Pointer Analysis

CSE 501

Spring 15

Course Outline

- Static analysis
  - Dataflow and abstract interpretation
  - Applications ← We are here
- Beyond general-purpose languages
- Program Verification
- Dynamic analysis
- New compilers

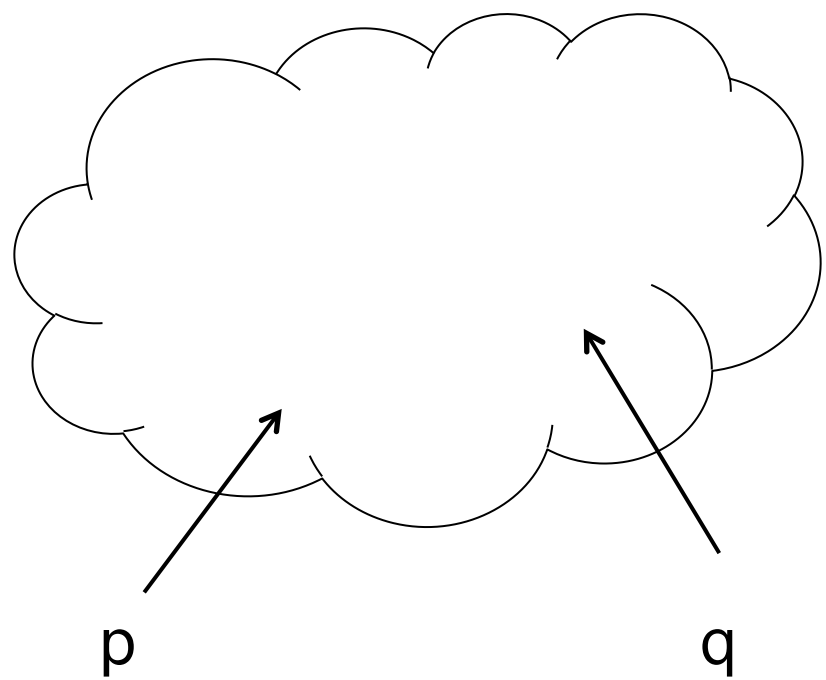
Today

- Intro to pointer analysis
  - What's the big deal?
- Different aspects of the problem
- Two solutions
  - Andersen-style
  - Steensgaard-style

Pointer Analysis

What's the problem?

```
int * p = malloc(…)
int * q = …
…
…
p = q;
p = &q2;
*p = q;
foo(p)
```



Uses

- Alias analysis:
  - For every pair of pointers in the program, determine if they can ever point to the same memory location
- Compiler optimization
  - *p = a + b;
    x = a + b;
  - a + b is not redundant if *p aliases a or b
  - Same for constant propagation, dead code elimination, etc

Uses

- Program parallelization
  - Converting sequential code into parallel implementations automatically
- Shape analysis
  - Find properties of data structures in the heap
- Detecting memory problems
  - Leaks, *NULL, security holes

Why is it hard?

- Complexity: huge in both space and time
  - How many pointers are there in a program?
  - Analyze every program point
  - Need to consider all paths to each program point
- Whole / part of the program?
  - Issues with external libraries
- The problem is undecidable
  [Landi 92, Ramalingam 94]

Designing a pointer analysis

- Must vs may
- Model programs and heap
- Model aggregates
- Analysis sensitivities

Representing points-to information

- Variable pairs that refer to the same memory location
  - <*a, b>, <*c, b>, <*a, *c>
  - *a and b alias, same with *c and b
- Points-to pairs:
  - <a → b>, <c → b>
  - a points to b, and c points to b (hence *a and *c are alias)
- Alias sets:
  - {*a, b, *c}
  - They all point to the same memory location

- Convert from one to another?
  - What are the tradeoffs?

