

CS 701 FINAL EXAM (REMINDER)

**Friday, December 12, 4:00–6:00 P.M.,
1289 Computer Science.**

ANDERSEN'S ALGORITHM

An algorithm to build a points-to graph for a C program is presented in: "Program Analysis and Specialization for the C programming Language," L.O. Andersen, 1994.

The algorithm examines statements that create pointers, one by one, in textual order (the algorithm is flow-insensitive). Each statement updates the points-to graph if it can create new points-to relationships.

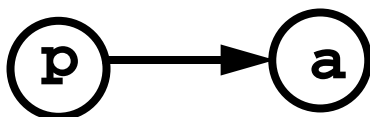
Six kinds of statements are considered:

- $p = \&a;$
- $p = q;$
- $p = *r;$
- $*p = \&a;$
- $*p = q;$
- $*p = *r;$

We will detail the points-to graph updates each of the statements induces.

1. $p = \&a;$

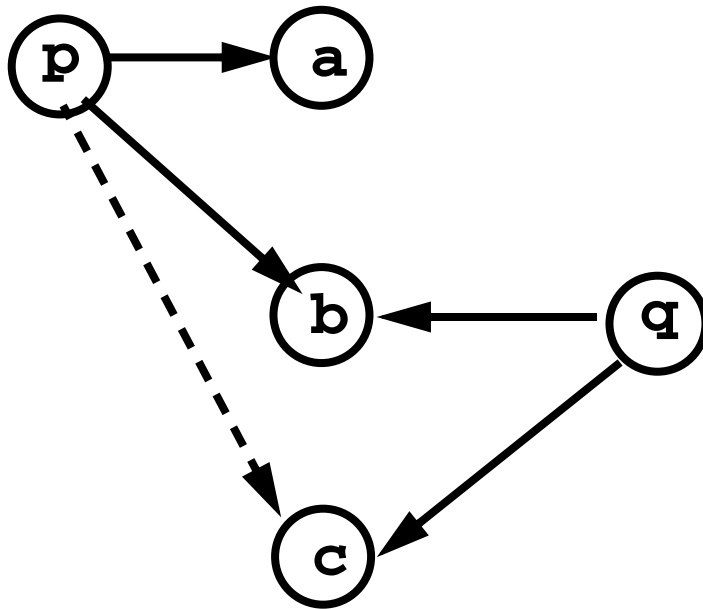
We add an arc from p to a , showing p can possibly point to a :



2. $p = q;$

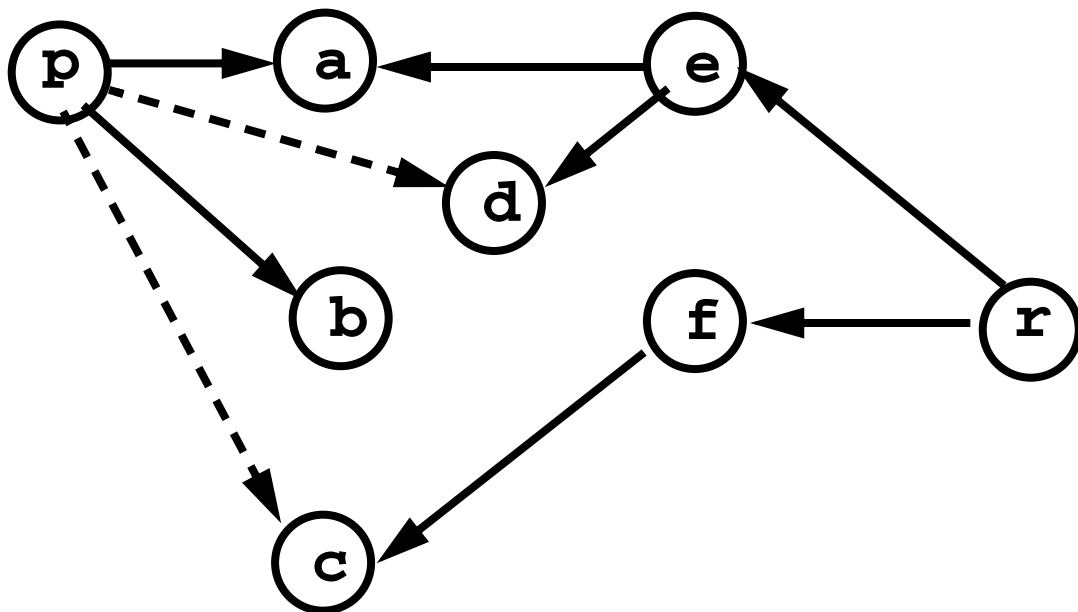
We add arcs from p to everything q points to. If new arcs from q are later added, corresponding arcs from p must also be added (this implies an iterative or worklist algorithm).

