Pointer Analysis

B. Steensgaard: **Points-to Analysis in Almost Linear Time.** POPL 1996

M. Hind: **Pointer analysis: haven't we solved this problem yet?** PASTE 2001

Presented by Ronnie Barequet

23.03.14

Outline

- Introduction
  - What is “Pointer Analysis”
  - Motivation
  - Initial approach
  - Dimension of problem
- Steensgaard’s Algorithm
  - Algorithm and its complexity
  - Pros and Cons
- Alternatives and additional features:
  - Andersen’s algorithm
  - More dimensions
  - Measures of precision
  - Hybrid algorithms
- Conclusions
- Discussion

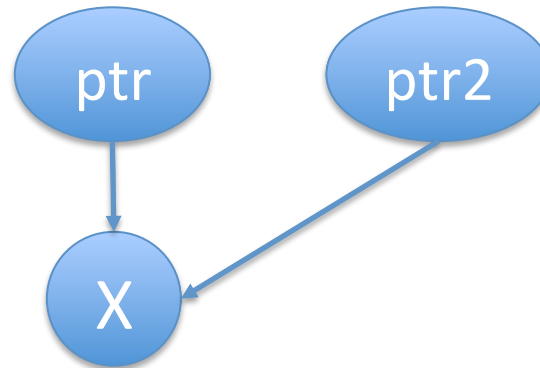
Introduction: Pointer Analysis

- At which locations does a pointer refer to?
- Example:

X = 5

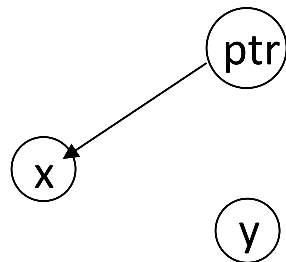
Ptr = &x

Ptr2 = ptr



Points-to graph

- Nodes are program variables
- Edge (a,b): variable ‘a’ points-to variable ‘b‘



- Out-degree of node may be more than one
  - May point to analysis
  - Depending on path taken

Motivation

- Compute accesses of portions of code
- Liveness
  - Garbage collection
  - Complier Optimizations
  - Further static analysis improvement in efficiency and accuracy
  - Improve scalability of pointer analysis! [Khedker-Mycroft-Rawat 12]
- Optimization of runtime
  - Locality of pages
  - parallelization
  - profiling

Motivation (cont.)

- Data Flow:
  - Call graph construction
  - Basis for increasingly sensitive analysis
- Alias Analysis
  - when are two expression equal?
  - Want to be able to know this also when code uses multiple dereferences.
- Typestates
  - Constant propagation
  - Use before define [Yahav et al 06]
- Formal verification

Dimensions

- Intraprocedural / interprocedural
- Flow-sensitive / flow-insensitive
- Context-sensitive / context-insensitive
- May versus must
- …

