Program Analysis and Verification

0368-4479

Noam Rinetzky

Lecture 10: Pointer Analysis

Slides credit: Roman Manevich, Mooly Sagiv, Eran Yahav

Abstract Interpretation [Cousot'77]

- Mathematical foundation of static analysis
  - Abstract domains
    - Abstract states
    - Join (⊔)
  - Transformer functions
    - Abstract steps
  - Chaotic iteration
    - Abstract computation
    - Structured Programs

Lattices
(D, ⊑, ⊔, ⊓, ⊥, ⊤)

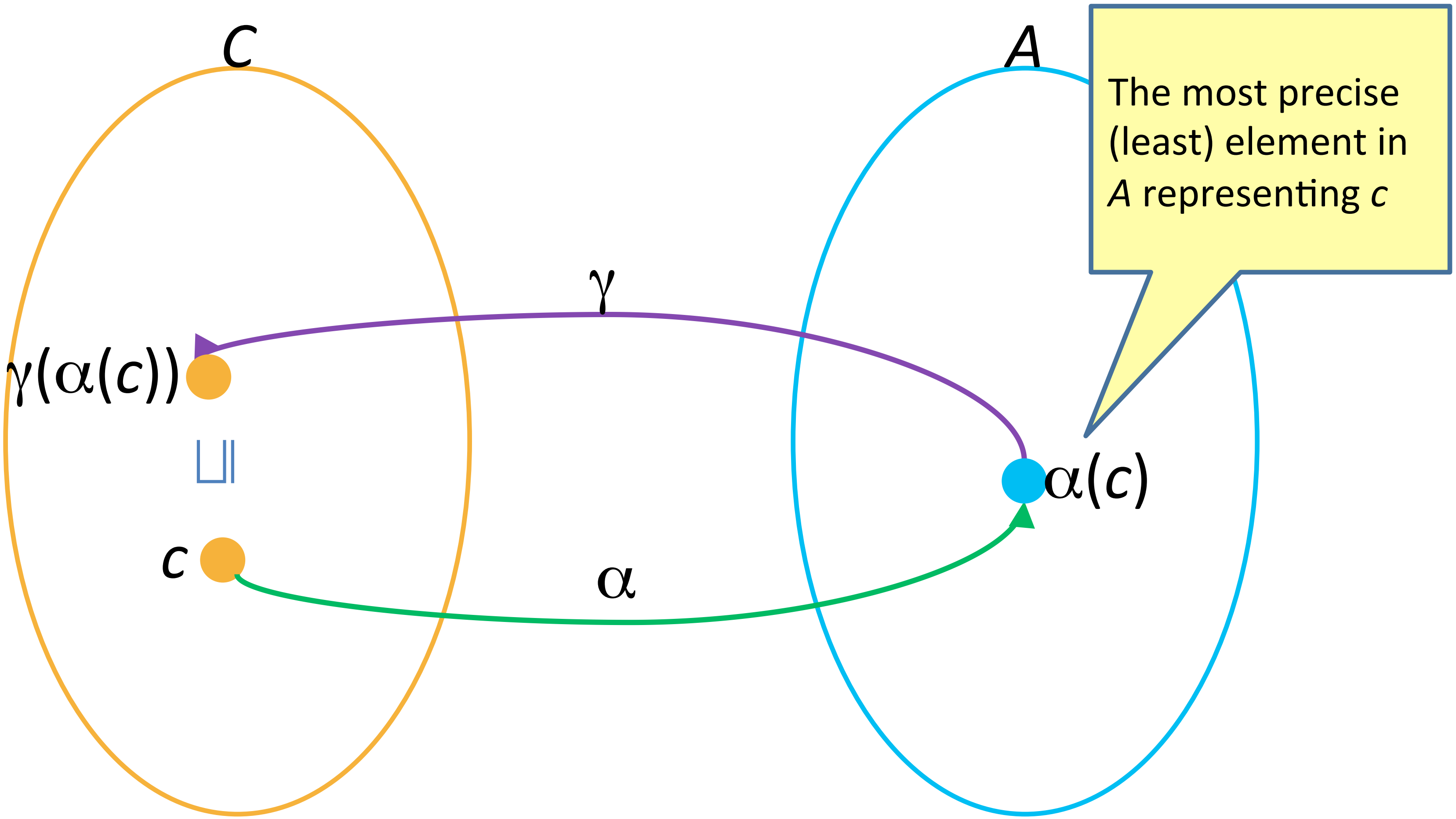
Monotonic functions

Fixpoints

The collecting lattice

- Lattice for a given control-flow node v:
  $L_v = (2^{\mathbf{State}}, \subseteq, \cup, \cap, \emptyset, \mathbf{State})$
- Lattice for entire control-flow graph with nodes V:
  $$L_{CFG} = \mathrm{Map}(V, L_v)$$
- We will use this lattice as a baseline for static analysis and define abstractions of its elements

Galois Connection

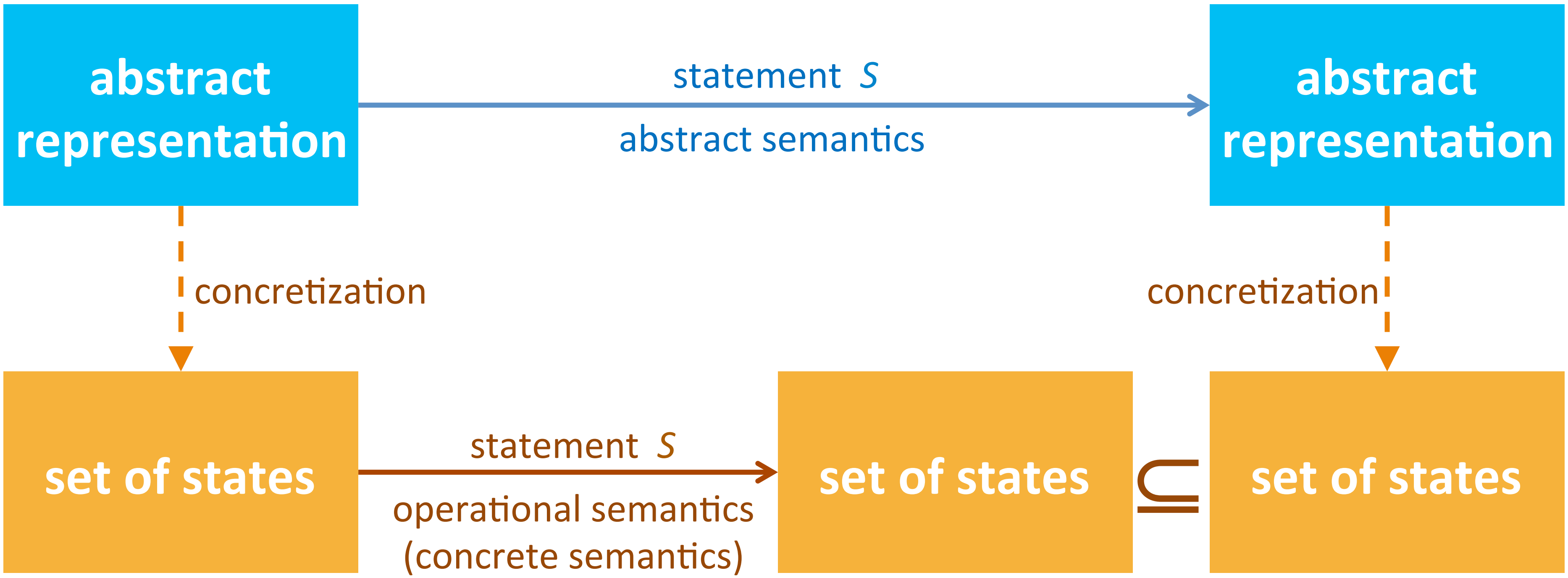


$$c \sqsubseteq \gamma(\alpha(c))$$

Galois Connection

- Given two complete lattices
  $C = (D^C, \sqsubseteq^C, \sqcup^C, \sqcap^C, \bot^C, \top^C)$ – concrete domain
  $A = (D^A, \sqsubseteq^A, \sqcup^A, \sqcap^A, \bot^A, \top^A)$ – abstract domain
- A Galois Connection (GC) is quadruple (C, α, γ, A) that relates C and A via the monotone functions
  - The abstraction function $\alpha : D^C \rightarrow D^A$
  - The concretization function $\gamma : D^A \rightarrow D^C$
- for every concrete element $c \in D^C$ and abstract element $a \in D^A$
  $\alpha(\gamma(a)) \sqsubseteq a$ and $c \sqsubseteq \gamma(\alpha(c))$
- Alternatively $\alpha(c) \sqsubseteq a$ iff $c \sqsubseteq \gamma(a)$

Abstract (conservative) interpretation



Plan

- Understand the problem
- Mention some applications
- Simplified problem
  - Only variables (no object allocation)
- Reference analysis
- Andersen’s analysis
- Steensgaard’s analysis
- Generalize to handle object allocation

Constant propagation example

```
x = 3;

y = 4;

z = x + 5;
```

Constant propagation example with pointers

```
x = 3;

*p = 4;

z = x + 5;
```

Is x always 3 here?

Constant propagation example with pointers

pointers affect
most program analyses

```
p = &y;
x = 3;
*p = 4;
z = x + 5;
```

x is always 3

```
else
  p = &y;
x = 3;
*p = 4;
z = x + 5;
```

x may be 3 or 4
(i.e., x is unknown in our lattice)

```
= &x;
= 3;
*p = 4;
z = x + 5;
```

x is always 4

Constant propagation example with pointers

```
p = &y;
x = 3;
*p = 4;
z = x + 5;
```

p always points-to y

```
if (?)
  p = &x;
else
  p = &y;
x = 3;
*p = 4;
z = x + 5;
```

p may point-to x or y

```
p = &x;
x = 3;
*p = 4;
z = x + 5;
```

p always points-to x

Points-to Analysis

- Determine the set of targets a pointer variable could point-to (at different points in the program)
  - "p points-to x"
    - "p stores the value &x"
    - "*p denotes the location x"
  - targets could be variables or locations in the heap (dynamic memory allocation)
    - p = &x;
    - p = new Foo(); or p = malloc (…);
  - must-point-to vs. may-point-to

Constant propagation example with pointers

```
*q = 3;

*p = 4;

z = *q + 5;
```

Can *p denote the same location as *q?

what values can this take?

