Outline

Introduction and Rationale

Introduction

- Last time we saw flow sensitive points-to analysis
- Computes information at every point of a program
  - Precise
  - The information is a (large) graph — expensive!

Flow-Insensitive analysis

- Compute just one graph for the entire program
- Consider all statements regardless of control-flow
- SSA or similar forms can recover some precision

Introduction and Rationale

Introduction

- Last time we saw flow sensitive points-to analysis
- Computes information at every point of a program
  - Precise
  - The information is a (large) graph — expensive!

Flow-Insensitive analysis

- Compute just one graph for the entire program
- Consider all statements regardless of control-flow
- SSA or similar forms can recover some precision

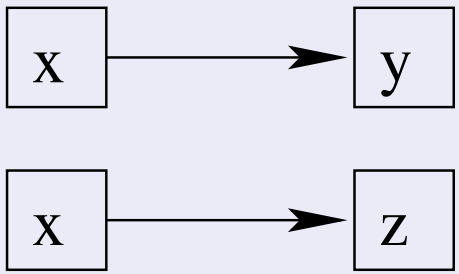
A little comparison

Code

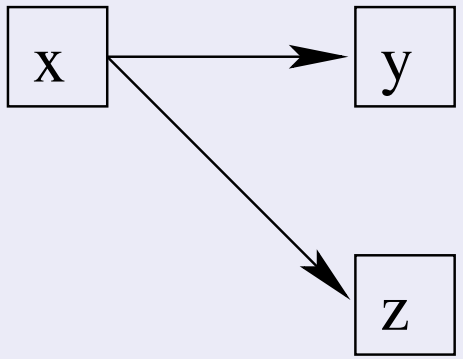
```
int x;
int *y, *z;
x = &y;

x = &z;
```

Flow-Sensitive



Flow-Insensitive



Outline

Andersen's algorithm

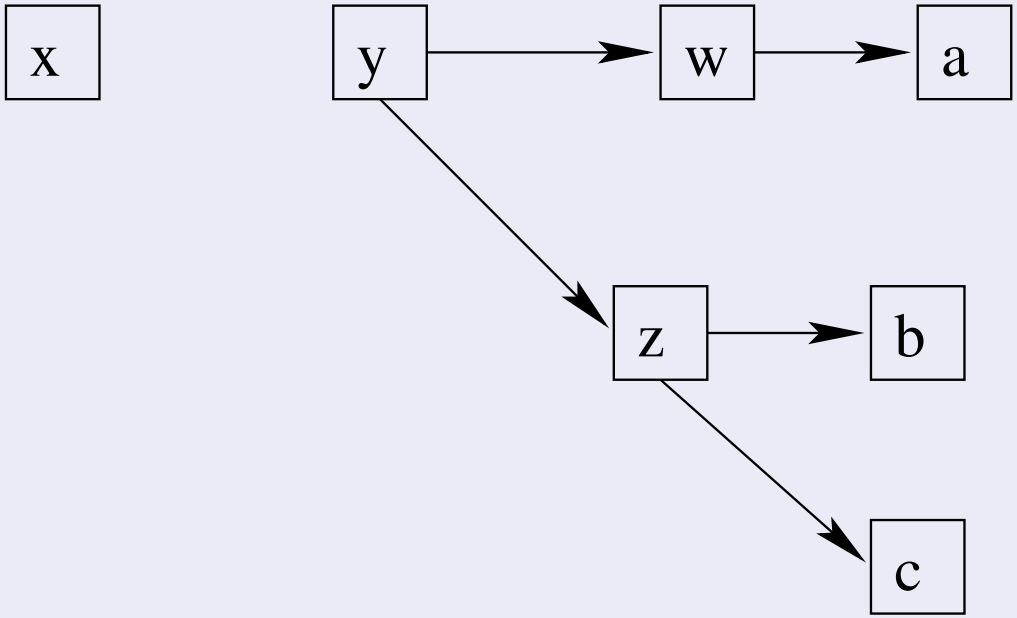
- Essentially the immediate adaptation of the usual dataflow points-to algorithm to be flow-insensitive
- Since do not know the order of statements, can say less:
  - `x = &y` — can only know that $y \in pt(x)$
  - `x = y` — can only know that $pt(y) \subseteq pt(x)$
- When analyzing, collect such constraints
- Can use a fixed-point computation to compute the actual points-to sets