Flow Sensitivity

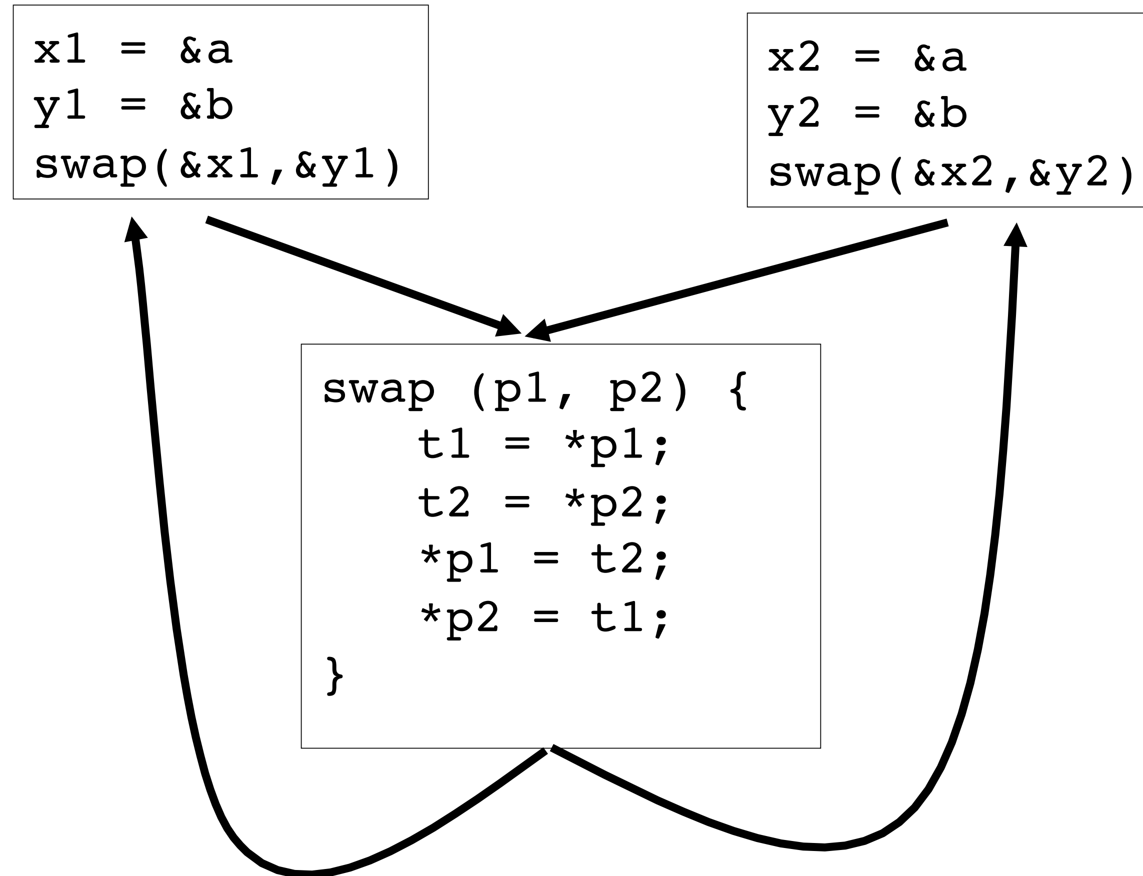
- **Flow-sensitive analysis:** compute state (points-to graph in our case) for each program code\basic block
- **Flow-insensitive analysis:** compute least upper bound of states at any time in program execution

- Persistent data structure are vital for flow sensitive analyses.