More terminology

- *p and *q are said to be aliases (in a given concrete state) if they represent the same location
- Alias analysis
  - Determine if a given pair of references could be aliases at a given program point
  - *p may-alias *q
  - *p must-alias *q

Pointer Analysis

- Points-To Analysis
  - may-point-to
  - must-point-to

- Alias Analysis
  - may-alias
  - must-alias

Applications

- Compiler optimizations
  - Method de-virtualization
  - Call graph construction
  - Allocating objects on stack via escape analysis
- Verification & Bug Finding
  - Datarace detection
  - Use in preliminary phases
  - Use in verification itself

Points-to analysis: a simple example

`p = &x;`	{p=&x}
`q = &y;`	{p=&x ∧ q=&y}
`if (?) {`	
`  q = p;`	{p=&x ∧ q=&x}
`}`	{p=&x ∧ (q=&y ∨ q=&x)}
`x = &a;`	{p=&x ∧ (q=&y ∨ q=&x) ∧ x=&a}
`y = &b;`	{p=&x ∧ (q=&y ∨ q=&x) ∧ x=&a ∧ y=&b}
`z = *q;`	{p=&x ∧ (q=&y∨q=&x) ∧ x=&a ∧ y=&b ∧ (z=x∨z=y)}

We will usually drop variable-equality information

How would you construct an abstract domain to represent these abstract states?

Points-to lattice

- **P**oints-**t**o
  - *PT-factoids*[x] = { *x*=&*y* | *y* $\in$ Var} $\cup$ false
    PT[x] = ($2^{PT\text{-}factoids}$, $\subseteq$, $\cup$, $\cap$, *false*, *PT-factoids*[x])
    (interpreted disjunctively)
- How should combine them to get the abstract states in the example?
  `{p=&x ∧ (q=&y∨q=&x) ∧ x=&a ∧ y=&b}`

Points-to lattice

- **P**oints-**t**o
  - $PT\text{-}factoids[x] = \{ x{=}\&y \mid y \in \text{Var}\} \cup \text{false}$
    $PT[x] = (2^{PT\text{-}factoids}, \subseteq, \cup, \cap, false, PT\text{-}factoids[x])$
    (interpreted disjunctively)
- How should combine them to get the abstract states in the example?
  `{p=&x` $\wedge$ `(q=&y`$\vee$`q=&x)` $\wedge$ `x=&a` $\wedge$ `y=&b}`
- $D[x] = \text{Disj}(VE[x]) \times \text{Disj}(PT[x])$
- For all program variables: $D = D[x_1] \times \ldots \times D[x_k]$

Points-to analysis

```
a = &y
x = &a;
y = &b;
if (?) {
  p = &x;
} else {
  p = &y;
}

*x = &c;
*p = &c;
```

How should we handle this statement?

Strong update

{x=&a ∧ y=&b ∧ (p=&x∨p=&y) ∧ ~~a=&y~~}

{x=&a ∧ y=&b ∧ (p=&x∨p=&y) ∧ a=&c}

{(x=&a∨x=&c) ∧ (y=&b∨y=&c) ∧ (p=&x∨p=&y)}

Weak update

Questions

- When is it correct to use a strong update? A weak update?
- Is this points-to analysis precise?
- What does it mean to say
  - p must-point-to x at program point u
  - p may-point-to x at program point u
  - p must-not-point-to x at program u
  - p may-not-point-to x at program u

Points-to analysis, formally

- We must formally define what we want to compute before we can answer many such questions

PWhile syntax

- A primitive statement is of the form
  - x := null
  - x := y
  - x := *y
  - x := &y;
  - *x := y
  - skip

  (where x and y are variables in Var)

Omitted (for now)
- Dynamic memory allocation
- Pointer arithmetic
- Structures and fields
- Procedures

PWhile operational semantics

- **State** : $(Var \rightarrow Z) \cup (Var \rightarrow Var \cup \{null\})$
- $[\![$ x = y $]\!]$ s =
- $[\![$ x = *y $]\!]$ s =
- $[\![$ *x = y $]\!]$ s =
- $[\![$ x = null $]\!]$ s =
- $[\![$ x = &y $]\!]$ s =

PWhile operational semantics

- **State** : (Var→Z) ∪ (Var→Var∪{null})
- ⟦ x = y ⟧ s = s[x↦s(y)]
- ⟦ x = *y ⟧ s = s[x↦s(s(y))]
- ⟦ *x = y ⟧ s = s[s(x)↦s(y)]
- ⟦ x = null ⟧ s = s[x↦null]
- ⟦ x = &y ⟧ s = s[x↦y]

must say what happens if null is dereferenced

PWhile collecting semantics

- *CS*[*u*] = set of concrete states that can reach program point *u* (CFG node)

Ideal PT Analysis: formal definition

- Let u denote a node in the CFG

- Define IdealMustPT(u) to be

  $\{ (p,x) \mid$ **forall** s in $CS[u].\ s(p) = x \}$

- Define IdealMayPT(u) to be

  $\{ (p,x) \mid$ **exists** s in $CS[u].\ s(p) = x \}$

May-point-to analysis: formal Requirement specification

May/Must Point-To Analysis

may

Compute R: V -> $2^{Vars'}$ such that
R(u) ⊇ IdealMayPT(u)

must

For every vertex u in the CFG,
compute a set R(u) such that
R(u) ⊆ { (p,x) | ∃s∈CS[u]. s(p) = x }

Var' = Var U {null}

May-point-to analysis: formal Requirement specification

Compute R: V -> $2^{Vars'}$ such that
$R(u) \supseteq IdealMayPT(u)$

- An algorithm is said to be correct if the solution R it computes satisfies

$$\forall u \in V.\ R(u) \supseteq IdealMayPT(u)$$

- An algorithm is said to be precise if the solution R it computes satisfies

$$\forall u \in V.\ R(u) = IdealMayPT(u)$$

- An algorithm that computes a solution R_1 is said to be more precise than one that computes a solution R_2 if

$$\forall u \in V.\ R_1(u) \subseteq R_2(u)$$

(May-point-to analysis) *Algorithm A*

- Is this algorithm correct?
- Is this algorithm precise?

- Let's first completely and formally define the algorithm

Points-to graphs

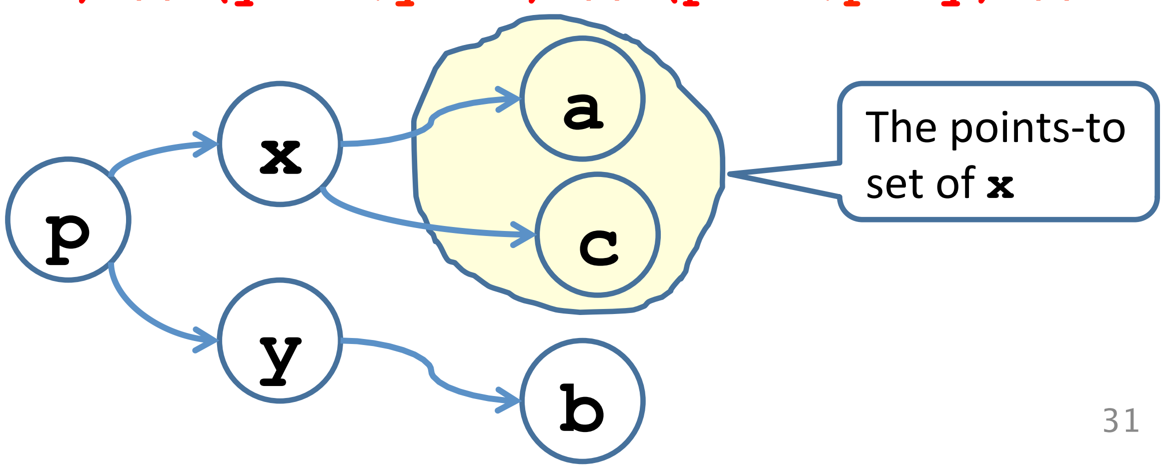
```
x = &a;
y = &b;
if (?) {
  p = &x;
} else {
  p = &y;
}

*x = &c;
*p = &c;
```

{x=&a ∧ y=&b ∧ (p=&x∨p=&y)}

{x=&a ∧ y=&b ∧ (p=&x∨p=&y) ∧ a=&c}

{(x=&a∨x=&c) ∧ (y=&b∨y=&c) ∧ (p=&x∨p=&y) ∧ a=&c}



Algorithm A: A formal definition the "Data Flow Analysis" Recipe

- Define join-semilattice of abstract-values
  - PTGraph ::= (Var, Var$\times$Var')
  - $g_1 \sqcup g_2$ = ?
  - $\bot$ = ?
  - $\top$ = ?
- Define transformers for primitive statements
  - $[\![\text{stmt}]\!]^{\#}$: PTGraph $\rightarrow$ PTGraph

Algorithm A: A formal definition the "Data Flow Analysis" Recipe

- Define join-semilattice of abstract-values
  - PTGraph ::= (Var, Var$\times$Var')
  - $g_1 \sqcup g_2$ = (Var, $E_1 \cup E_2$)
  - $\bot$ = (Var, {})
  - $\top$ = (Var, Var$\times$Var')
- Define transformers for primitive statements
  - $[\![\text{stmt}]\!]^\#$: PTGraph $\rightarrow$ PTGraph

Algorithm A: transformers

- Abstract transformers for primitive statements
  - $[\![\text{ stmt }]\!]^\#$: PTGraph → PTGraph
- $[\![\text{ x := y }]\!]^\#$ (Var, E) = ?
- $[\![\text{ x := null }]\!]^\#$ (Var, E) = ?
- $[\![\text{ x := \&y }]\!]^\#$ (Var, E) = ?
- $[\![\text{ x := *y }]\!]^\#$ (Var, E) = ?
- $[\![\text{ *x := \&y }]\!]^\#$ (Var, E) = ?