For example (the dashed arc is newly added):



3. $p = *r;$

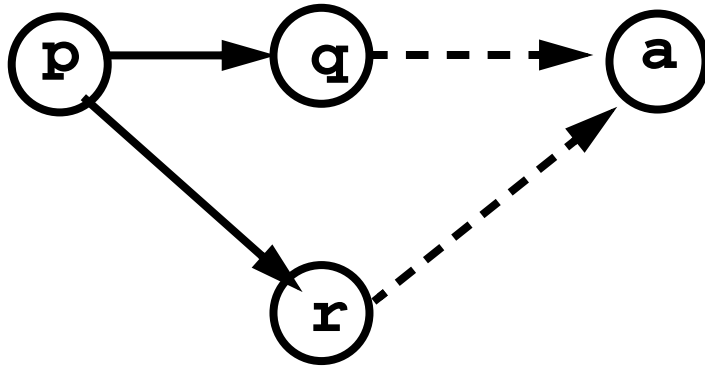
Let S be all the nodes r points to. Let T be all the nodes members of S point to. We add arcs from p to all nodes in T . If later pointer assignments increase S or T , new arcs from p must also be added (this again implies an iterative or worklist algorithm). For example (dashed arcs are newly added):



4. `*p = &a;`

Add an arc to `a` from all nodes `p` points to. If new arcs from `p` are later added, new arcs to `a` must be added (this implies an iterative or worklist algorithm).

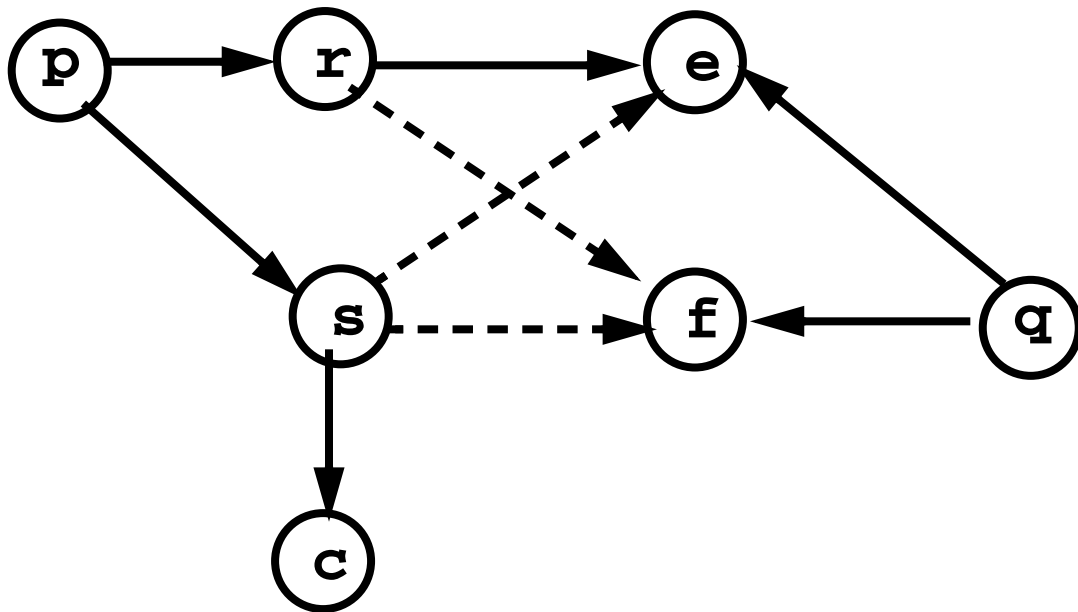
For example (dashed arcs are newly added):