Constraints for C

1. `x = &y` — $y \in pt(x)$
2. `x = y` — $pt(y) \subseteq pt(x)$

Constraints for C II

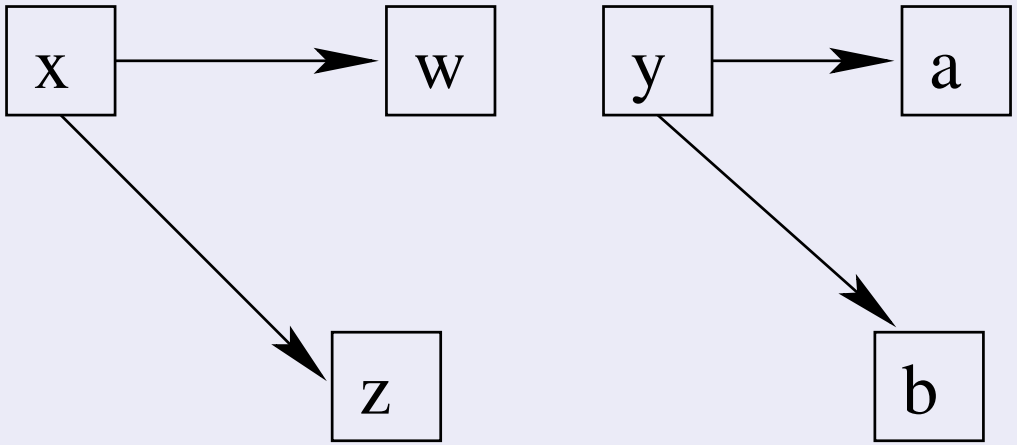
```
x = *y
```



$\forall a \in pt(y).pt(a) \subseteq pt(x)$

Constraints for C III

```
*x = y
```



$\forall w \in pt(x).pt(y) \subseteq pt(w)$

Constraints for C — Summary

1. `x = &y` — $y \in pt(x)$
2. `x = y` — $pt(y) \subseteq pt(x)$
3. `x = *y` — $\forall a \in pt(y).pt(a) \subseteq pt(x)$
4. `*x = y` — $\forall w \in pt(x).pt(y) \subseteq pt(w)$

Constraints for Java

1. Stack variables can not be pointed to, only heap objects can be
2. Can take advantage of type safety
3. The following is one memory abstraction:
   - Name objects by allocation site
   - Variables point to objects
   - Fields of objects point to objects

Constraints for Java II

1. `x = y` — $pt(y) \subseteq pt(x)$
2. `y.f = x` — $\forall o \in pt(y)\,(pt(x) \subseteq pt(o.f))$
3. `x = y.f` — $\forall o \in pt(y)\,(\texttt{o.f} \subseteq pt(x))$

Cost of the algorithm

Asymptotically

- Implicitly have a constraint graph, $O(n)$ nodes, $O(n^2)$ edges
- The fixed point computation essentially computes transitive closure — which is an $O(n^3)$ computation

In practice

- Usually, nowhere near that bad...
- ... but can be bad enough to be unusable

Cost of the algorithm

Asymptotically

- Implicitly have a constraint graph, $O(n)$ nodes, $O(n^2)$ edges
- The fixed point computation essentially computes transitive closure — which is an $O(n^3)$ computation

In practice

- Usually, nowhere near that bad...
- ... but can be bad enough to be unusable

Actual Performance (from [ShHo97])