Algorithm A: transformers

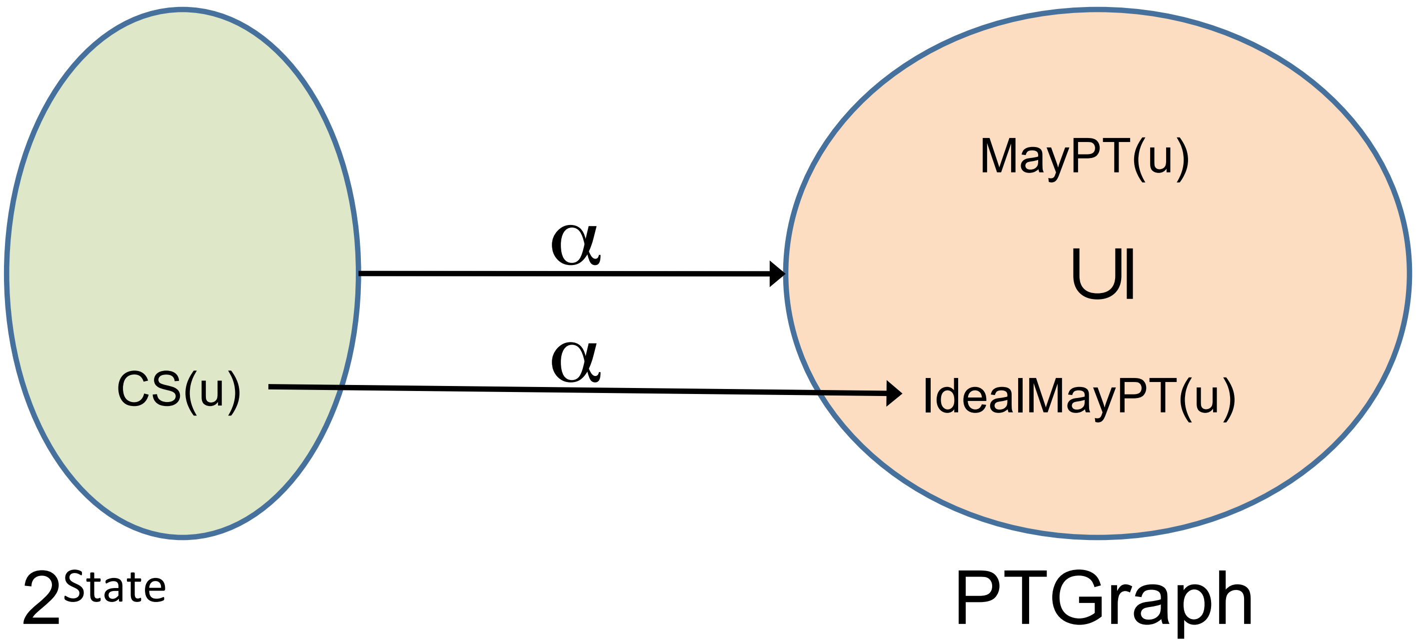
- Abstract transformers for primitive statements
  - ⟦ stmt ⟧# : PTGraph → PTGraph
- ⟦ x := y ⟧# (Var, E) = (Var, E[succ(x)=succ(y)]
- ⟦ x := null ⟧# (Var, E) = (Var, E[succ(x)={null}]
- ⟦ x := &y ⟧# (Var, E) = (Var, E[succ(x)={y}]
- ⟦ x := *y ⟧# (Var, E) = (Var, E[succ(x)=succ(succ(y))]
- ⟦ *x := &y ⟧# (Var, E) = ???

Correctness & precision

- We have a complete & formal definition of the problem
- We have a complete & formal definition of a proposed solution

- How do we reason about the correctness & precision of the proposed solution?

Points-to analysis (abstract interpretation)



$\alpha(Y)$ = { (p,x) | exists s in Y. s(p) = x }

IdealMayPT (u) = α (CS(u))

Concrete transformers

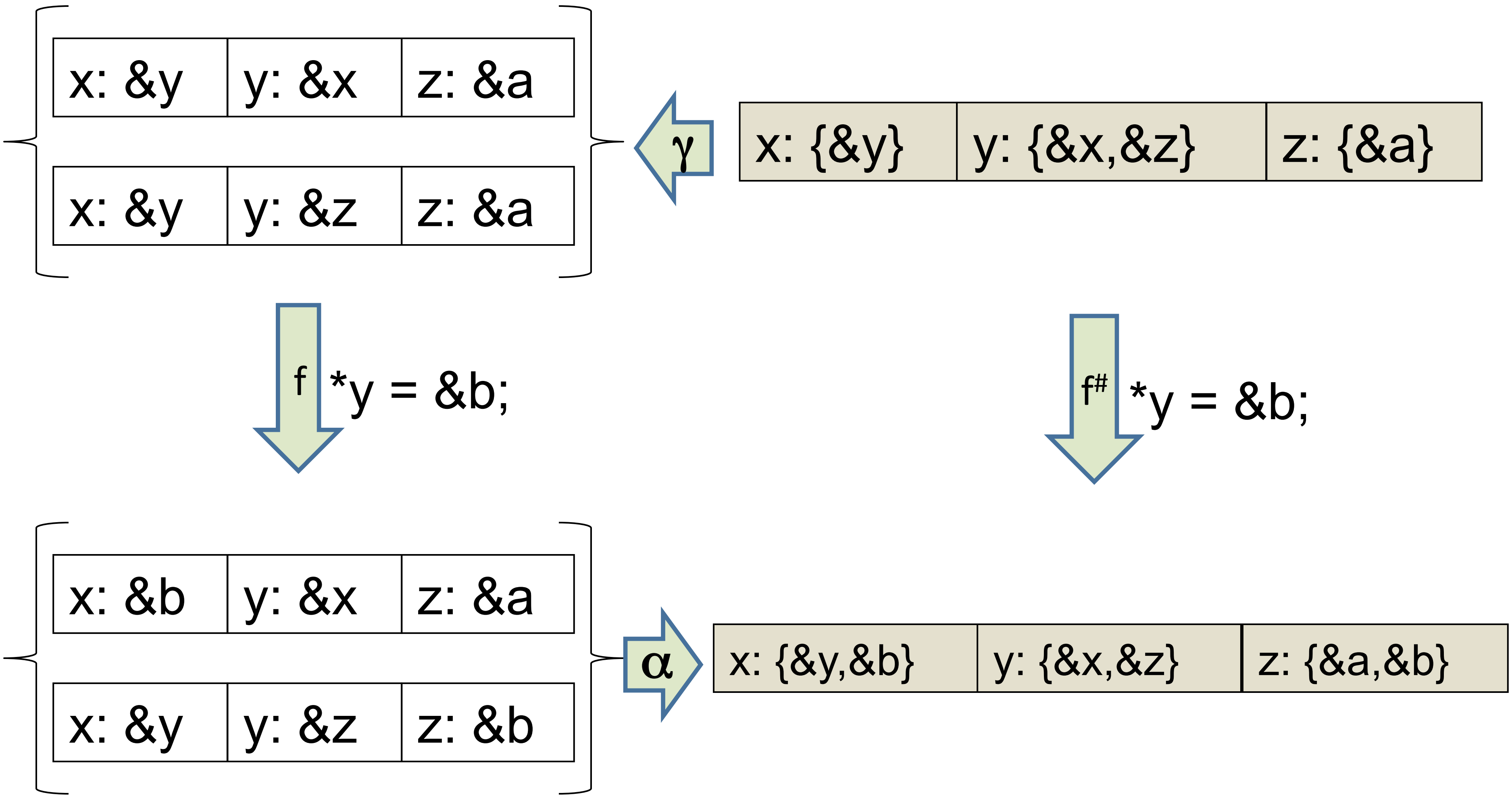
- $CS[stmt] : State \rightarrow State$
- $[\![\ x = y\ ]\!]\ s = s[x \mapsto s(y)]$
- $[\![\ x = {*y}\ ]\!]\ s = s[x \mapsto s(s(y))]$
- $[\![\ {*x} = y\ ]\!]\ s = s[s(x) \mapsto s(y)]$
- $[\![\ x = null\ ]\!]\ s = s[x \mapsto null]$
- $[\![\ x = \&y\ ]\!]\ s = s[x \mapsto y]$

- $CS^*[stmt] : 2^{State} \rightarrow 2^{State}$
- $CS^*[st]\ X = \{\ CS[st]s \mid s \in X\ \}$