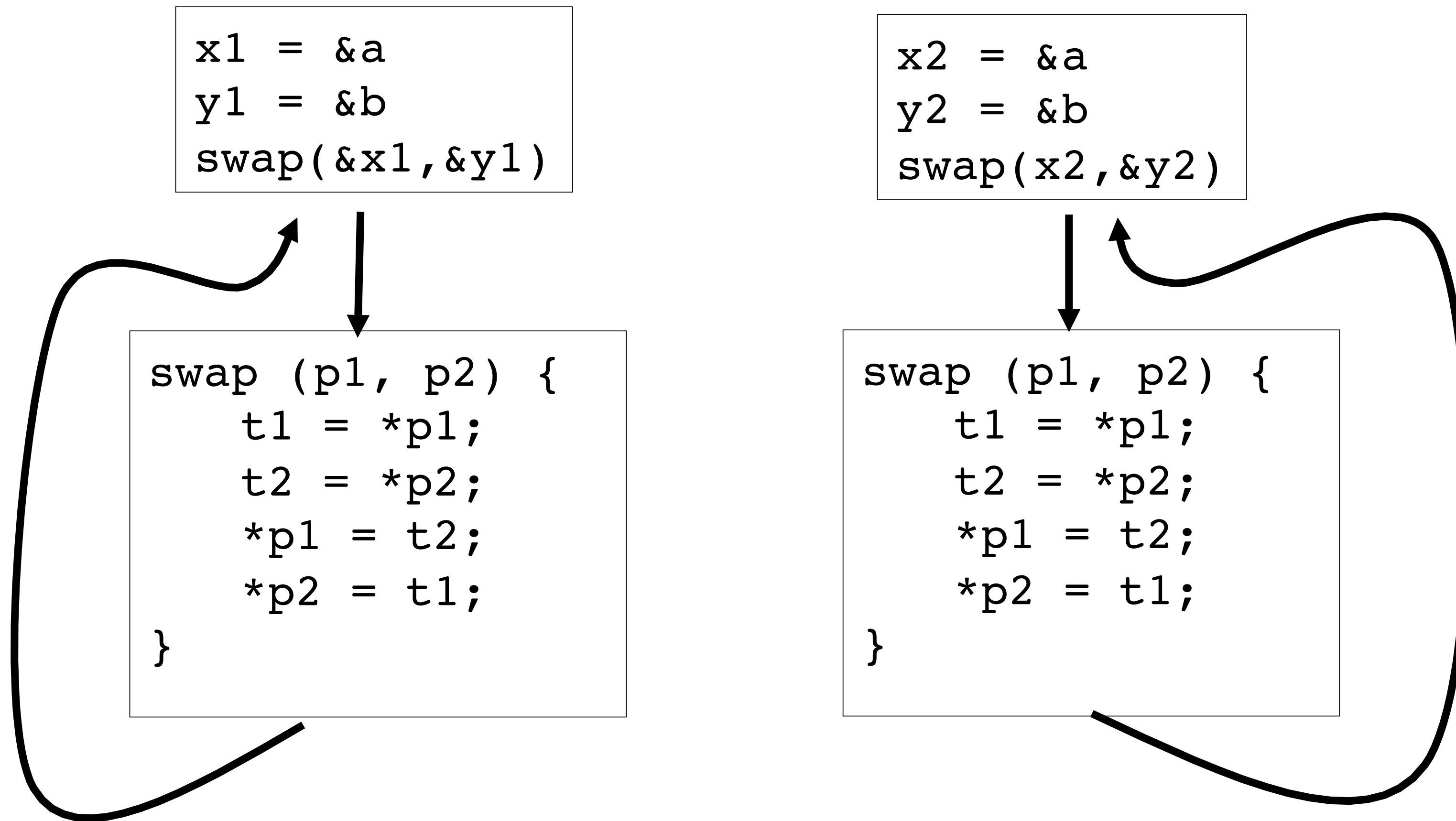
Context Sensitivity

- Is calling context considered?
- Context-sensitive approach:
  - treat each function call separately just like real program execution would
  - problem: what do we do for recursive functions?
    - approximate
  - Can be exponential
- Context-insensitive approach:
  - merge information from all call sites of a particular function
  - in effect, inter-procedural analysis problem is reduced to intra-procedural analysis problem

Context-insensitive approach



Context-sensitive approach



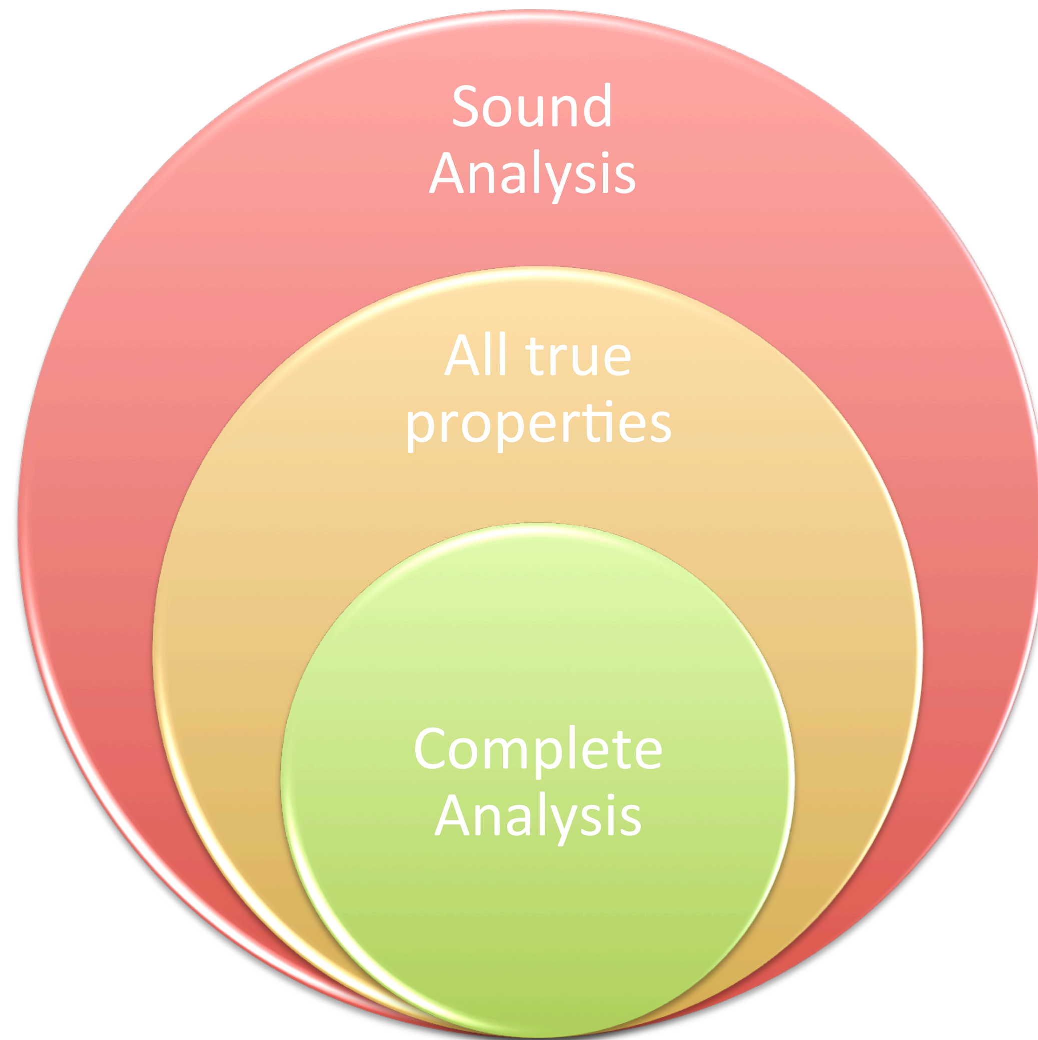
Dimensions (cont.)