Modeling the heap

- Lump everything into one

- By allocation site
  - Each call to new / malloc is a node
  - Doesn't differentiate between multiple objects allocated by the same site

- Specialized data structures
  - Map of "memory address" to object

Modeling Aggregates

- Arrays
  - Each element is treated as individual location
  - Entire array as a single location
  - First / last element distinct from others

- Classes / Structures
  - Each field is treated as individual location
  - Lump all fields together

Sensitivity

- Flow sensitive

```
x = y
z = x
```

```
z = x
x = y
```

- 1-Context sensitive

```
x = foo(y)
z = foo(q)

foo (x) {
 return x;
}
```

- Path sensitive

```
if (c)
   x = z
else
   x = y
```

```
if (c)
   x = y
else
   x = z
```

- Field sensitive

```
o.f = x
o.f = y
```

```
o.f = x
o.g = y
```

Pointer-induced Aliasing: A Problem Classification [Landi and Ryder, POPL 90]

Alias Mechanism	Intraprocedural May Alias	Intraprocedural Must Alias	Interprocedural May Alias	Interprocedural Must Alias
Reference Formals, No Pointers, No Structures	–	–	Polynomial[1, 5]	Polynomial[1, 5]
Single level pointers, No Reference Formals, No Structures	Polynomial	Polynomial	Polynomial	Polynomial
Single level pointers, Reference Formals, No Pointer Reference Formals, No Structures	–	–	Polynomial	Polynomial
Multiple level pointers, No Reference Formals, No Structures	$\mathcal{NP}$-hard	Complement is $\mathcal{NP}$-hard	$\mathcal{NP}$-hard	Complement is $\mathcal{NP}$-hard
Single level pointers, Pointer Reference Formals, No Structures	–	–	$\mathcal{NP}$-hard	Complement is $\mathcal{NP}$-hard
Single level pointers, Structures, No Reference Formals	$\mathcal{NP}$-hard[14]	Complement is $\mathcal{NP}$-hard	$\mathcal{NP}$-hard[14]	Complement is $\mathcal{NP}$-hard

Table 1: Alias problem decomposition and classification

A Pointer Language

- (Assume x and y are pointers)
- y = &x
  - y points to x
- y = x
  - If x points to z then y points to z
- *y = x
  - If y points to z and z is a pointer, and if x points to w then z now points to w
- y = *x
  - If x points to z and z is a pointer, and if z points to w then y not points to w

A Pointer Language

- points-to(x): set of variables that pointer variable x may point to

- Example: points-to(x) = {y, z}
  - x can point to either y or z

Andersen's-style Pointer Analysis

- Flow, context insensitive, inclusion-based algorithm

Statement	Constraint	Meaning
y = &x	$y \supseteq \{x\}$	$x \in$ points-to(y)
y = x	$y \supseteq x$	points-to(y) $\supseteq$ points-to(x)
y = *x	$y \supseteq *x$	$\forall v \in$ points-to(x). points-to(y) $\supseteq$ points-to(x)
*y = x	$*y \supseteq x$	$\forall v \in$ points-to(y). points-to(v) $\supseteq$ points-to(x)

An Example

p = &a; $p \supseteq \{a\}$
q = p; $q \supseteq p$
p = &b; $p \supseteq \{b\}$
r = p; $r \supseteq p$

Solving the equations:

Points-to	
p	{a, b}
q	{a, b}
r	{a, b}
a	{}
b	{}

Example from Prof. Stephen Chong

Another Example

p = &a;
q = &b;
*p = q;
r = &c;
s = p;
t = *p;
*s = r;

$p \supseteq \{a\}$
$q \supseteq p$
$*p \supseteq q$
$r \supseteq \{c\}$
$s \supseteq p$
$t \supseteq *p$
$*s \supseteq r$

Points-to	
p	{a}
q	{b}
r	{c}
s	{a}
t	{b, c}
a	{b, c}
b	{}
c	{}

Precision

p = &a;

q = &b;