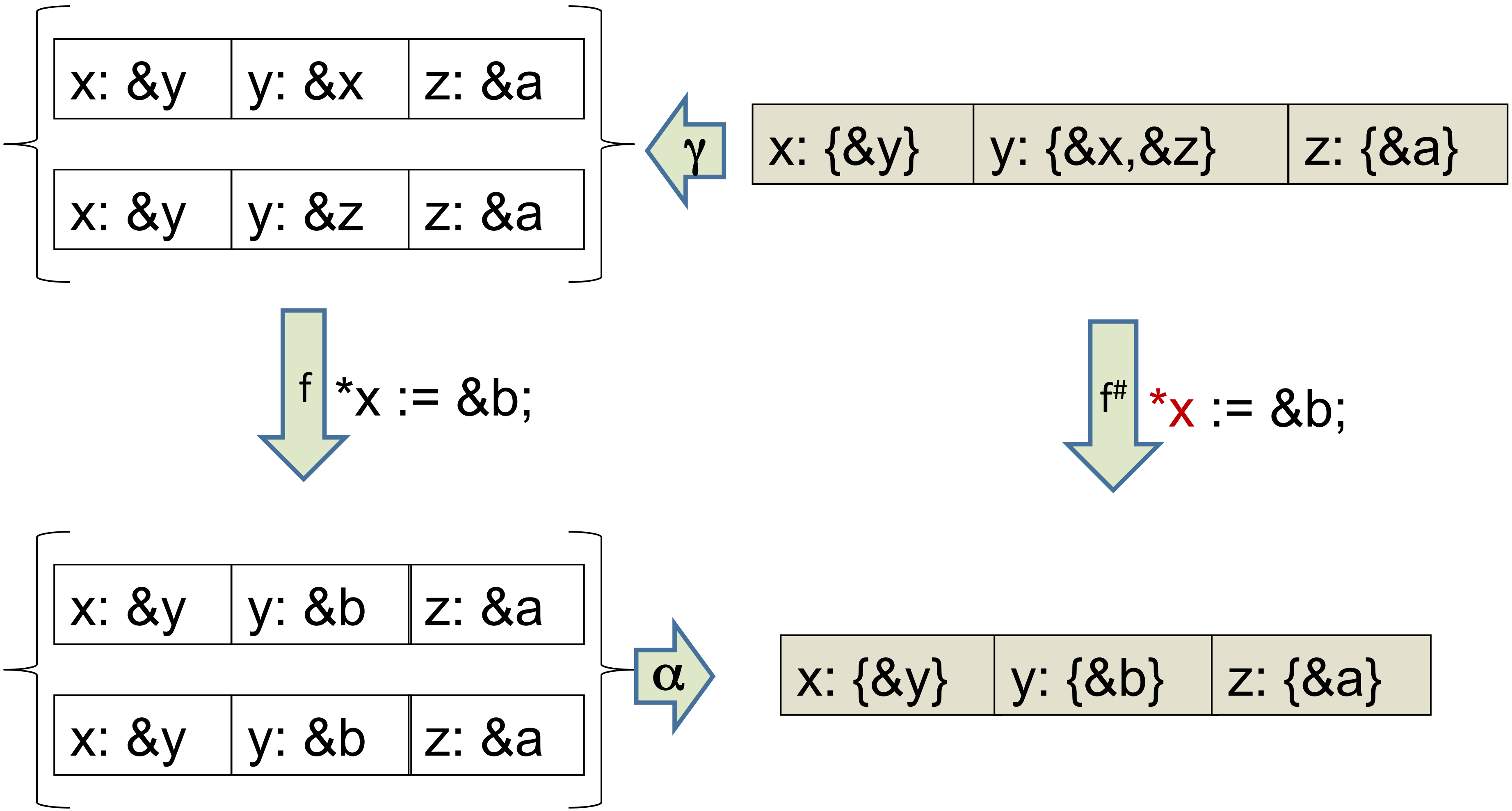
Abstract transformers

- $[\![\text{stmt}]\!]^\#$: PTGraph → PTGraph
- $[\![\text{x := y}]\!]^\#$ (Var, E) = (Var, E[succ(x)=succ(y)]
- $[\![\text{x := null}]\!]^\#$ (Var, E) = (Var, E[succ(x)={null}]
- $[\![\text{x := \&y}]\!]^\#$ (Var, E) = (Var, E[succ(x)={y}]
- $[\![\text{x := *y}]\!]^\#$ (Var, E) = (Var, E[succ(x)=succ(succ(y))]
- $[\![\text{*x := \&y}]\!]^\#$ (Var, E) = ???

Algorithm A: transformers Weak/Strong Update



Algorithm A: transformers Weak/Strong Update



Abstract transformers

- ⟦ *x := &y ⟧# (Var, E) =

  if succ(x) = {z} **then** (Var, E[succ(z)={y}]

  else succ(x)={z_1,…,z_k} where k>1

  (Var, E[succ(z_1)=succ(z_1)∪{y}]

  …

  [succ(z_k)=succ(z_k)∪{y}]

Some dimensions of pointer analysis

- Intra-procedural / inter-procedural
- Flow-sensitive / flow-insensitive
- Context-sensitive / context-insensitive
- Definiteness
  - May vs. Must
- Heap modeling
  - Field-sensitive / field-insensitive
- Representation (e.g., Points-to graph)

Andersen's Analysis

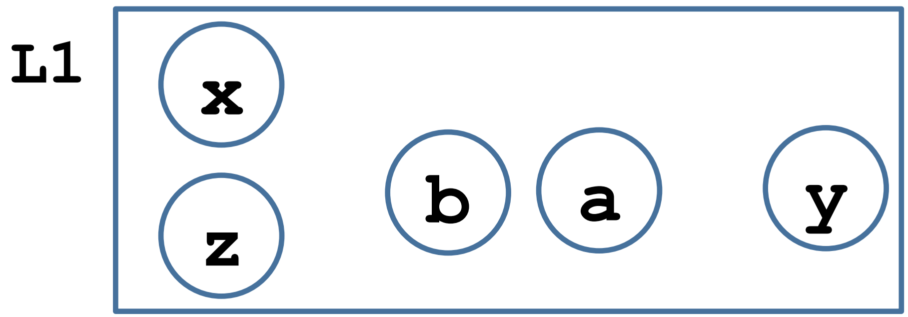
- A flow-insensitive analysis
  - Computes a single points-to solution valid at all program points
  - Ignores control-flow – treats program as a set of statements
  - Equivalent to merging all vertices into one (and applying *Algorithm A*)
  - Equivalent to adding an edge between every pair of vertices (and applying *Algorithm A*)

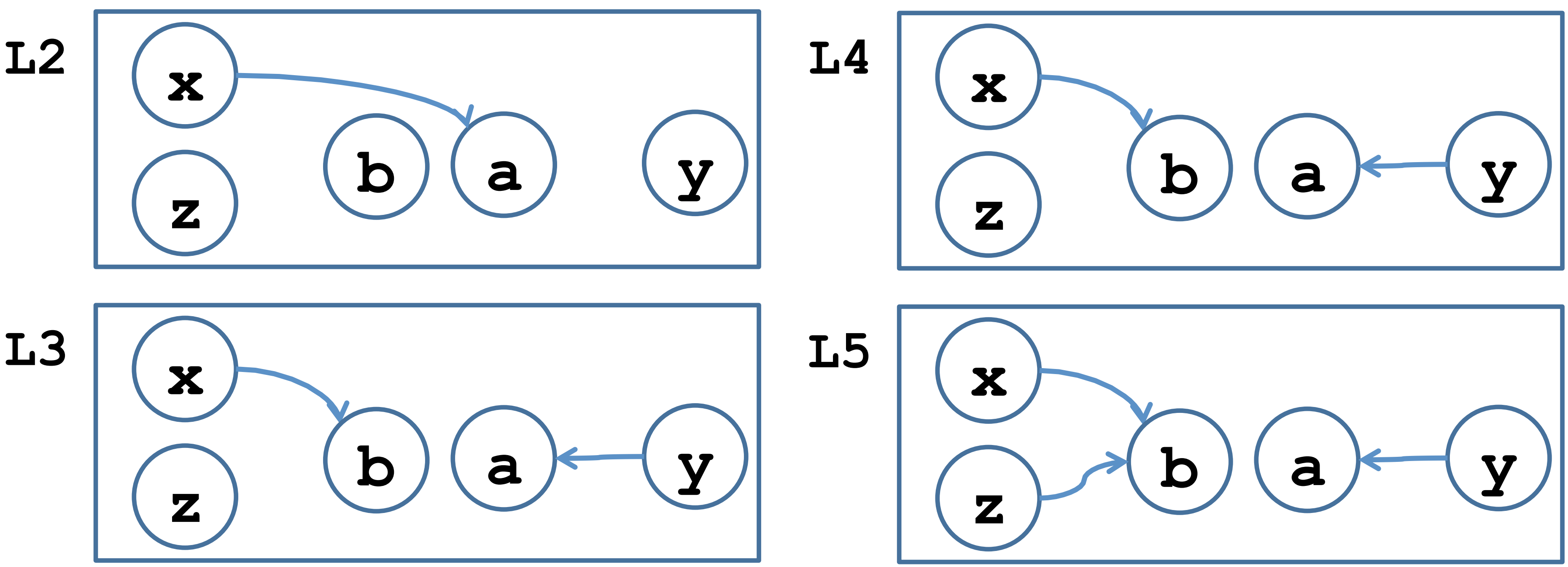
  - A (conservative) solution $R: Vars \rightarrow 2^{Vars'}$ such that

    $R \supseteq IdealMayPT(u)$ for every vertex u

Flow-sensitive analysis

```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```

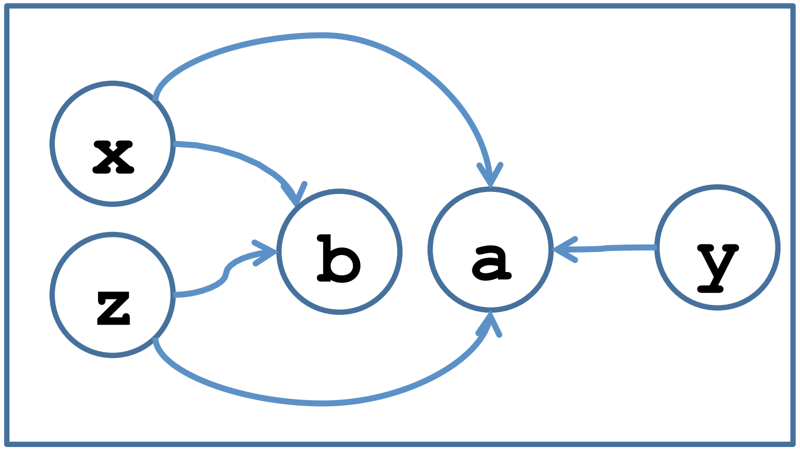




Flow-insensitive analysis

```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```

L1-5



Andersen’s analysis

- Strong updates?

- Initial state?

Why flow-insensitive analysis?