5. $*p = q;$

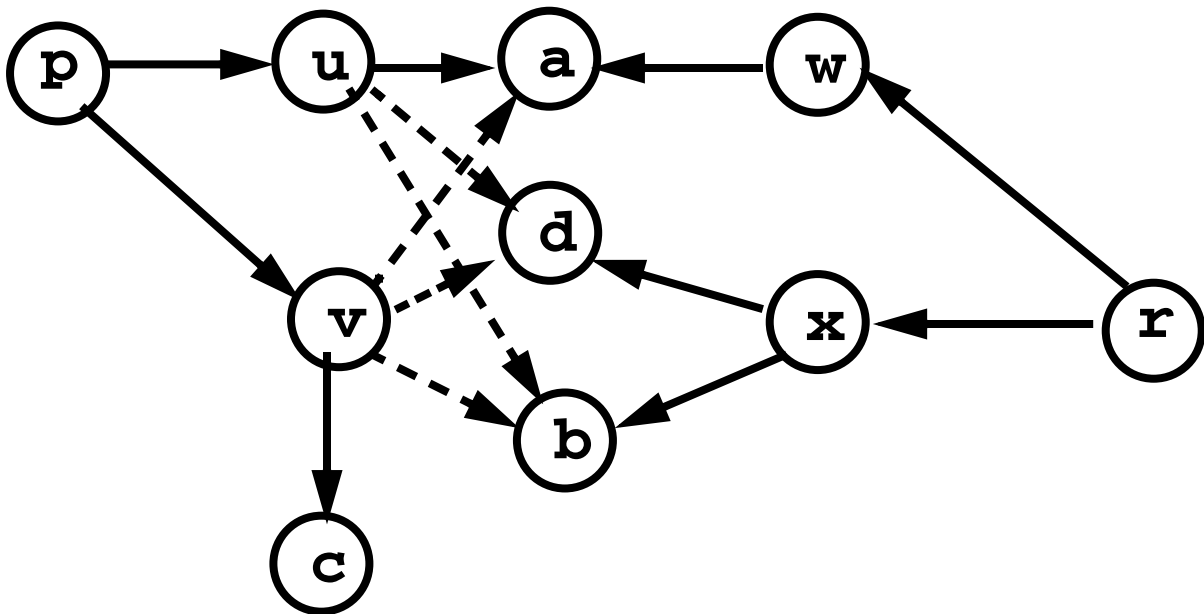
Nodes pointed to by p must be linked to all nodes pointed to by q . If later pointer assignments add arcs from p or q , this assignment must be revisited (this again implies an iterative or worklist algorithm).

For example (dashed arcs are newly added):



6. $*p = *r;$

Let S be all the nodes r points to.
Let T be all the nodes members of S point to.
We add arcs from all nodes p points to to all nodes in T .
If later pointer assignments increase S or T or link new nodes to p , this assignment must be revisited (this again implies an iterative or worklist algorithm).
For example (dashed arcs are newly added):



Example

Consider the following pointer manipulations:

```
p1 = &a;
```

```
p2 = &b;
```

```
p1 = p2;
```

```
r = &p1;
```

```
*r = &c;
```

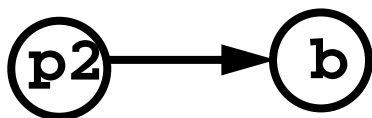
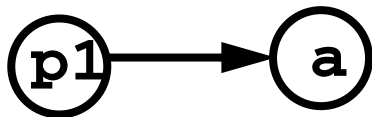
```
p3 = *r;
```

```
p2 = &d;
```

We start with:

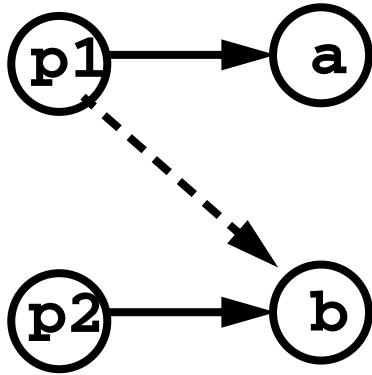
```
p1 = &a;
```

```
p2 = &b;
```



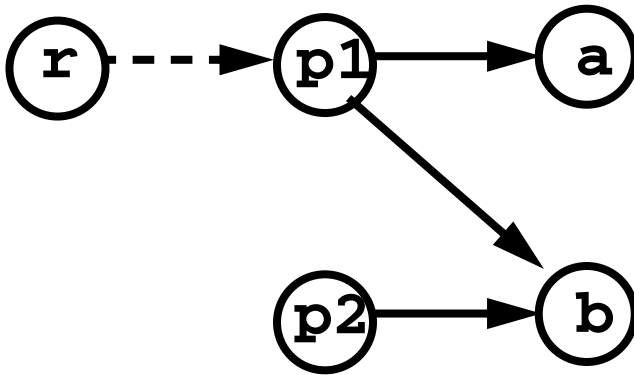
Next:

```
p1 = p2;
```



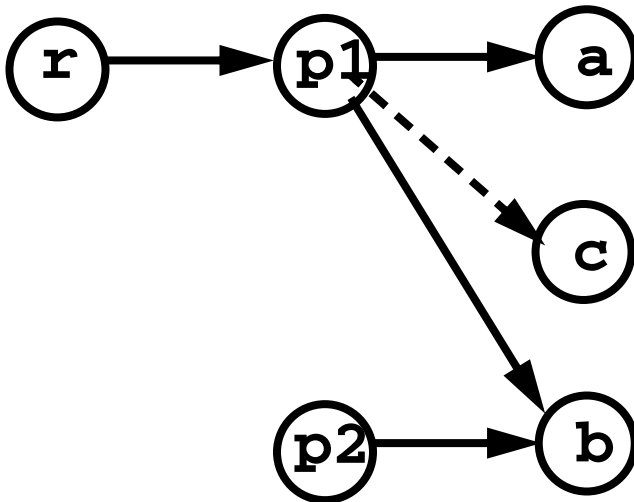
Then:

```
r = &p1;
```



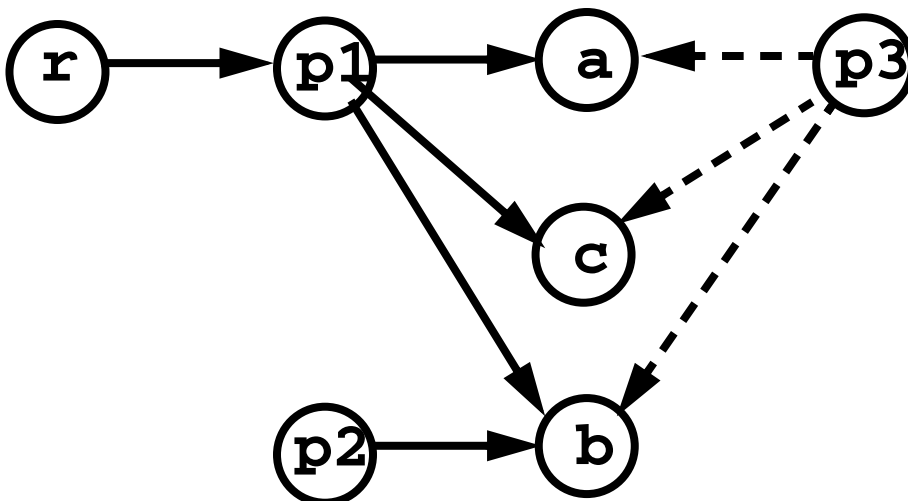
Next:

`*r = &c;`



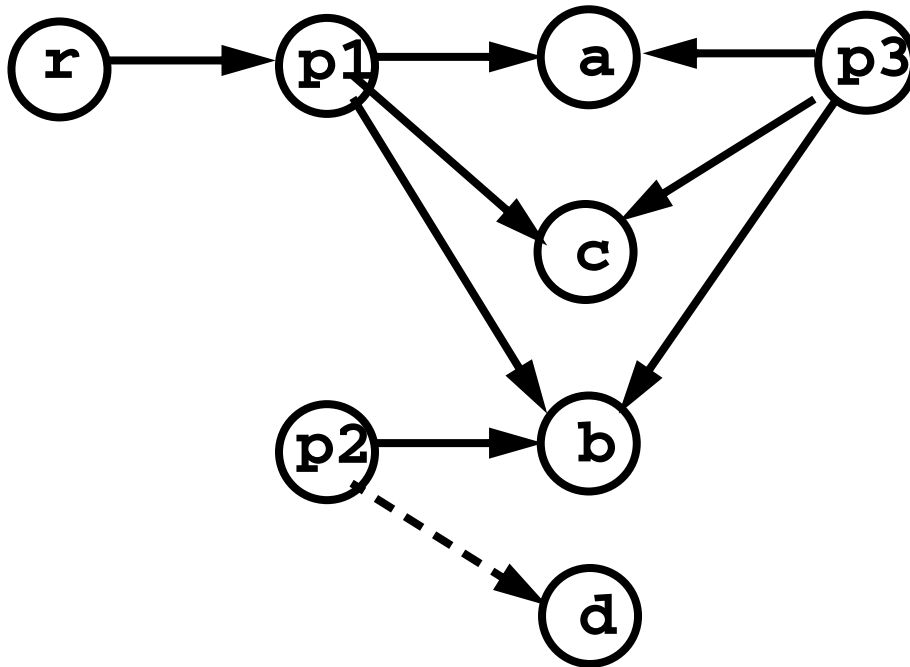
Then:

`p3 = *r;`



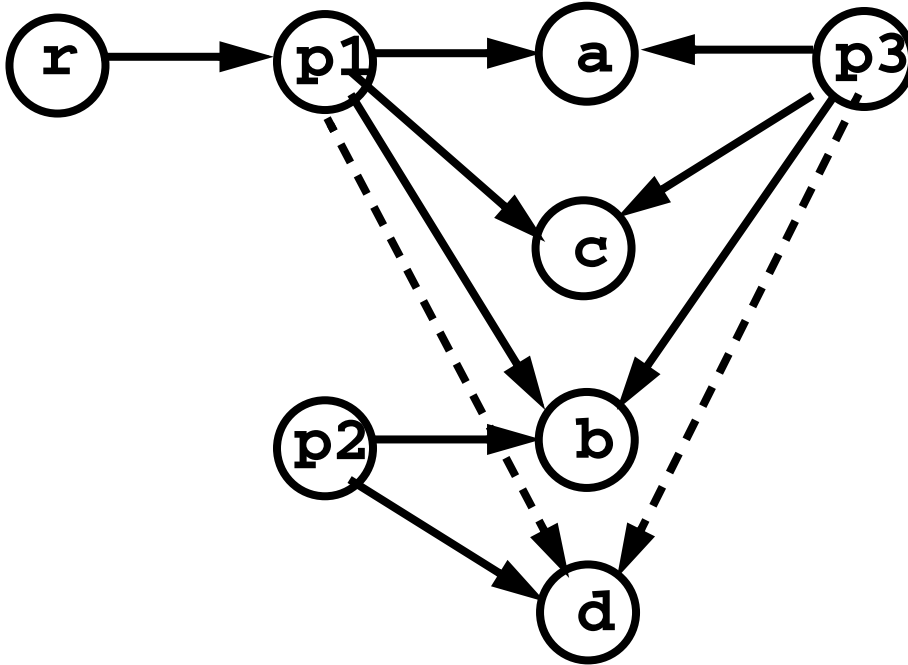
Finally:

`p2 = &d;`



But we aren't quite done yet. This algorithm is **flow-insensitive**, so we must consider other execution orders (and iterative re-execution). If we make another pass through the assignments, we see that the final

assignment to **p2** can flow to **p1**, and then to **p3** through **r**:



This points-to graph is rather dense, but it does capture all the ways pointer values might propagate through the various pointer assignments.