- Memory modeling
  - For global\automatic variables, easy: use a single "node"
  - For dynamic memory – problem, multiple allocations. suggestions:
    - For each alloc – one node
      - Context\path sensitivity?
    - One node for each type
    - Create abstract shape of heap (shape analysis)

Abstraction

- Undecidable problems
- *Abstract* problem to decidable overapproximation or underapproximation

- **Sound static analysis** overapproximates
  - Identifies all properties but may report some false alarms that cannot actually occur.
  - No false negatives.
- **Complete static analysis** underapproximates
  - Any property reported corresponds to actual one, but no guarantee that all actual properties will be reported



We will discuss (mainly) sound analysis

Steensgaard’s Algorithm

Problem Statement

- Consider may points-to analysis.
- *Flow insensitive*
  - Compute just one graph for the entire program
  - Consider all statements regardless of control-flow
  - SSA form makes flow sensitivity less worthwhile
- *Context insensitive* (for now)
- Pointer actions consist of following types:
  - p = &a (address of, includes allocation statements)
  - p = q
  - p = *q
  - *p = q
- Want to compute relation pts : $P \cup A \rightarrow 2^A$

Steensgaard's Algorithm

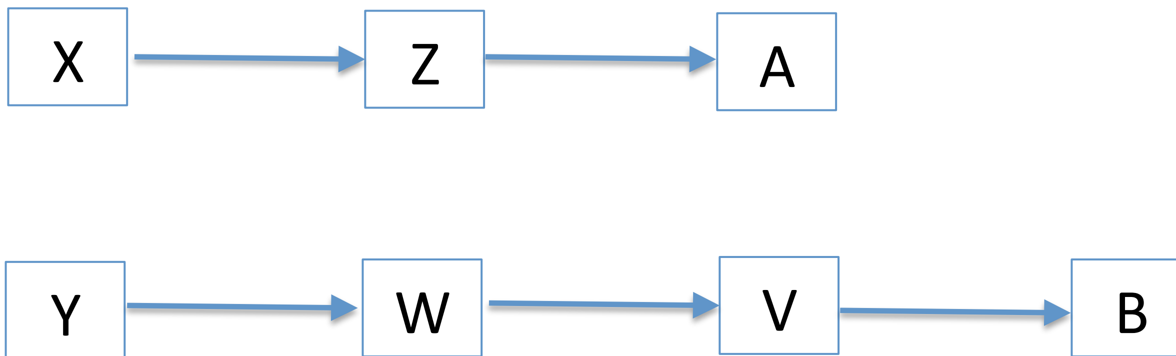
- View memory slots as abstract types (roles) in the program.
- Each pointer can point to one type only
  - Strong intuition for strongly typed language, such as Java. What about C\x8086?
- View pointer assignments as constraints on types
- Features
  - Also functions are considered memory slots