- Reduced space requirements
  - A single points-to solution
- Reduced time complexity
  - No copying
    - Individual updates more efficient
  - No need for joins
  - Number of iterations?
  - A cubic-time algorithm
- Scales to millions of lines of code
  - Most popular points-to analysis
- Conventionally used as an upper bound for precision for pointer analysis

Andersen’s analysis as set constraints

- $[\![$ x := y $]\!]^{\#}$ $PT[x] \subseteq PT[y]$
- $[\![$ x := null $]\!]^{\#}$ $PT[x] \subseteq \{null\}$
- $[\![$ x := &y $]\!]^{\#}$ $PT[x] \subseteq \{y\}$
- $[\![$ x := *y $]\!]^{\#}$ $PT[x] \subseteq PT[z]$ for all $z \in PT[y]$
- $[\![$ *x := &y $]\!]^{\#}$ $PT[z] \subseteq PT[y]$ for all $z \in PT[x]$

Cycle elimination

- Andersen-style pointer analysis is $O(n^3)$ for number of nodes in graph
  - Improve scalability by reducing n
- Important optimization
  - Detect strongly-connected components in PTGraph and collapse to a single node
    - Why? In the final result all nodes in SCC have same PT
  - How to detect cycles efficiently?
    - Some statically, some on-the-fly

Steensgaard's Analysis

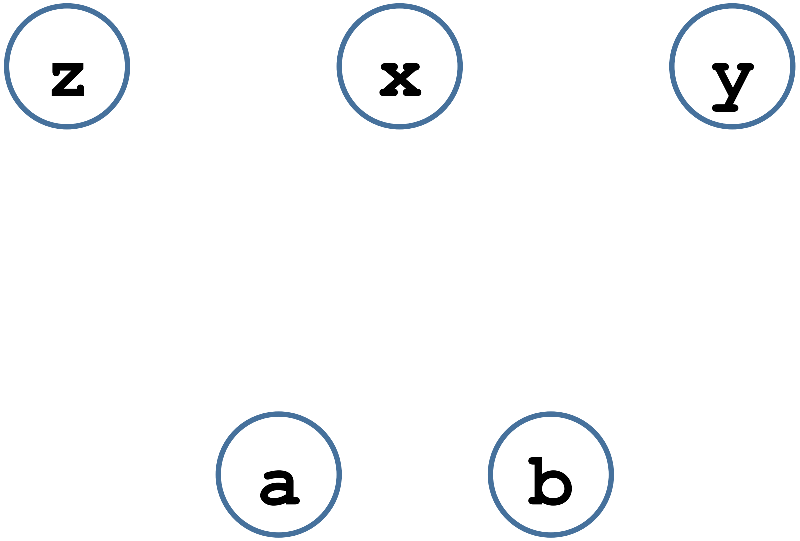
- Unification-based analysis
- Inspired by type inference
  - An assignment lhs := rhs is interpreted as a constraint that lhs and rhs have the same type
  - The type of a pointer variable is the set of variables it can point-to
- "Assignment-direction-insensitive"
  - Treats lhs := rhs as if it were both lhs := rhs and rhs := lhs

Steensgaard's Analysis

- An almost-linear time algorithm
  - Uses union-find data structure
  - Single-pass algorithm; no iteration required
- Sets a lower bound in terms of performance

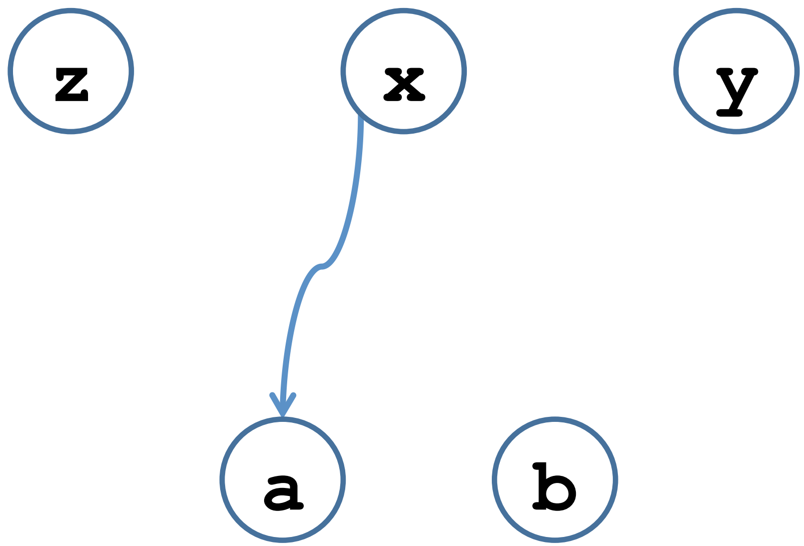
Steensgaard's analysis initialization

```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```



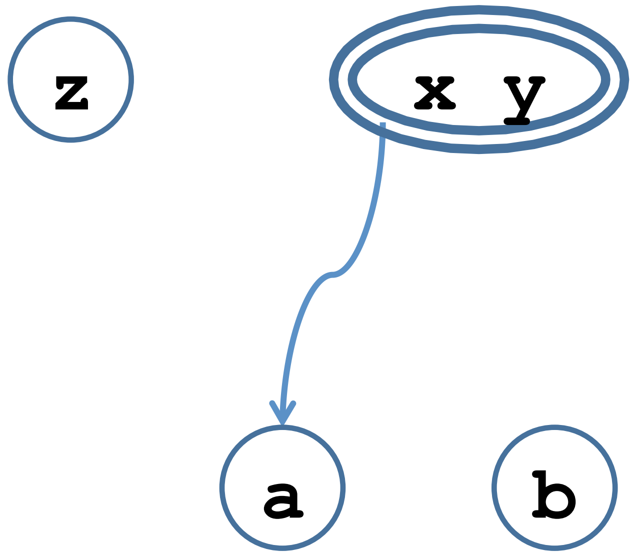
Steensgaard's analysis **x=&a**

```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```



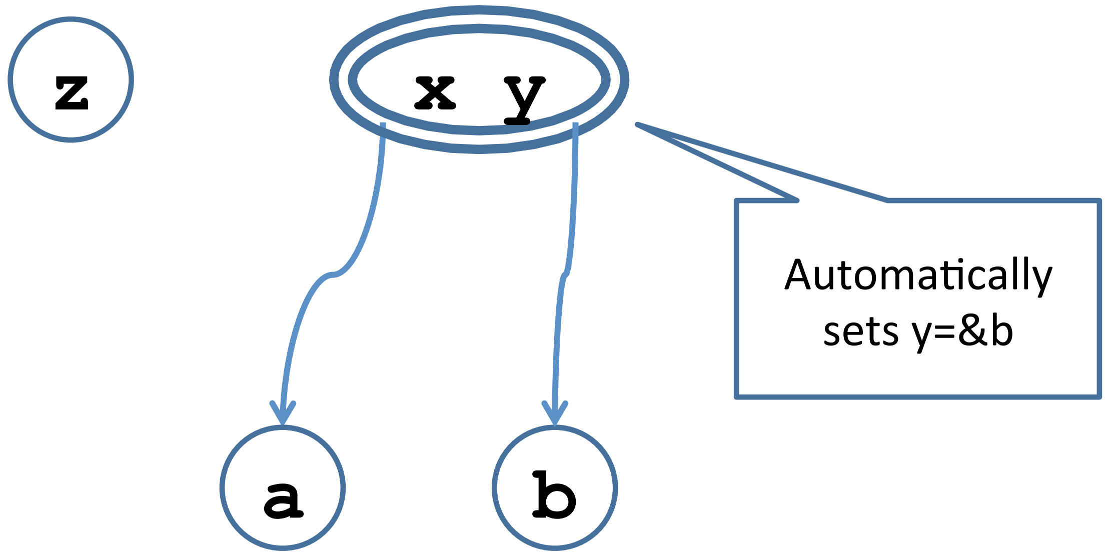
Steensgaard's analysis **y=x**

```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```



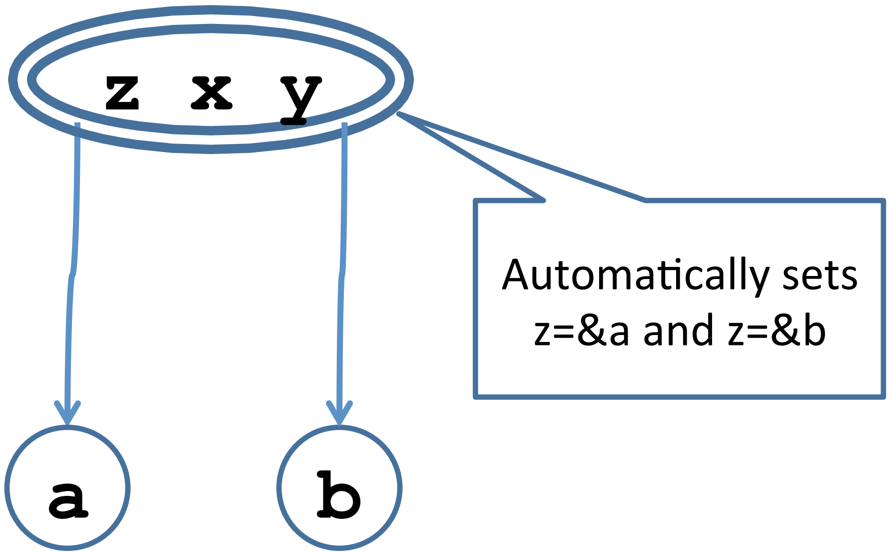
Steensgaard's analysis **x=&b**