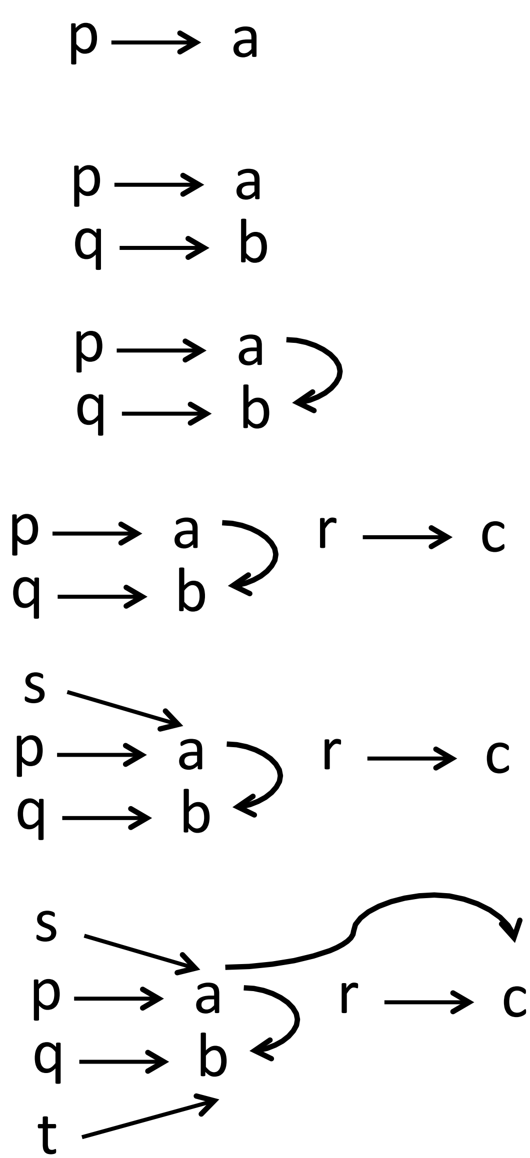
*p = q;

r = &c;

s = p;

t = *p;

*s = r;



Points-to	
p	{a}
q	{b}
r	{c}
s	{a}
t	{b, c}
a	{b, c}
b	{}
c	{}

Precision

p = &a;

q = &b;

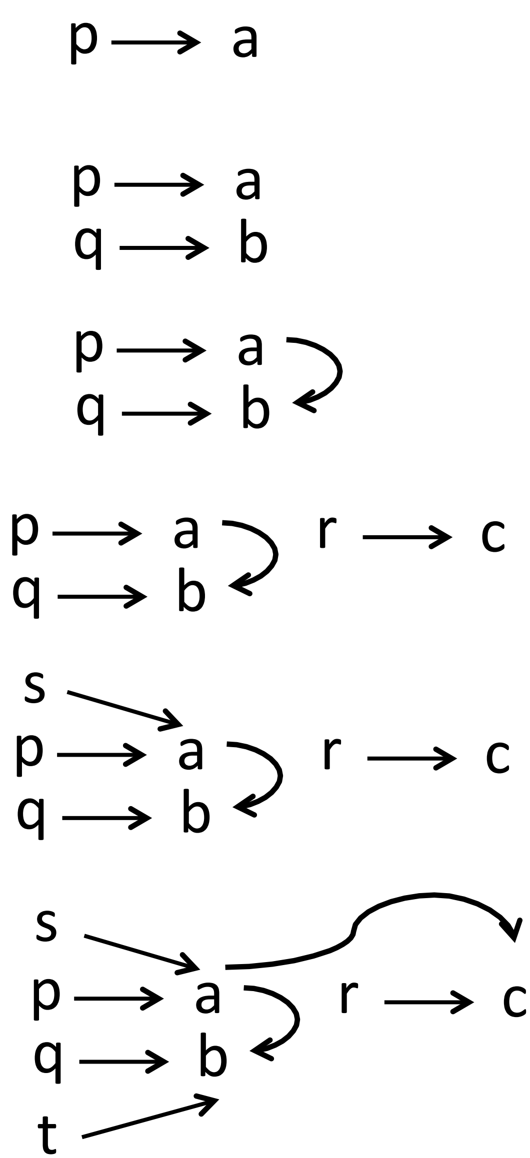
*p = q;

r = &c;

s = p;

t = *p;