Formulation as constraints

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

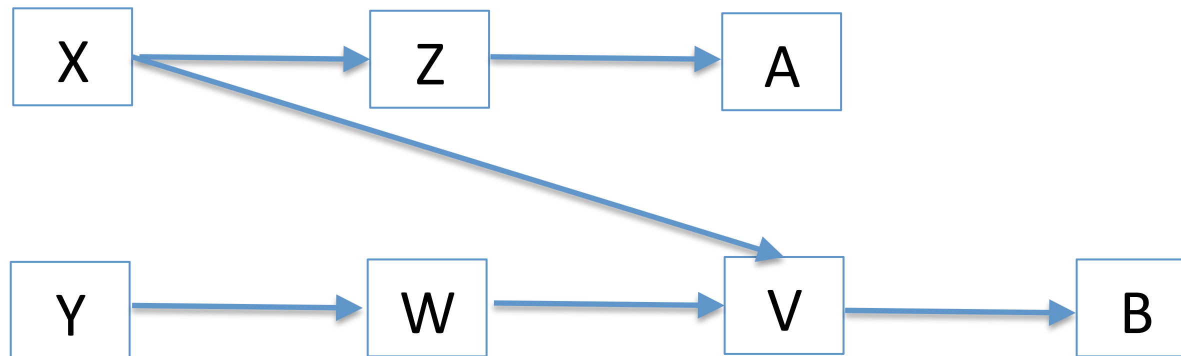
Steensgaar’d Algorithm - Example

z = &A



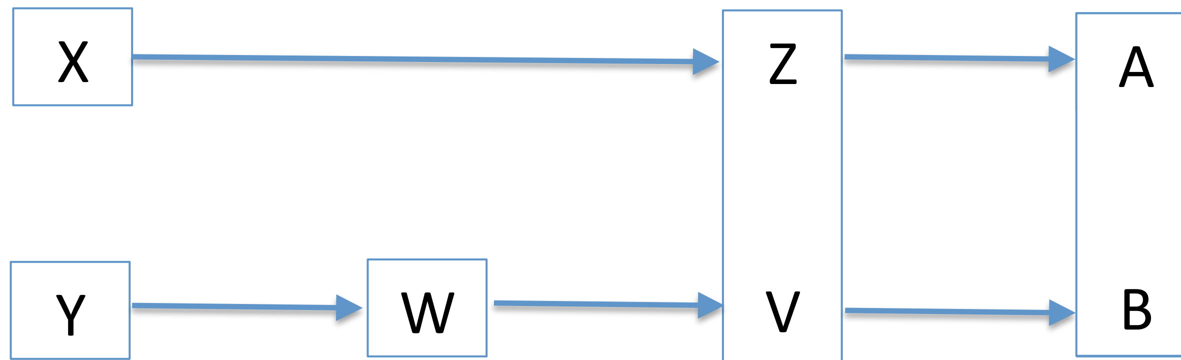
Steensgaar’d Algorithm - Example

x = *y



Steensgaar'd Algorithm - Example

x = *y



Original Formulation as Typing rules

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \quad A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

pts(X) = pts(Y)

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\tau \times _) \quad A \vdash \mathsf{y} : \tau}{A \vdash \mathit{welltyped}(\mathsf{x} = \&\mathsf{y})}$$

loc(Y) ∈ pts(X)

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \quad A \vdash \mathsf{y} : \mathbf{ref}(\mathbf{ref}(\alpha_2) \times _) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash \mathit{welltyped}(\mathsf{x} = *\mathsf{y})}$$

∀ v∈pts(Y). pts(X) = pts(v)

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha) \quad A \vdash \mathsf{y}_i : \mathbf{ref}(\alpha_i) \quad \forall i \in [1 \dots n] : \alpha_i \trianglelefteq \alpha}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{op}(\mathsf{y}_1 \dots \mathsf{y}_n))}$$

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\mathbf{ref}(_) \times _)}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{allocate}(\mathsf{y}))}$$

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\mathbf{ref}(\alpha_1) \times _) \quad A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash \mathit{welltyped}(*\mathsf{x} = \mathsf{y})}$$

∀ v∈pts(X). pts(v) = pts(Y)

