimination

- Impart optimization for Anderson-style analysis
- Detect strongly connected components in points-to graph, collapse to a singe node
  - Why? All nodes in an SCC will have the same points-to relation at the end of analysis
- How to detect cycles efficiently?
  - Some reduction can be done statically, some on-the-fly as new edges added
  - See Fast and Accurate Pointer Analysis for Millions of Lines of Code, Hardekopf and Lin, PLDI 2007.

Andersen-Style Pointer Analysis: Cycle Elimination



Steensgaard-Style Pointer Analysis *[Steensgaard1996POPL]*

- Points-to Analysis in almost linear time
  - Uses equality constraints instead of subset constraints
  - Unification based approach: assignment unifies the graph nodes, e.g., x = y (unified x and y in the same node), also called union-find algorithm, exclusion-based approaches, nearly linear complexity
- O(n · α(n)), where α(n) is the inverse Ackermann's function, α(2132) < 4
  - Scalable
  - Less precise than Andersen-style, thus more

Steensgaard-Style Pointer Analysis

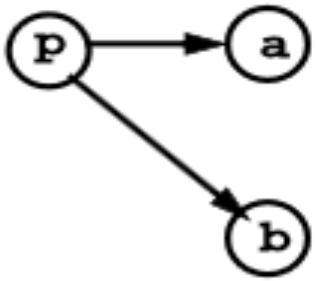
- Key idea: maintain a set of disjoint sets and supports two operations:
  - FIND(x): return the set containing x
  - UNION(x, y): union the two sets containing x and y

Steensgaard-Style Pointer Analysis *[Steensgaard1996POPL]*

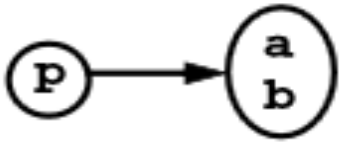
```
merge(x, y)
{
    x = FIND(x); y = FIND(y);
    if (x == y) then return;
    UNION(x,y);
    merge(points-to(x),points-to(y));
}

for each constraint LHS = RHS
    merge(LHS,RHS)
```

Andersen vs. Steensgaard Style Pointer Analysis

That is, in Andersen's Algorithm we might have



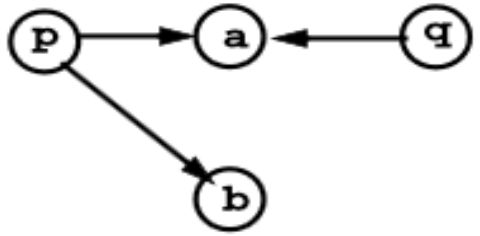
In Steensgaard's Algorithm we would instead have



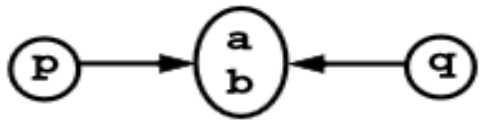
In effect any two locations that might be pointed to by the same pointer are placed in a single equivalence class.

Andersen vs. Steensgaard Style Pointer Analysis

Steensgaard's Algorithm is sometimes less accurate than Andersen's Algorithm. For example, the following points-to graph, created by Andersen's Algorithm, shows that `p` may point to `a` or `b` whereas `q` may only point to `a`:



In Steensgaard's Algorithm we get



incorrectly showing that if `p` may point to `a` or `b` then so may `q`.

Andersen vs. Steensgaard Style Pointer Analysis

- Horwitz and Shapiro examined 61 C programs, ranging in size from 300 to 24,300 lines.
- As expected, Steensgaard is less precise: On average points-to sets are 4 times bigger; at worst 15 times bigger.
- As expected, Andersen is slower. On average 1.5 times slower: at worst 31 times slower.
- Both are much better than the naive "address taken" approach.
- Bottom line: Use Andersen for small programs, use Steensgaard (or something else) for large programs.

Andersen vs. Steensgaard Style Pointer Analysis

Name	Size (LoC)	Andersen(sec)	Steensgaard(sec)
triangle	1986	2.9	0.8
gzip	4584	1.7	1.1
li	6054	738.5	4.7
bc	6745	5.5	1.6
less	12152	1.9	1.5
make	15564	260.8	6.1
tar	18585	23.2	3.6
espresso	22050	1373.6	10.2
screen	24300	514.5	10.1

75MHz SuperSPARC, 256MB RAM

[Shapiro-Horwitz POPL'97]

Points-to Analyses Work in Real Data Flow Problems?

In "Which Pointer Analysis Should I Use," Hind and Pioli survey the effectiveness of a number of points-to analyses in actual data flow analyses (mod/ref, liveness, reaching defs, interprocedural constant propagation).