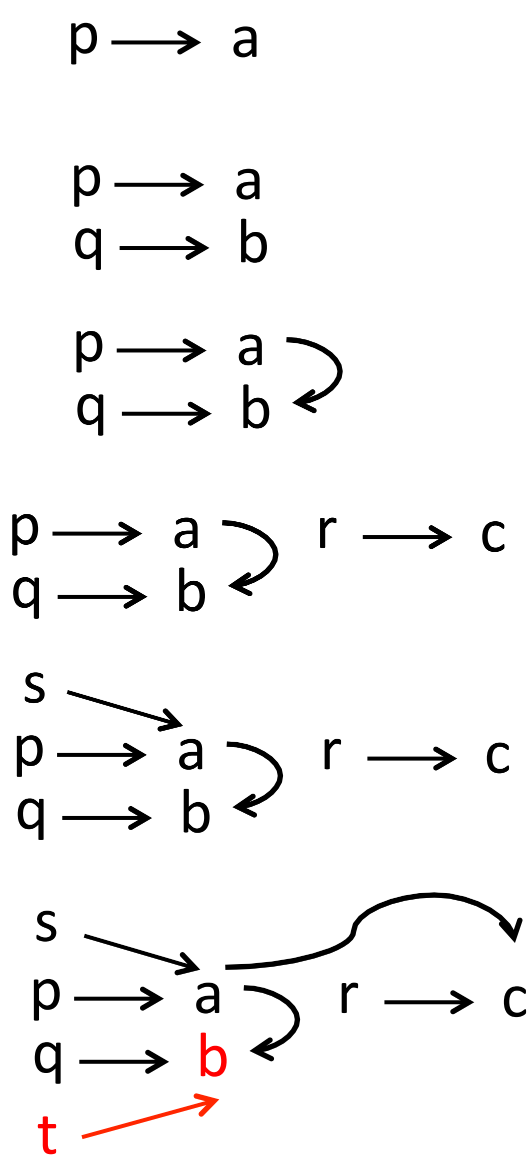
*s = r;



Points-to	
p	{a}
q	{b}
r	{c}
s	{a}
t	{b, c}
a	{b, c}
b	{}
c	{}

Precision

```
p = &a;

q = &b;

*p = q;

r = &c;

s = p;

t = *p;

*s = r;
```



Points-to	
p	{a}
q	{b}
r	{c}
s	{a}
t	{b, c}
a	{b, c}
b	{}
c	{}

Andersen as Graph Closure

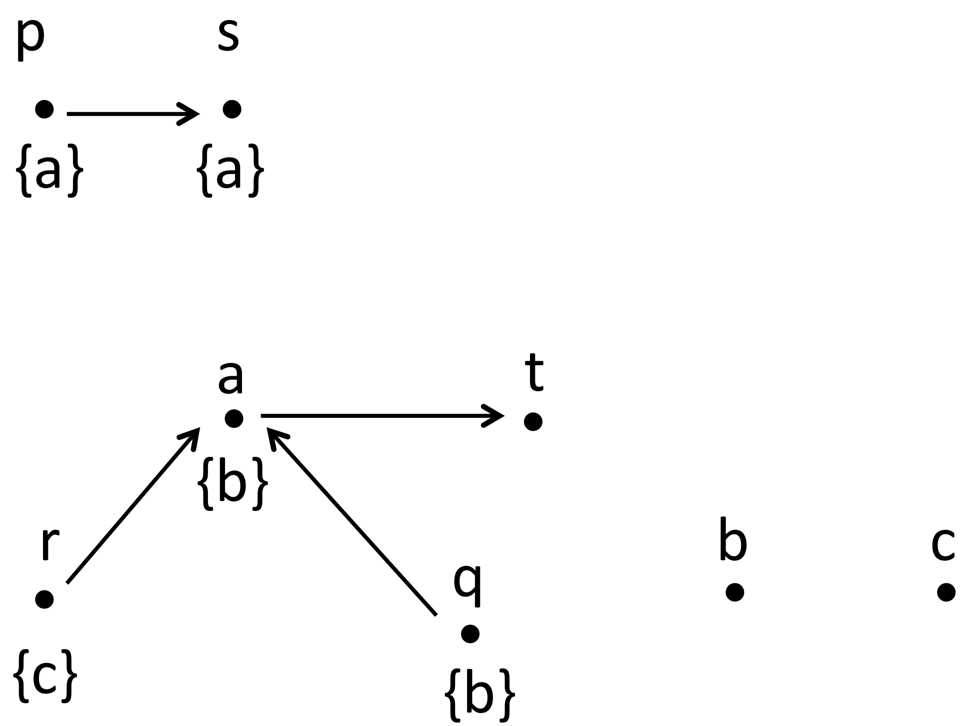
- One node for each memory location
  - i.e., elements in any points-to set
- Each node contains a points-to set
- Solve equations by computing transitive closure of graph, and add edges according to constraints