```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```



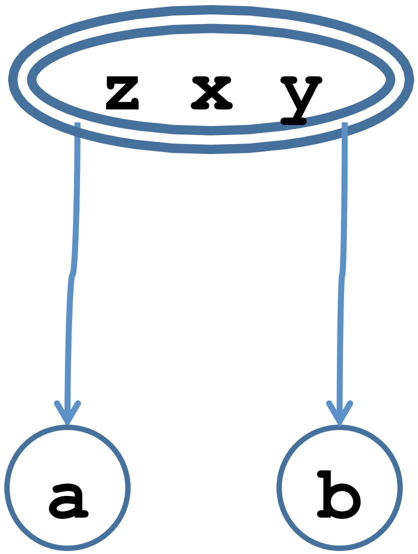
Steensgaard's analysis **z=x**

```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```



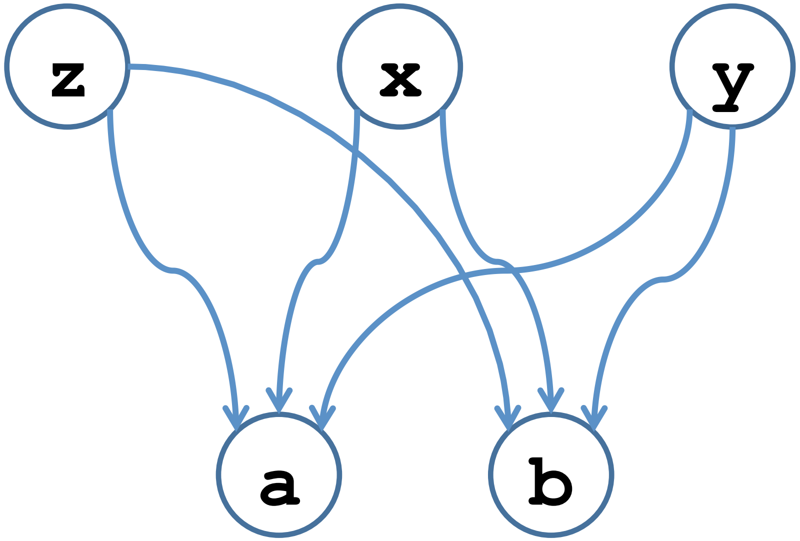
Steensgaard's analysis final result

```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```



Andersen's analysis final result

```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```



Another example

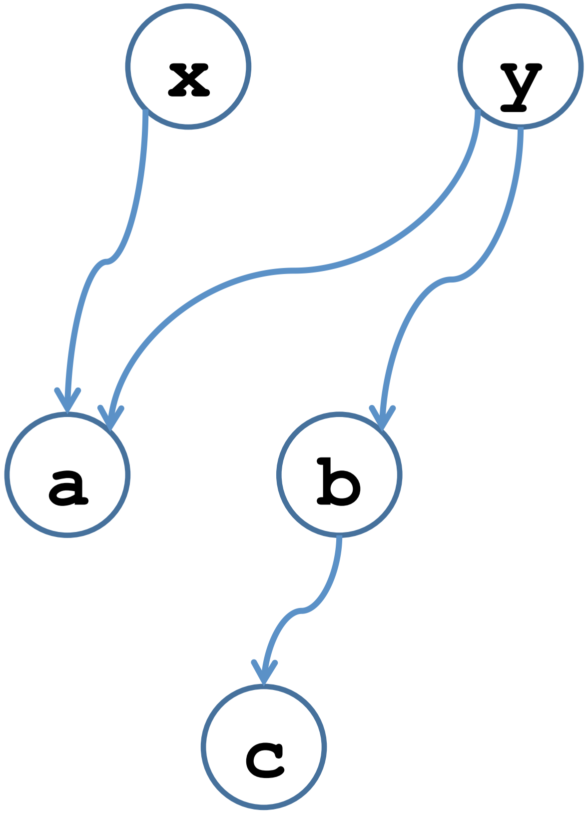
```
L1: x = &a;
L2: y = x;
L3: y = &b;
L4: b = &c;
L5:
```

Andersen's analysis result = ?

```
L1: x = &a;
L2: y = x;
L3: y = &b;
L4: b = &c;
L5:
```

Another example

```
L1: x = &a;
L2: y = x;
L3: y = &b;
L4: b = &c;
L5:
```

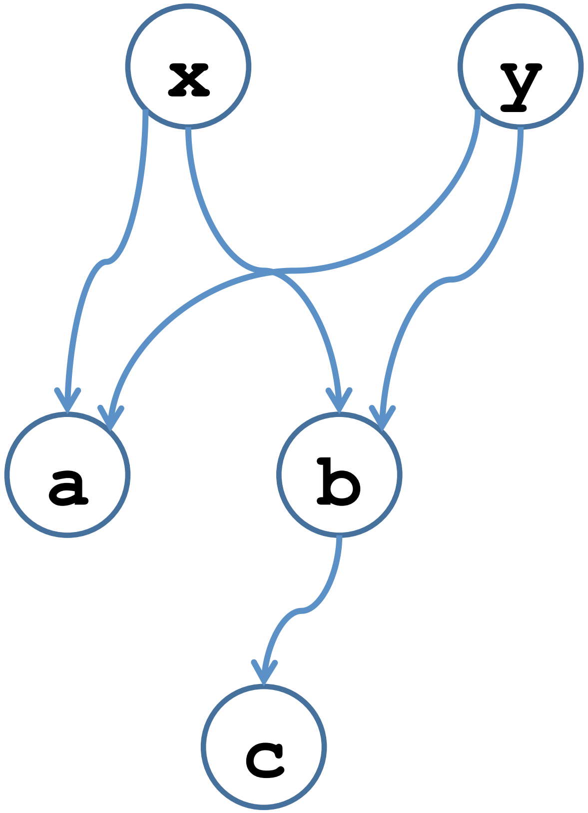


Steensgaard's analysis result = ?

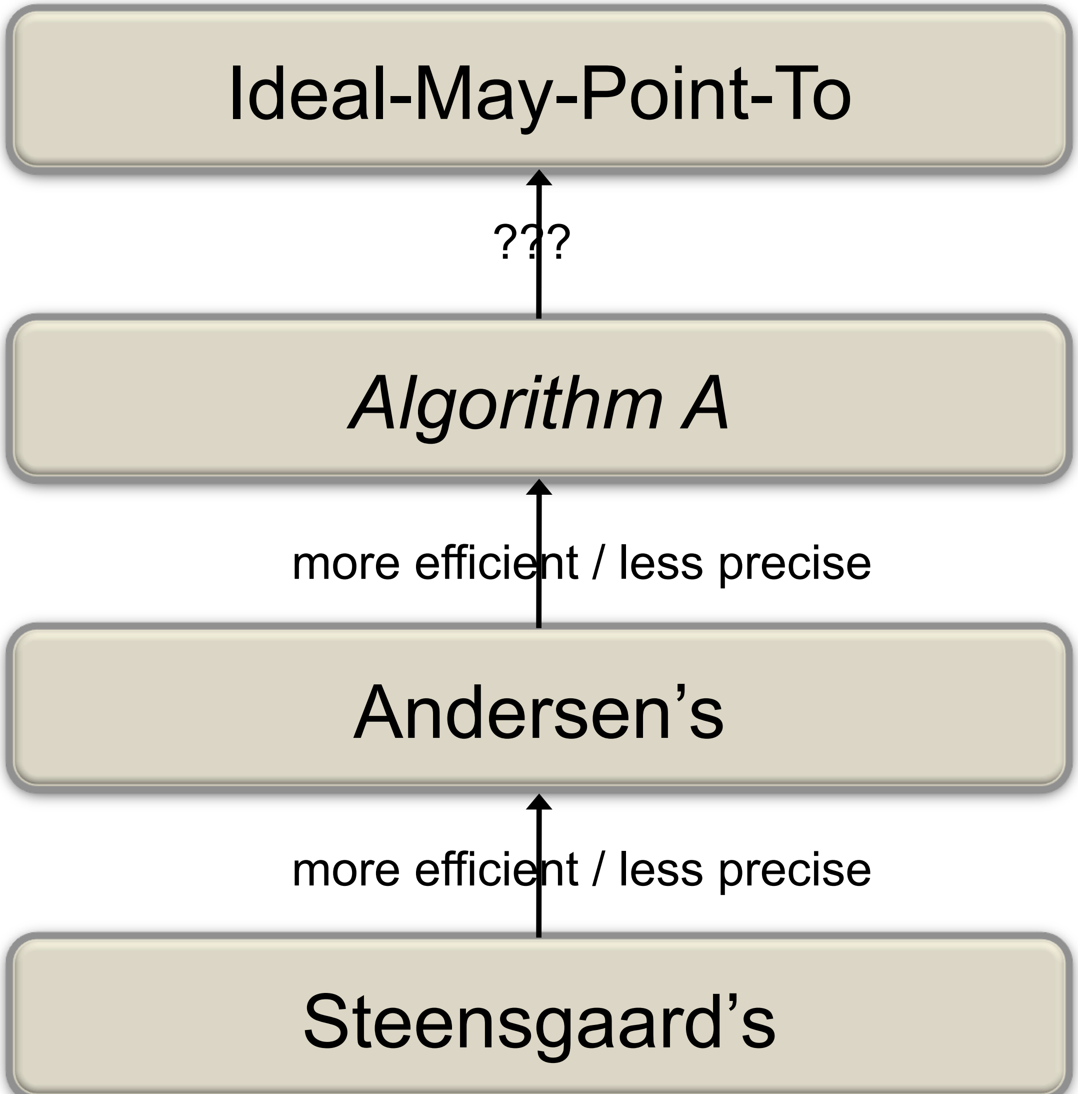
```
L1: x = &a;
L2: y = x;
L3: y = &b;
L4: b = &c;
L5:
```

Steensgaard's analysis result =