Name	Size (LoC)	Time (sec)
triangle	1986	2.9
gzip	4584	1.7
li	6054	738.5
bc	6745	5.5
less	12152	1.9
make	15564	260.8
tar	18585	23.2
espresso	22050	1373.6
screen	24300	514.5

75MHz SuperSPARC, 256MB RAM

Reducing the cost

- Cycles in a graph must have the same points-to sets, so can be collapsed to a single node [FäFoSuAi98]
  - In some cases runs at much as 50x faster
  - `li` is done in 30.25 seconds, `espresso` in 27 seconds, on UltraSparc in 167-400Mhz
- If two variables have the same points-to sets, they can be collapsed [RoCh00]
  - Around 2x improvement in run time, 3x lower memory usage
- BDDs (Reduced Ordered Binary Decision Diagrams) have been used to represent the graph more sparsely [BeLhQiHeUm03]

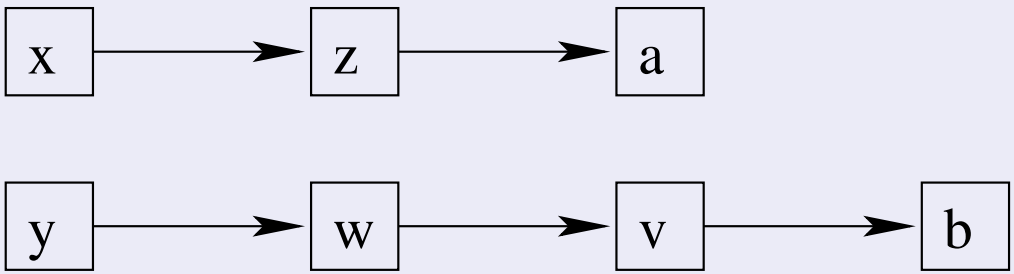
Outline

Overview

- Can view the problem as trying to assign synthetic types to each reference — so it points to objects of specified type
- A type is defined recursively as pointing to an another type
- Hence, proceeds as a type inference algorithm, doing unification
- `x = y` — $\tau(x) = \tau(y)$, so take $pt(x) = pt(y)$
- Each type points to one other type, so the points-to graph has at most 1 out edge for each node (but each node can be many variables)
  - Graph is of linear size — fast!
  - Limits precision