Andersen as Graph Closure

Statement	Constraint	Meaning	Graph Operation
y = &x	y ⊇ {x}	x∈ points-to(y)	Nothing
y = x	y ⊇ x	points-to(y) ⊇ points-to(x)	Add edge from x to y
y = *x	y ⊇ *x	∀v∈ points-to(x). points-to(y) ⊇ points-to(x)	Nothing
*y = x	*y ⊇ x	∀v∈ points-to(y). points-to(v) ⊇ points-to(x)	Nothing

Same Example, as Graph

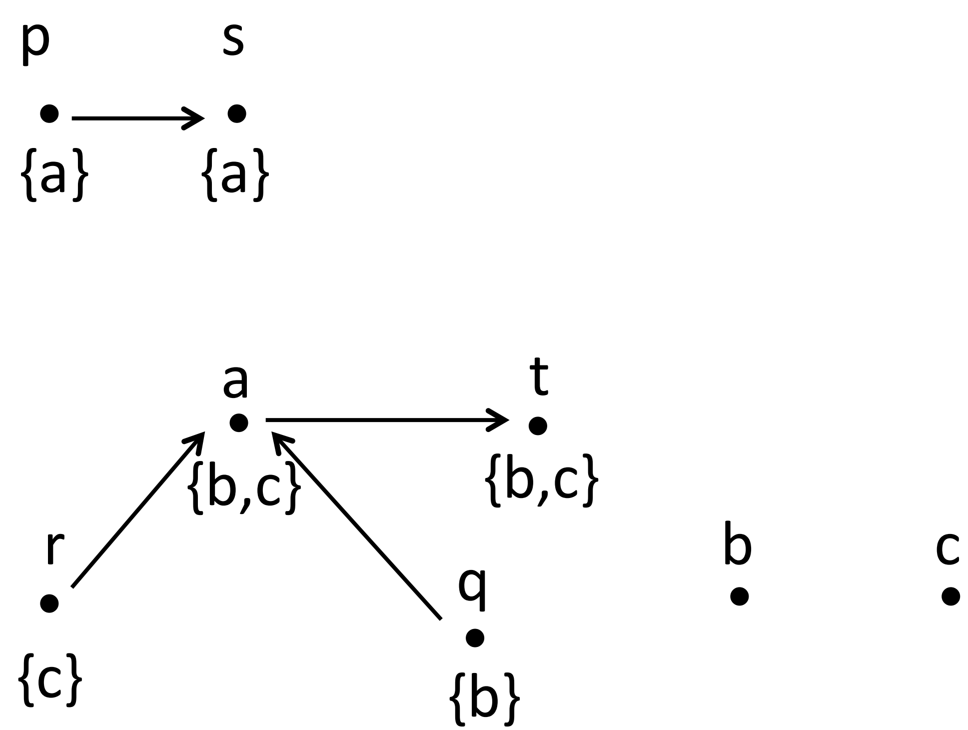
p = &a;	$p \supseteq \{a\}$
q = &b;	$q \supseteq p$
*p = q;	$*p \supseteq q$
r = &c;	$r \supseteq \{c\}$
s = p;	$s \supseteq p$
t = *p;	$t \supseteq *p$
*s = r;	$*s \supseteq r$



y = x	**y ⊇ x**	**points-to(y) ⊇ points-to(x)**	**Add edge from x to y**

Same Example, as Graph

p = &a;	p ⊇ {a}
q = &b;	q ⊇ p
*p = q;	*p ⊇ q
r = &c;	r ⊇ {c}
s = p;	s ⊇ p
t = *p;	t ⊇ *p
*s = r;	*s ⊇ r



y = x	**y ⊇ x**	**points-to(y) ⊇ points-to(x)**	**Add edge from x to y**

Worklist Algorithm

```
// Init graph and points-to sets using base constraints
W = { nodes with non-empty points-to sets }
while W is not empty {
  v = choose from W
  for each constraint v ⊇ x
      add edge x → v, and add x to W if edge is new
  for each a ∈ points-to(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 {
    points-to(q) = points-to(q) U points-to(v),
                        and add q to W if points-to(q) changed
  }
}
```