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(_ \times \mathbf{lam}(\alpha_1 \dots \alpha_n)(\alpha_{n+1} \dots \alpha_{n+m})) \quad A \vdash \mathsf{f}_i : \mathbf{ref}(\alpha_i) \quad A \vdash \mathsf{r}_j : \mathbf{ref}(\alpha_{n+j}) \quad \forall s \in S^* : A \vdash \mathit{welltyped}(s)}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{fun}(\mathsf{f}_1 \dots \mathsf{f}_n) \rightarrow (\mathsf{r}_1 \dots \mathsf{r}_m)\ S^*)}$$

$$\frac{A \vdash \mathsf{x}_j : \mathbf{ref}(\alpha'_{n+j}) \quad A \vdash \mathsf{p} : \mathbf{ref}(_ \times \mathbf{lam}(\alpha_1 \dots \alpha_n)(\alpha_{n+1} \dots \alpha_{n+m})) \quad A \vdash \mathsf{y}_i : \mathbf{ref}(\alpha'_i) \quad \forall i \in [1 \dots n] : \alpha'_i \trianglelefteq \alpha_i \quad \forall j \in [1 \dots m] : \alpha_{n+j} \trianglelefteq \alpha'_{n+j}}{A \vdash \mathit{welltyped}(\mathsf{x}_1 \dots \mathsf{x}_m = \mathsf{p}(\mathsf{y}_1 \dots \mathsf{y}_n))}$$

Steensgaard's Algorithm - Implementation

- Can be efficiently implemented using Union - Find algorithm
  - Nearly linear time: $O(n\alpha(n))$
  - $\alpha(2^{80}) \leq 4$
- Each constraint is processed just once!

Steensgaar'd Algorithm - Advantages

- Very efficient
- Interprocedural
  – each invocation provides a copy constraints on formal parameters
- Modification for context sensitivity is possible
  – Due to high efficiency of context insensitive version
  – Summaries for non-recursive function easy
- Fast preliminary understanding of where data flows, where the "interesting logic" is.

Steensgaard's Algorithm – Shortcoming

- Imprecise analysis
- Example:

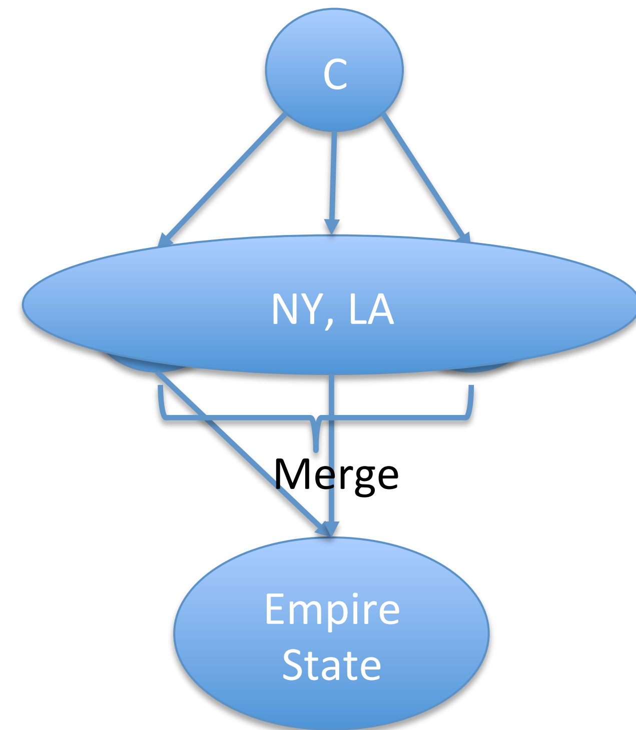
City NY

City LA

NY->building = Empire-State

If (query == 'NY') {c = &NY}

Else {c = &LA}



Alternative Algorithms and Additional Features

A different variation: Andersen's Algorithm

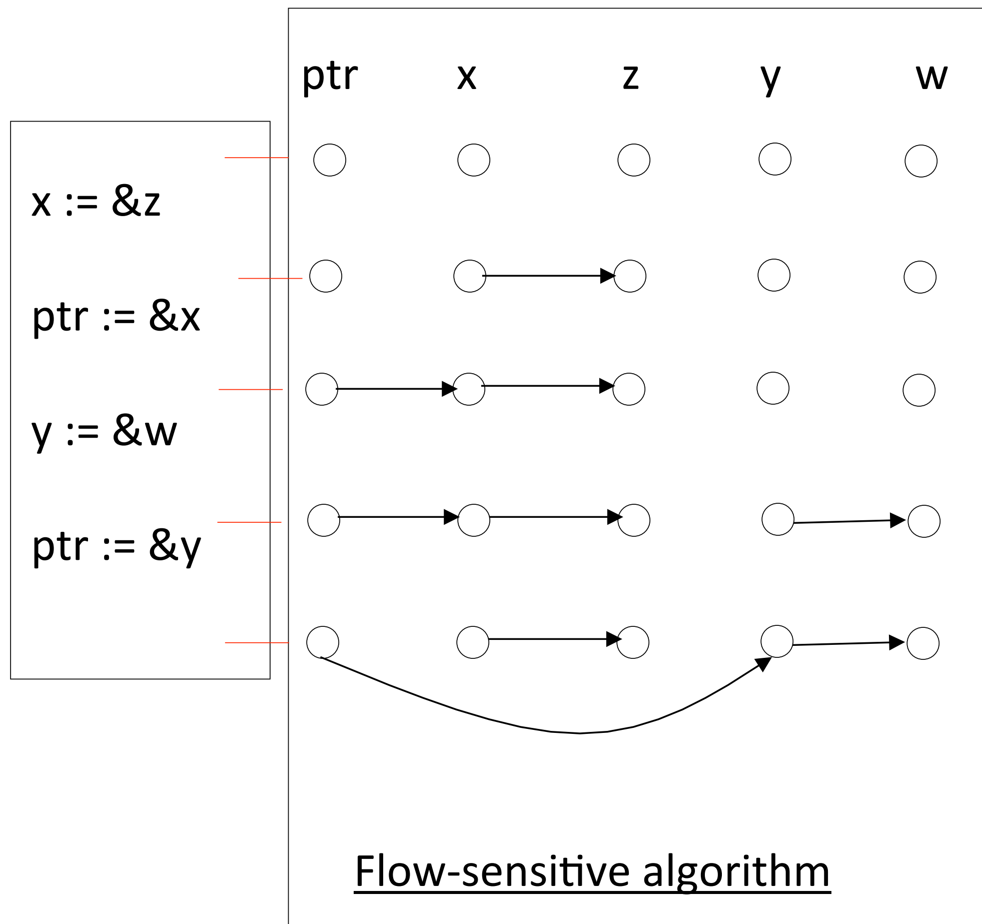
- Pointer assignments are *subset constraints*

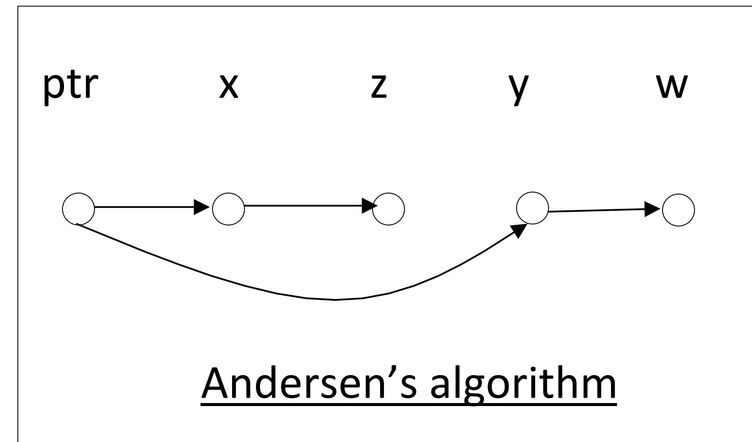
Constraint type	Assignment	Constraint	Meaning
Base (address)	a = &b	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
Simple (copy)	a = b	$a \supseteq b$	$pts(a) \supseteq pts(b)$
Complex (load)	a = *b	$a \supseteq *b$	$\forall v \in pts(b).\ pts(a) \supseteq pts(v)$
Complex (store)	*a = b	$*a \supseteq b$	$\forall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