```
L1: x = &a;
L2: y = x;
L3: y = &b;
L4: b = &c;
L5:
```



May-points-to analyses



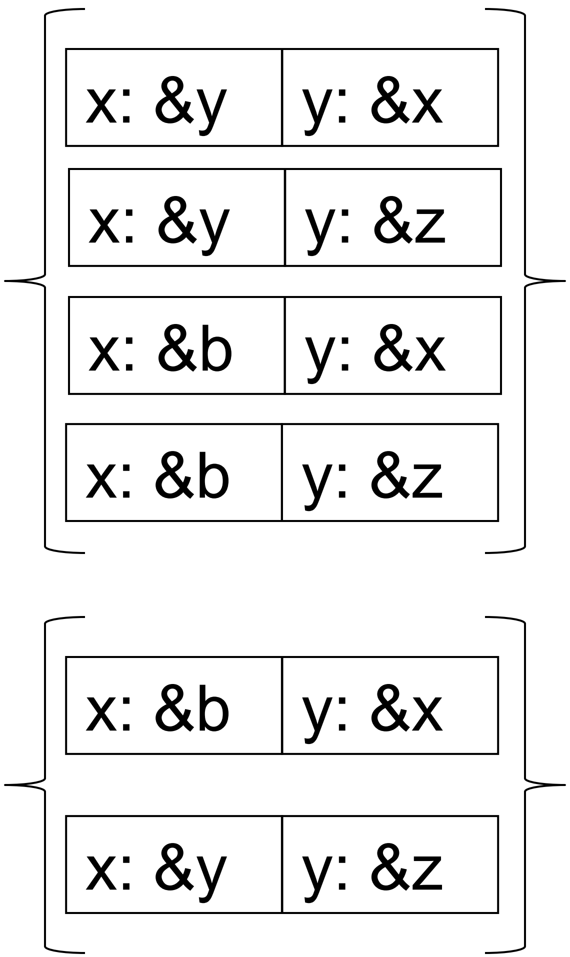
Ideal points-to analysis

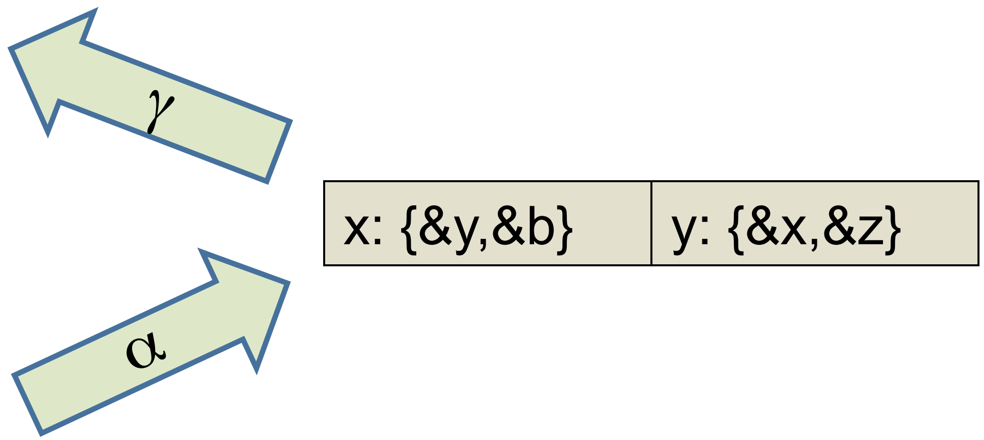
- A sequence of states $s_1 s_2 \ldots s_n$ is said to be an execution (of the program) iff
  - s_1 is the Initial-State
  - $s_i \rightsquigarrow s_{i+1}$ for 1 <= I < n
- A state s is said to be a reachable state iff there exists some execution $s_1 s_2 \ldots s_n$ is such that $s_n = s$.
- CS(u) = { s | (u,s) is reachable }
- IdealMayPT (u) = { (p,x) | $\exists$ s $\in$ CS(u). s(p) = x }
- IdealMustPT (u) = { (p,x) | $\forall$ s $\in$ CS(u). s(p) = x }

Does *Algorithm A* compute the most precise solution?

Ideal vs. *Algorithm A*

- Abstracts away correlations between variables
  - Relational analysis vs.
  - Independent attribute (Cartesian)





Does *Algorithm A* compute the most precise solution?

Is the precise solution computable?

- Claim: The set CS(u) of reachable concrete states (for our language) is computable

- Note: This is true for any collecting semantics with a finite state space

Computing CS(u)

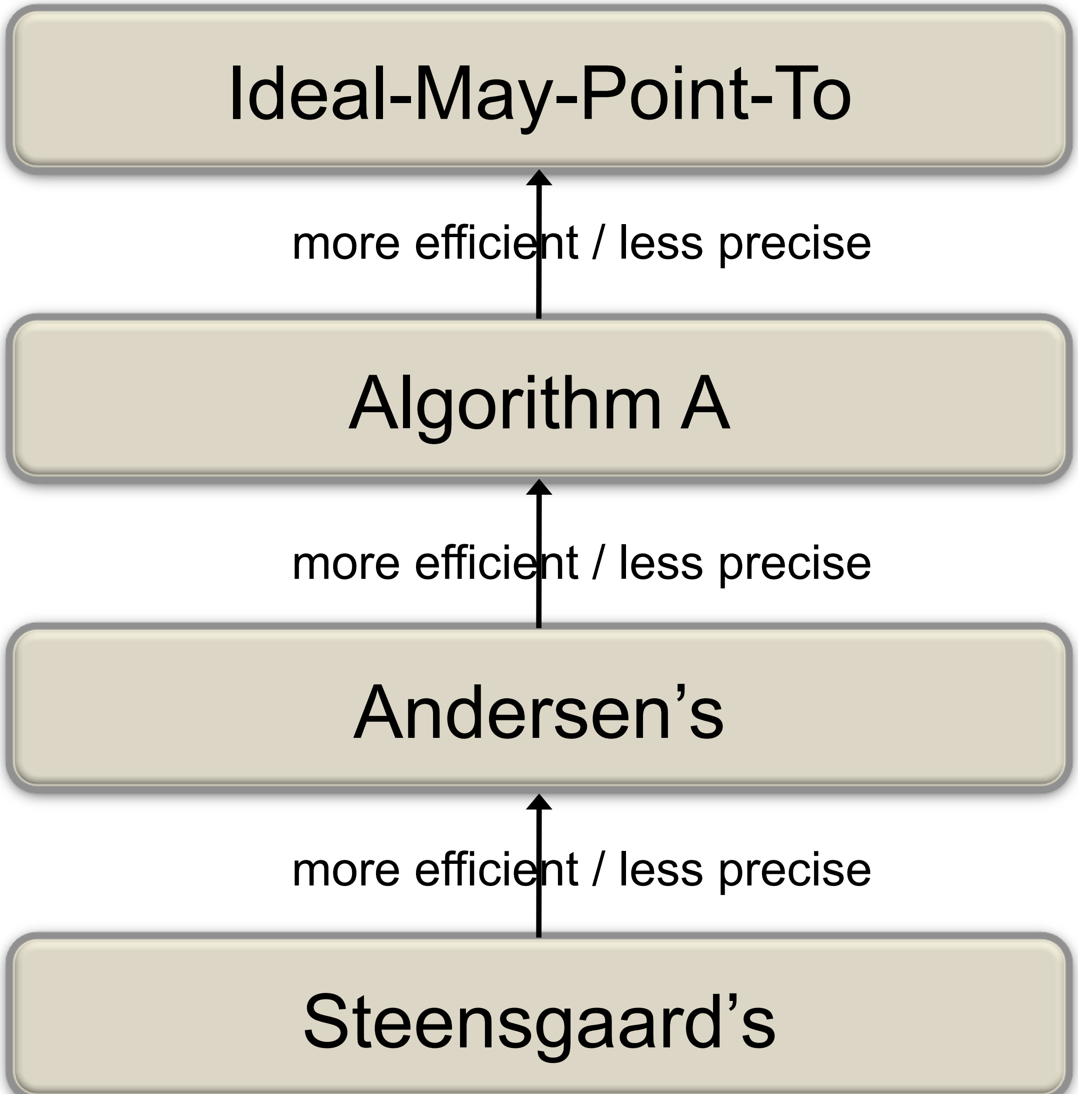
Precise points-to analysis: decidability

- Corollary: Precise may-point-to analysis is computable.
- Corollary: Precise (demand) may-alias analysis is computable.
  - Given ptr-exp1, ptr-exp2, and a program point u, identify if there exists some reachable state at u where ptr-exp1 and ptr-exp2 are aliases.
- Ditto for must-point-to and must-alias
- … for our restricted language!

Precise Points-To Analysis: Computational Complexity

- What's the complexity of the least-fixed point computation using the collecting semantics?
- The worst-case complexity of computing reachable states is exponential in the number of variables.
  - Can we do better?
- Theorem: Computing precise may-point-to is PSPACE-hard even if we have only two-level pointers

May-Point-To Analyses

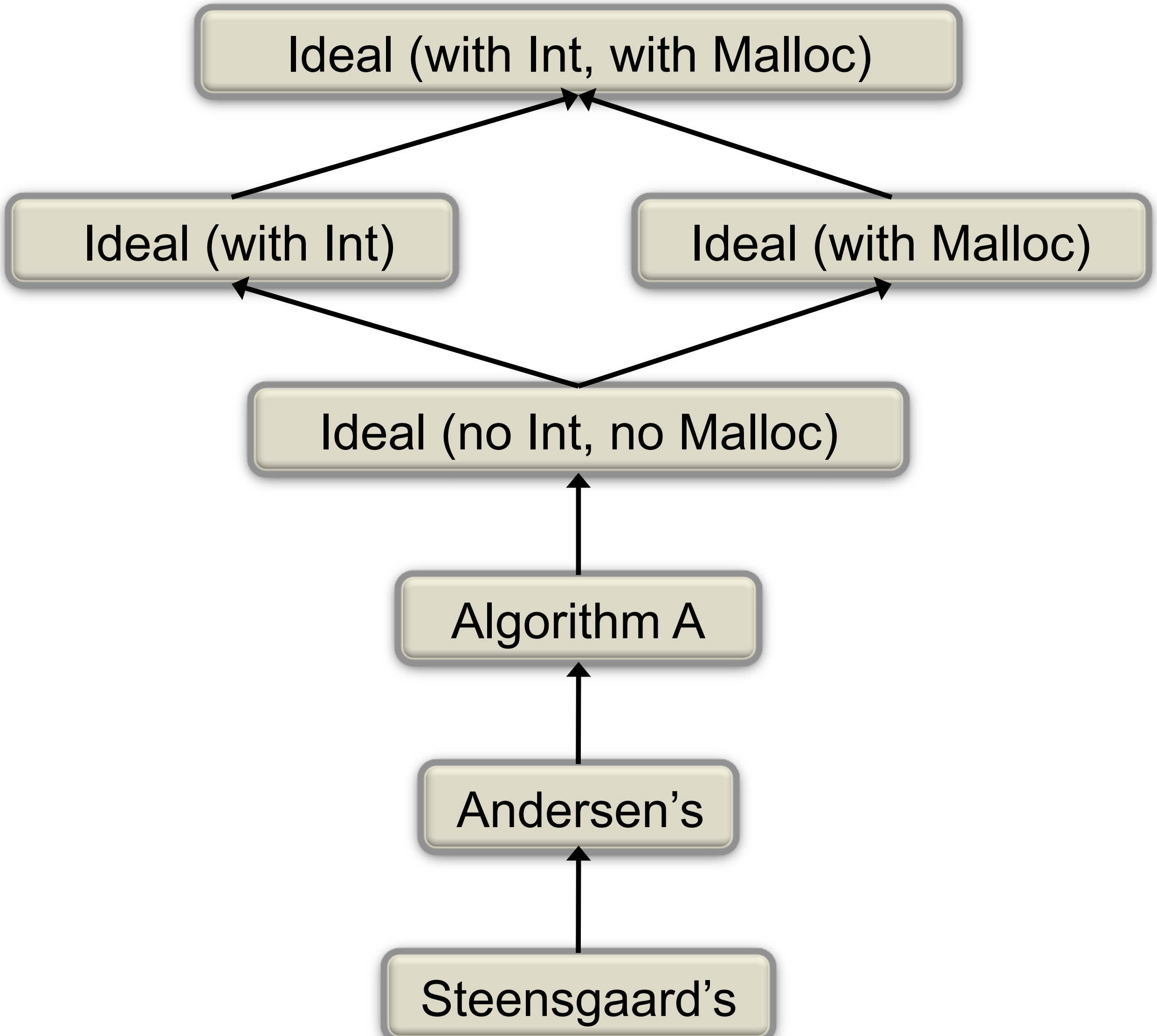


Precise points-to analysis: caveats

- Theorem: Precise may-alias analysis is undecidable in the presence of dynamic memory allocation
  - Add "x = new/malloc ()" to language
  - State-space becomes infinite

- Digression: Integer variables + conditional-branching also makes any precise analysis undecidable

High-level classification



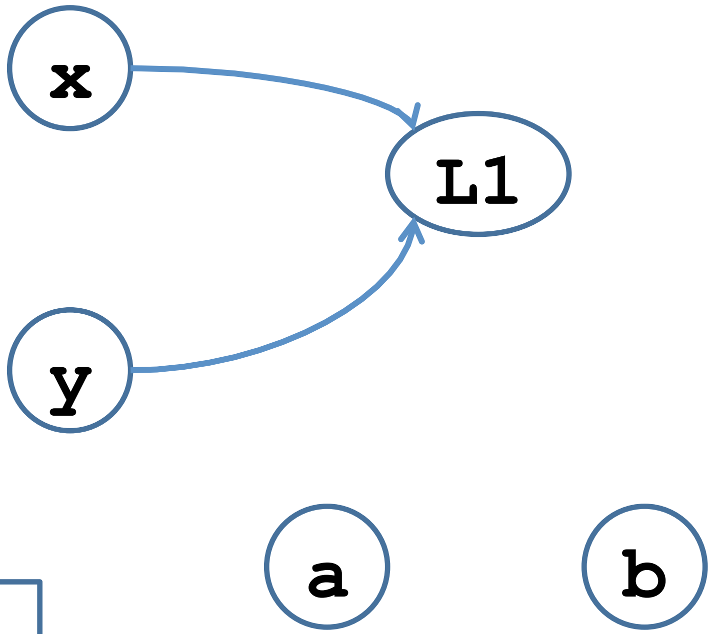
Handling memory allocation

- s: x = new () / malloc ()
- Assume, for now, that allocated object stores one pointer
  - s: x = malloc (sizeof(void*))
- Introduce a pseudo-variable V_s to represent objects allocated at statement s, and use previous algorithm
  - Treat s as if it were “x = &V_s”
  - Also track possible values of V_s
  - Allocation-site based approach
- Key aspect: V_s represents a set of objects (locations), not a single object
  - referred to as a summary object (node)

Dynamic memory allocation example

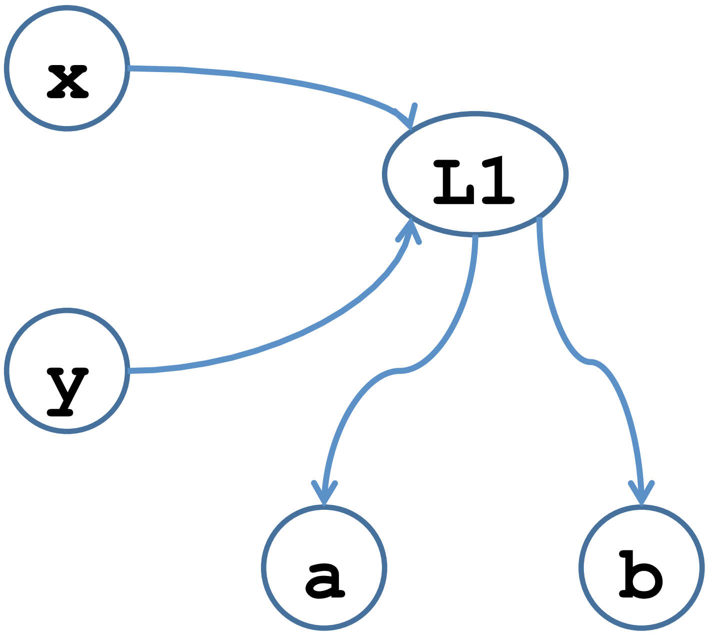
```
L1: x = new O;
L2: y = x;
L3: *y = &b;
L4: *y = &a;
```

How should we handle these statements



Summary object update