Their conclusions are essentially the same across all these analyses:

- Steensgaard's analysis is significantly more precise than address-taken analysis and not significantly slower.
- Andersen's analysis produces modest, but consistent, improvements over Steensgaard's analysis.
- Both context-sensitive points-to analysis and flow-sensitive points-to analysis give little improvement over

Summary: Andersen vs. Steensgaard

- Both are flow-insensitive and context-insensitive
  - Control flow information is not used, the order of statements is not considered

- Differ in points-to set construction
  - Andersen-style: many out edges, one variable per node
  - Steensgaard-style: one out edge, many variables per node

- Andersen-style: inclusion-based, subset-based
  - the slowest but most precise flow-insensitive algorithm
- Steensgaard-style: equality-based, unification-based
  - the fastest but least precise

Advanced point-to analysis

Horwitz and Shapiro suggest each node in the points-to graph be limited to out degree k, where $1 \leq k \leq n$.

If k =1 then they have Steensgaard's approach.

If k =n (n is number of nodes in points to graph), then they have Andersen's approach.

Their worst case run-time is

$O(k^2 n)$, which is not much worse than Steensgaard if k is kept reasonably small.

The Horwitz-Shapiro Approach: 1997 POPL – Fast and Accurate Flow-Insensitive Points-To Analysis

Advanced point-to analysis

To use their approach assign each variable that may be pointed to to one of k categories.

Now if `p` may point to `x` and `p` may also point to `y`, we merge `x` and `y` only if they both are in the same category.

If `x` and `y` are in different categories, they aren't merged, leading to more accurate points-to estimates.

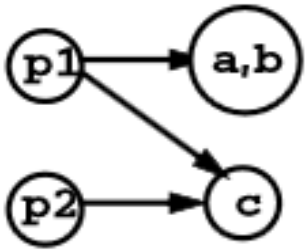
Advanced point-to analysis

Example

```
p1 = &a;
p1 = &b;
p1 = &c;
p2 = &c;
```

Say we have k = 2 and place `a` and `b` in category 1 and `c` in category 2.

We then build:



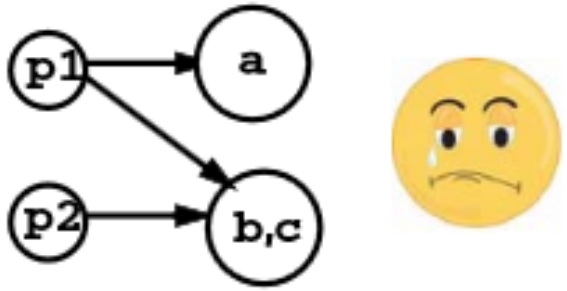
This points-to graph is just as accurate as that built by Andersen's approach.

Advanced point-to analysis

But...

What if we chose to place `a` in category 1 and `b` and `c` in category 2.

We now have:



This graph is inexact, since it tells us `p2` may point to `b`, which is false.

(Steensgaard would have been worse still, incorrectly telling us `p2` may point to `a` as well as `b` and `c`).

Advanced point-to analysis

What if we ran Shapiro and Horwitz's points-to analysis twice, each with different category assignments?

Each run may produce a different points-to graph. One may say `p2` points to `b` whereas the other says it can't.

Which do we believe?

Neither analysis misses a genuine points-to relation. Rather, merging of nodes sometimes creates false points-to information.

So we will believe `p2` may point to `b` only if all runs say so.

This means multiple runs may "filter out" many of the false points-to relations caused by merging.

Advanced point-to analysis

How Many Runs are Needed?

How are Categories to be Set?

We want to assign categories so that during at least one run, any pair of pointed-to variables are in different categories.

This guarantees that if all the runs tell us `p` may point to `a` and `b`, it is not just because `a` and `b` always happened to be assigned the same category.

To force different category assignments for each pair of variables, we assign each pointed-to variable an index and write that index in base k (the number of categories chosen).

Advanced point-to analysis

For example, if we had variables `a`, `b`, `c` and `d`, and chose k = 2, we'd use the following binary indices:

`a`	00
`b`	01
`c`	10
`d`	11

Note that the number of base k digits needed to represent indices from 0 to n-1 is just ceiling($\log_k$ n).

This number is just the number of runs we need!

Advanced point-to analysis

Why?

In the first run, we'll use the right most digit in a variable's index as its category.

In the next run, we'll use the second digit from the right, then the third digit from the right, ...

Any two distinct variables have different index values, so they must differ in at least digit position.

Advanced point-to analysis

Returning to our example,

`a` 00

`b` 01

`c` 10

`d` 11

On run #1 we give `a` and `c` category 0 and `b` and `d` category 1.

On run #2, `a` and `b` get category 0 and `c` and `d` get category 1.

So using just 2 runs in this simple case, we eliminate much of the inaccuracy Steensgaard's merging introduces.

Run time is now $O(\log_k(n)\ k^2\ n)$.

Advanced point-to analysis

How Well does this Approach Work?

On 25 tests, using 3 categories, Horwitz & Shapiro points-to sets on average are 2.67 larger than those of Andersen (Steensgaard's are 4.75 larger).

This approach is slower than Steensgaard but on larger programs it is 7 to 25 times faster than Andersen.