Calls are handled by treating pointer parameters and pointer returns as assignments, done at the points of call and return. Subprogram bodies are effectively inlined to capture the points-to relations they induce.

Given

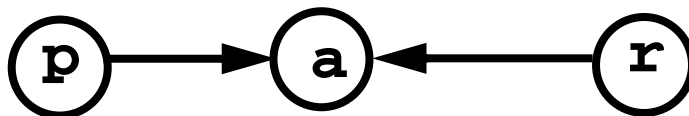
```
*int echo (*int r) {  
    return r; }  
p = echo (&a);
```

we see the implicit assignments

```
r = &a;
```

```
p = r;
```

and add the following points-to information:



As an optimization, libraries can be pre-analyzed to determine the points-to relations they induce. Most may use (read) pointers but don't create any new points-to relations visible outside their bodies. Call to such library routines can be ignored as far as the caller's points-to graph is concerned.

PERFORMANCE OF ANDERSEN'S ALGORITHM

Experience has shown that Andersen's Algorithm gives useful points-to data and is far superior to the naive address-taken approach.

Interestingly, experiments show that making the technique flow-sensitive or calling context-sensitive doesn't improve results very much on typical benchmarks.

But execution time for moderate to large programs can be a problem.

Careful analysis shows that Andersen's Algorithm can require $O(n^3)$ time (where n is the number of nodes in the points-to graph).

The reason for this larger-than-expected analysis time is that a statement like

`p = *q;`

can force the algorithm to visit n^2 nodes (q may point to n nodes and each of these nodes may point to n nodes). The number of pointer statements analyzed can be $O(n)$, leading to an $O(n^3)$ execution time.

STEENSGAARD'S ALGORITHM

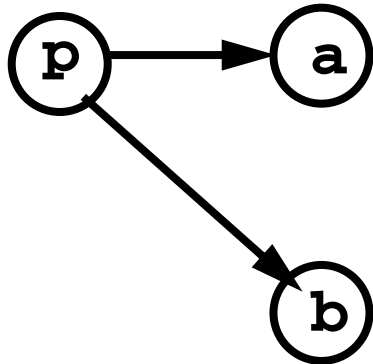
It would be useful to have a reasonably accurate points-to analysis that runs in essentially linear time so that **really** large programs could be handled.

This is what Steensgaard's Algorithm offers.

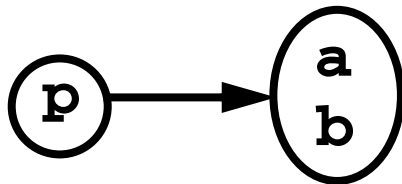
(Points-to Analysis in Almost Linear Time, B. Steensgaard, 1996 Principles of Programming Languages Conference.)

Steensgaard's Algorithm is essentially Andersen's Algorithm, simplified by **merging** nodes **a** and **b** if any pointer can reference both.