Worklist Algorithm

- Complexity is $O(n^3)$, where n = number of nodes in graph
- In practice, improve by eliminating cycles
  - Detect strongly connected components in points-to graph and collapse to single node
- How to detect cycles?
  - Some reduction can be done statically, some on-the-fly as new edges added
  - See *The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code,* Hardekopf and Lin, PLDI 2007

Steensgaard-style Analysis

- Similar to Andersen, except that each node can only point to one other node in points-to graph

Steensgaard-style Analysis

- Flow, context insensitive, unification-based algorithm

Statement	**Constraint**	**Meaning**
y = &x	$y \supseteq \{x\}$	$x \in$ points-to(y)
y = x	y = x	points-to(y) = points-to(x)
y = *x	y = *x	$\forall v \in$ points-to(x). points-to(y) = points-to(x)
*y = x	*y = x	$\forall v \in$ points-to(y). points-to(v) = points-to(x)

Steensgaard-style Analysis

- Flow, context insensitive, unification-based algorithm

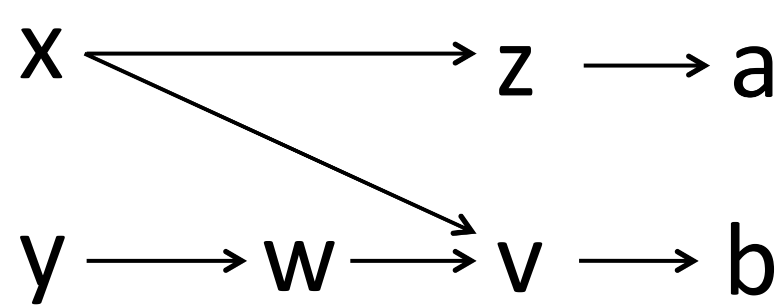
Statement	**Constraint**	**Meaning**
y = &x	$y \supseteq \{x\}$	$x \in$ points-to(y)
y = x	y = x	points-to(y) = points-to(x)
y = *x	y = *x	$\forall v \in$ points-to(x). points-to(y) = points-to(x)
*y = x	*y = x	$\forall v \in$ points-to(y). points-to(v) = points-to(x)

Steensgaard-style Analysis

- Implications for using equality constraints
  - Each statement is processed exactly once
  - Only one iteration of the worklist algorithm
  - Union-find / disjoint set data structure
  - Worst case complexity: O(n) (almost), where n = number of nodes in graph

  - Less precise than Andersen's

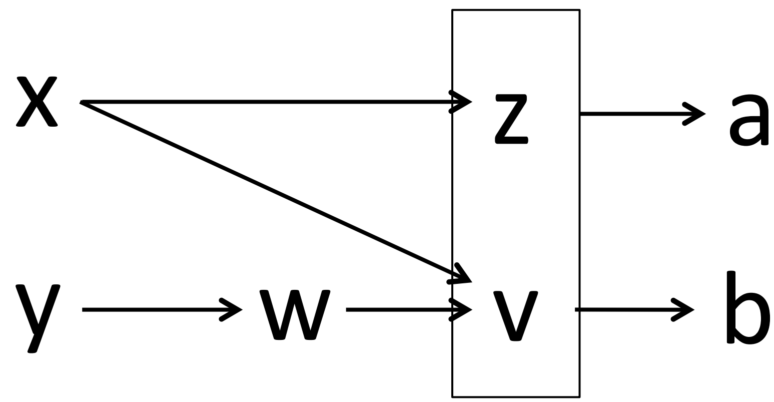
Example



x = *y

Statement	Constraint	Meaning
x = *y	x = *y	$\forall v \in$ points-to(y). points-to(x) = points-to(y)