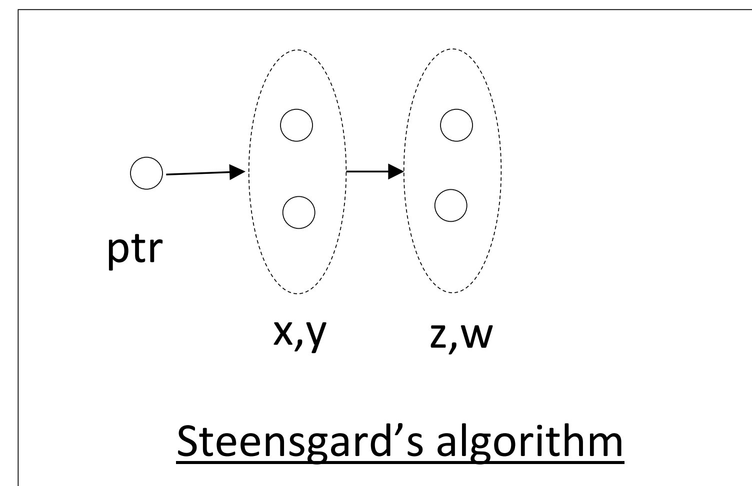
Andersen's Algorithm

- Compute minimal point under constraints
  - Work queue algorithm.
- Complexity $O(n^3)$.
  - In practice much better.
- Improvements
  - Copy edge cycle detection & contraction
    - [Hardekopf-Lin 07]
  - Strong updates
    - if out-degree of x in points-to graph is 1, we can do a strong update even for *x := …
    - Representation might require sensitivity to flow.
    - [Lhotak-Chung 11]
  - Analysis of Linux kernel in under two minutes

Summary of Steensgard, Andersen







More Practical Issues

- Imprecise analysis of structs, arrays.
  - neglecting offsets in analysis make it crude
    - "Field Sensitivity"
    - Easier in Java, harder in low-level languages
  - What about variable offsets?
    - Arrays, heaps…
    - Consider a nested packet parser
- "star" procedures cause massive spurious dataflow
- Imported procedure require stabs

Measures of Precision

- Average size of aliased sets
  - careful… depends on memory model.
- Compare to dynamic (fuzzer) reports
- Clients’ precision

Hybrid Algorithms

- D- level flow [Das 00]
  - Use Andersen's rules at top d-levels of PTG, Steensgaard's elsewhere
  - Assuming not many d-deep dereferences are taken
  - *Idea*: Adapt to approach context sensitive algorithm
- K-limiting points-to set [Shapiro-Horwitz97]
  - Steensgaard's algorithm can be thought as restricting the out-degree of the graph produced by Andersen's algorithm to 1, by merging nodes when that is exceeded
- Object sensitivity [Milanova et. al 02]
  - Inspired by context sensitivity
  - For context, use stack of method-invoking objects

Heuristics

- Limited area of enhanced precision
  - [Ryder et. al ’99]
- Probabilistic pointer analysis
  - Seems more useful for optimization

Conclusions

- Pointer analysis is a complex and developed field in static analysis.
- Highly useful in program analysis and optimization
- Many dimensions
  - heap modeling, offset classes, flow sensitivity, context sensitivity
  - $\Rightarrow$ different clients with different needs
  - Tradeoffs

Conclusions (cont.)

- Flow insensitive analysis as a first step is feasible nowadays
- Challenges in pointer analysis:
  - Context (Object) sensitivity
  - Path sensitivity
  - Modeling heap (Shape analysis)

Questions?

Discussion

References

- B. Steensgaard: **Points-to Analysis in Almost Linear Time**. POPL 1996: 32-41
- M. Hind: **Pointer analysis: haven't we solved this problem yet?** PASTE 2001: 54-61
- U. P. Khedker, A. Mycroft, P. Singh Rawat: **Liveness-Based Pointer Analysis**. SAS 2012: 265-282
- S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, E. Geay: **Effective typestate verification in the presence of aliasing**. ACM Trans. Softw. Eng. Methodol. 17(2) (2008)
- B. Hardekopf, C. Lin: **The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code.** PLDI 2007: 290-299
- O. Lhoták, K. Chung: **Points-to analysis with efficient strong updates**. POPL 2011: 3-16
- A. Milanova, A. Rountev, B. Ryder: **Parameterized object sensitivity for points-to and side-effect analyses for Java.** ISSTA 2002: 1-11
- M. Das: **Unification-based pointer analysis with directional assignments**. PLDI 2000: 35-46
- M. Shapiro, S. Horwitz: **Fast and Accurate Flow-Insensitive Points-To Analysis**. POPL 1997: 1-14
- A. Rountev, B. Ryder, W. Landi: **Data-Flow Analysis of Program Fragments**. ESEC / SIGSOFT FSE 1999: 235-252