```
L1: x = new O;
L2: y = x;
L3: *y = &b;
L4: *y = &a;
```



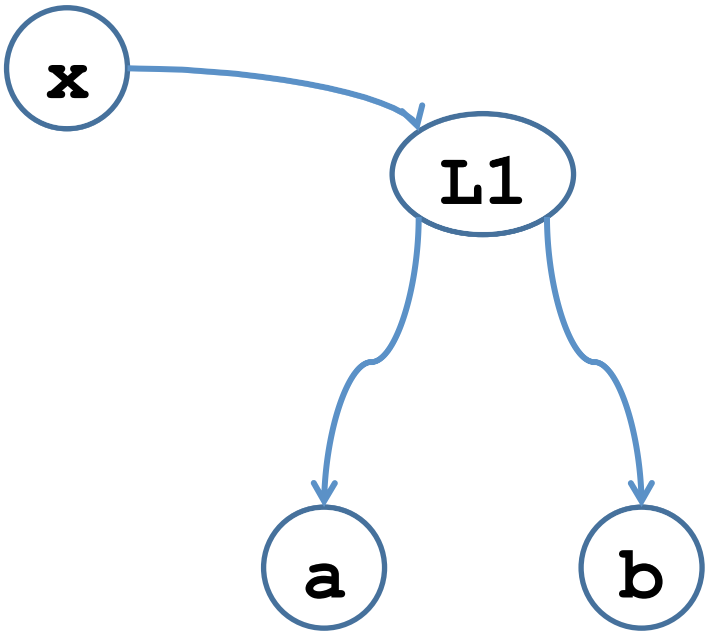
Object fields

- Field-insensitive analysis

```
class Foo {
    A* f;
    B* g;
}
L1: x = new Foo()

x->f = &b;

x->g = &a;
```



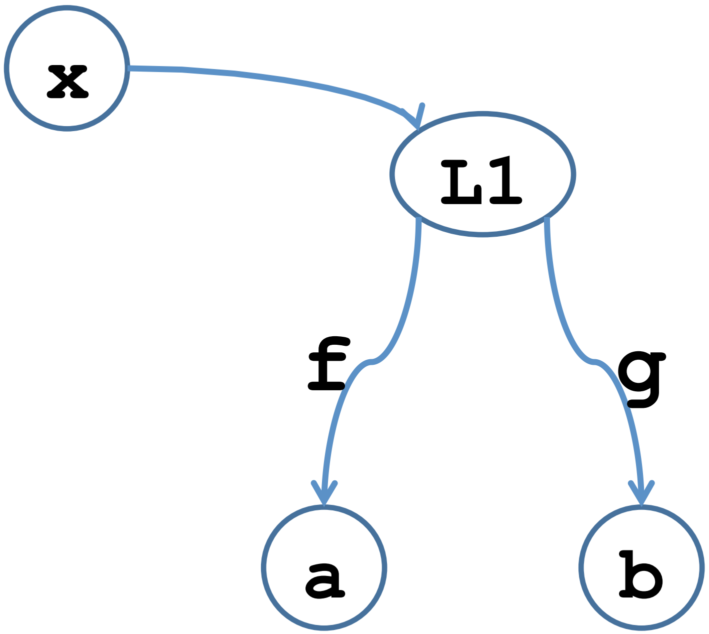
Object fields

- Field-sensitive analysis

```
class Foo {
      A* f;
      B* g;
}
L1: x = new Foo()

x->f = &b;


x->g = &a;
```



Other Aspects

- Context-sensitivity
- Indirect (virtual) function calls and call-graph construction
- Pointer arithmetic
- Object-sensitivity