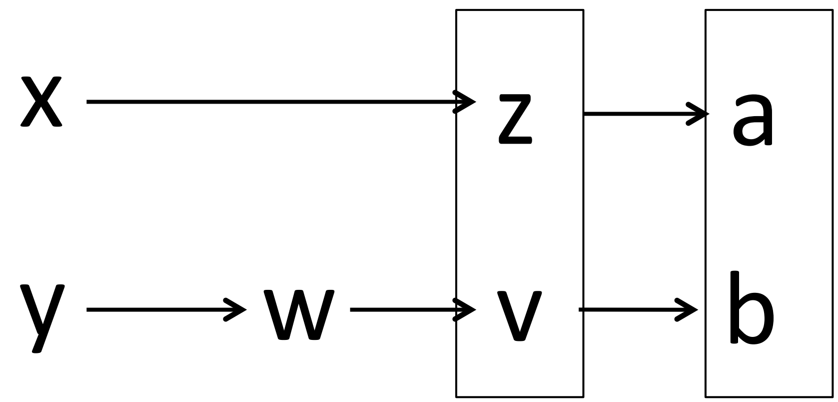
Example



$x = {*}y$

Statement	Constraint	Meaning
x = *y	x = *y	∀v∈ points-to(y). points-to(x) = points-to(y)

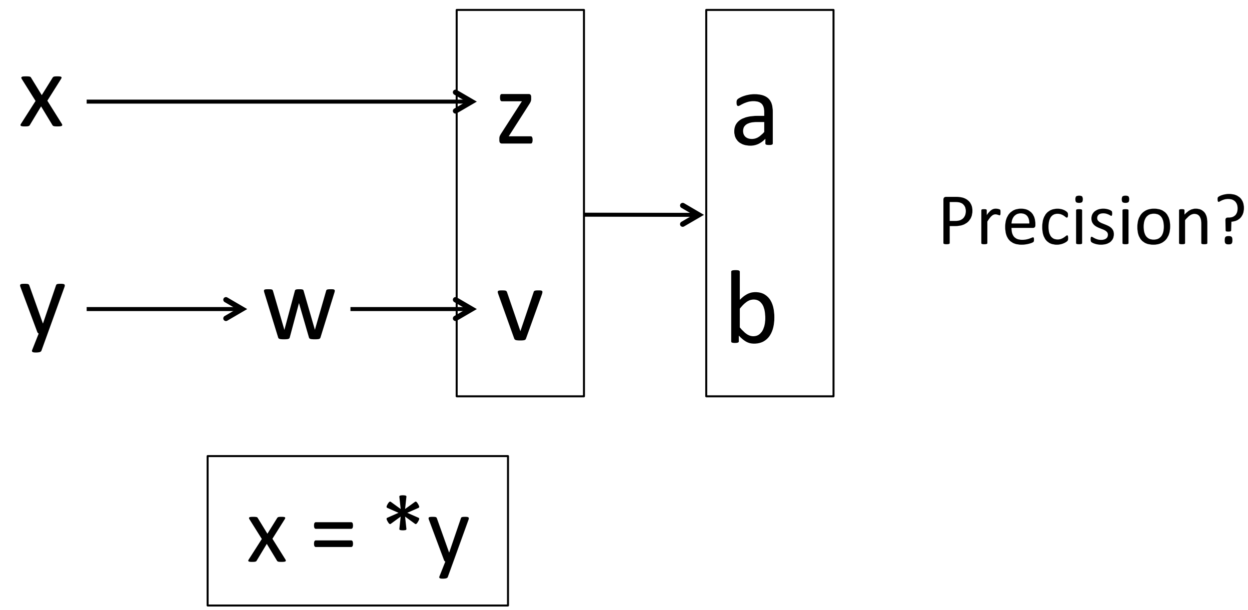
Example



x = *y

Statement	Constraint	Meaning
x = *y	x = *y	∀v∈ points-to(y). points-to(x) = points-to(y)

Example



Statement	Constraint	Meaning
x = *y	x = *y	∀v∈ points-to(y). points-to(x) = points-to(y)