That is, in Andersen's Algorithm we might have

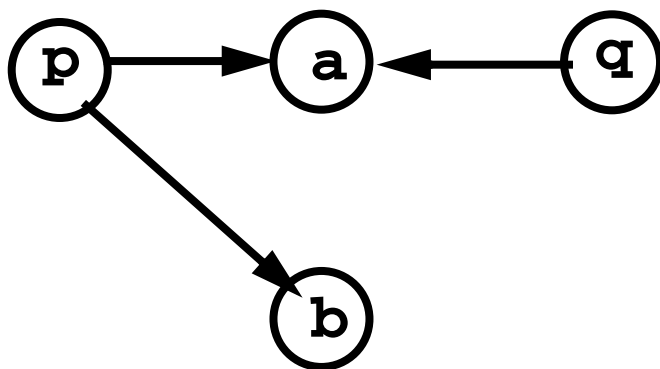


In Steensgaard's Algorithm we would instead have

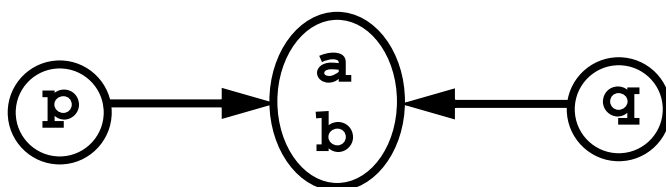


In effect any two locations that might be pointed to by the same pointer are placed in a single equivalence class.

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 .

But now statements like

`p = *q;`

can't force the algorithm to visit n^2 nodes, because multiple nodes referenced by the same pointer are always merged. Using the fast union-find algorithm, we can get an execution time of $O(n \alpha(n))$ which is essentially linear in n . Now very large programs can be analyzed, and without too much of a loss in precision.

ANDERSEN VS. STEENSGAARD IN PRACTICE

- 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.

THE HORWITZ-SHAPIRO APPROACH

It would be nice to have a points-to analysis that is parameterizable, ranging between the accuracy of Andersen and the speed of Steensgaard.

Horwitz and Shapiro (Fast and Accurate Flow-Insensitive Points-To Analysis, 1997 Principles of Programming Languages Conference) present a technique intermediate to those proposed by Andersen and Steensgaard.

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.

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

Now if \mathbf{p} may point to \mathbf{x} and \mathbf{p} may also point to \mathbf{y} , we merge \mathbf{x} and \mathbf{y} **only if** they both are in the same category.

If \mathbf{x} and \mathbf{y} are in different categories, they **aren't** merged, leading to more accurate points-to estimates.

Example

`p1 = &a;`

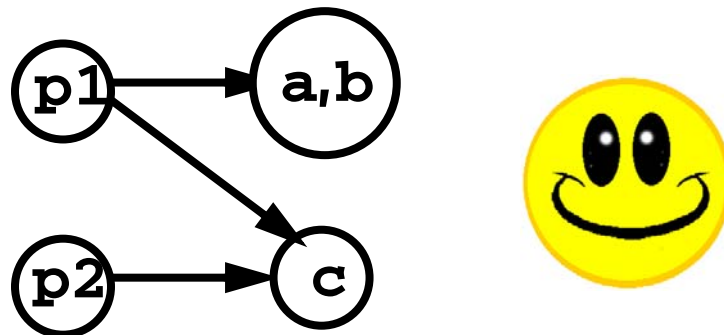
`p1 = &b;`

`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:

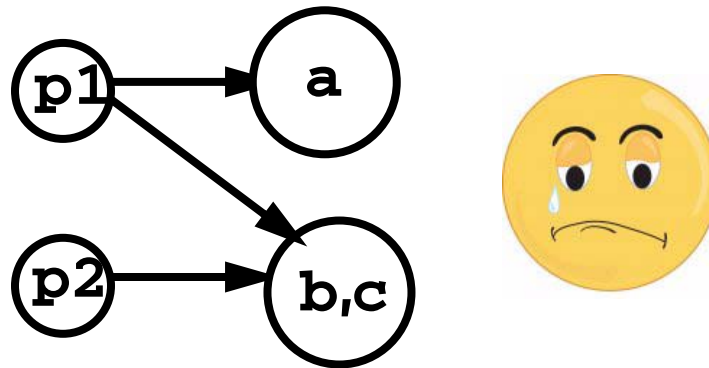


This points-to graph is just as accurate as that built by Andersen's approach.

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**).

ANOTHER Good Idea

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 p_2 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 p_2 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.

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).

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 $\text{ceiling}(\log_k n)$.

This number is just the number of runs we need!

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.

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 $O(\log_k(n) k^2 n)$.

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.

How Well do 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 Andersen's analysis.