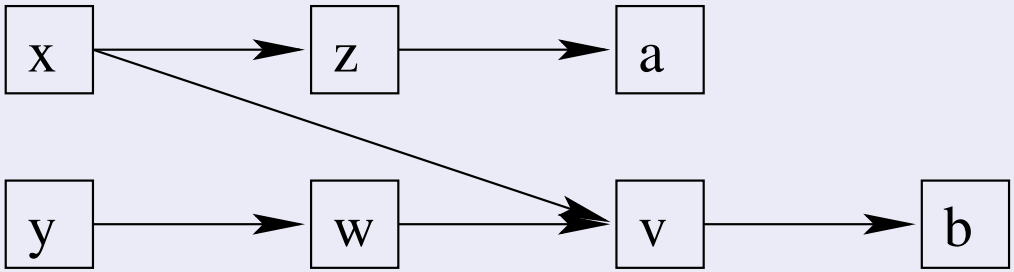
Processing of assignments

`x=*y`



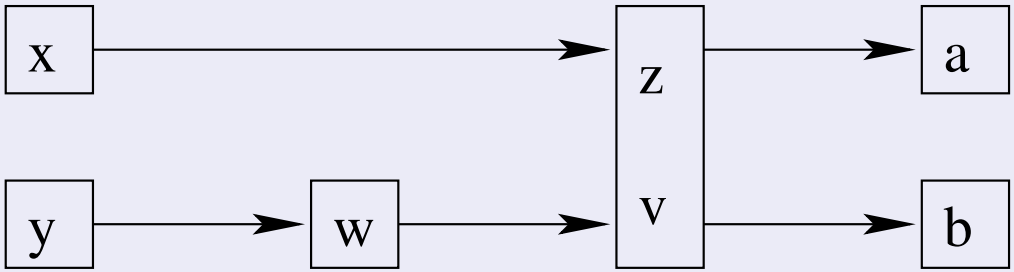
Processing of assignments

`x=*y`



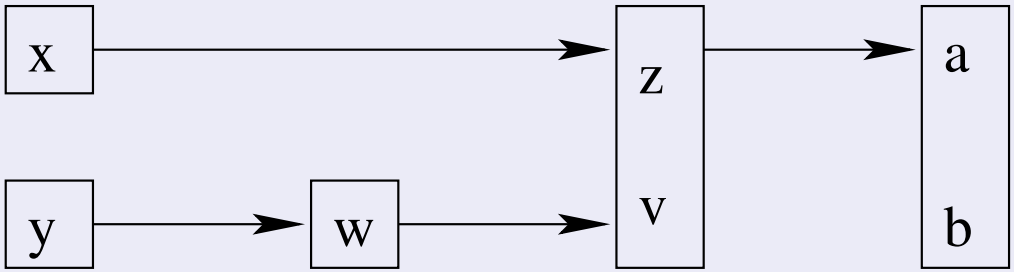
Processing of assignments

`x=*y`



Processing of assignments

x=*y



Making it work

There are a couple of problems that arise in practice

- Building a call graph
  - Make the type a pair, including a function pointer portion
  - Compute the set of functions that may point to using unification as well
- Integer assignments to pointers/lack of type safety
  - int* a = 0, *x = a, *y = b;
  - Will collapse them into a single node
  - Should only do unification if RHS is known to be a pointer
    - Don't unify if we don't see the RHS pointing to anything, just record an edge
    - Perform a unification if RHS gets to point to something

Complexity

It's fast!

- Asymptotically, $O(N\alpha(N, N))$
- Has been shown to analyze programs with millions of lines of code in under a minute

Outline

Some numbers — time (from [ShHo97])

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

Some numbers — Average alias set size (from [ShHo97])

Name	Size (LoC)	Andersen	Steensgaard
triangle	1986	4.01	21.93
gzip	4584	2.96	25.17
li	6054	171.14	457.89
bc	6745	18.57	83.55
less	12152	7.11	63.75
make	15564	74.70	414.03
tar	18585	17.41	53.7
espresso	22050	109.53	143.4
screen	24300	106.89	652.8

Relation between algorithms

- Andersen's algorithm can be viewed as type-inference, too
  - But with subtyping
  - `x = y`: $\tau(y) <: \tau(x)$, so $pt(y) \subseteq pt(x)$.
- Steensgaard's algorithm can be thought as restricting the out-degree of the graph procuded by Andersen's algorithm to 1, by merging nodes when that is exceeded

Outline

k-limiting

[ShHo97] provides a k-limiting algorithm, which with $k = 1$ behave as Steensgaard, with $k = N$ as Andersen

- Assign variables k colors
- Have a separate points-to slot for each color
- Do a few runs with different assignments, and intersect the results ($k^2 \log_k N$ factor slowdown)
- Average alias set size was shrunk by about 1.78,
- About 2x faster than Andersen when that runs slowly, but often slower than it — very high constant factors

One level flow

[Das2000] introduced an another heuristic.

Algorithm

- Observation: C programs mostly use pointers to pass in parameters, which are basically assignments
- Solution: Accurately model the simple cases by using containment constraints to refer to points-to sets of symbols in the assignment, but unify stuff further out
- Can get some context sensitivity on top of it, by labeling edges, and doing CFL reachability (makes it $O(n^3)$)

Accuracy

- Produces nearly identical sets as Andersen for most test programs (except one that used pointers to pointers)

Performance

- Asymptotically: linear memory use, quadratic time (in the constraint-solving phase)
- About 2x slower than Steensgaard's algorithm in practice
- Analyzes 1.4 million lines of code (Word97) in about 2 minutes on a 450Mhz Xeon

Discussion...