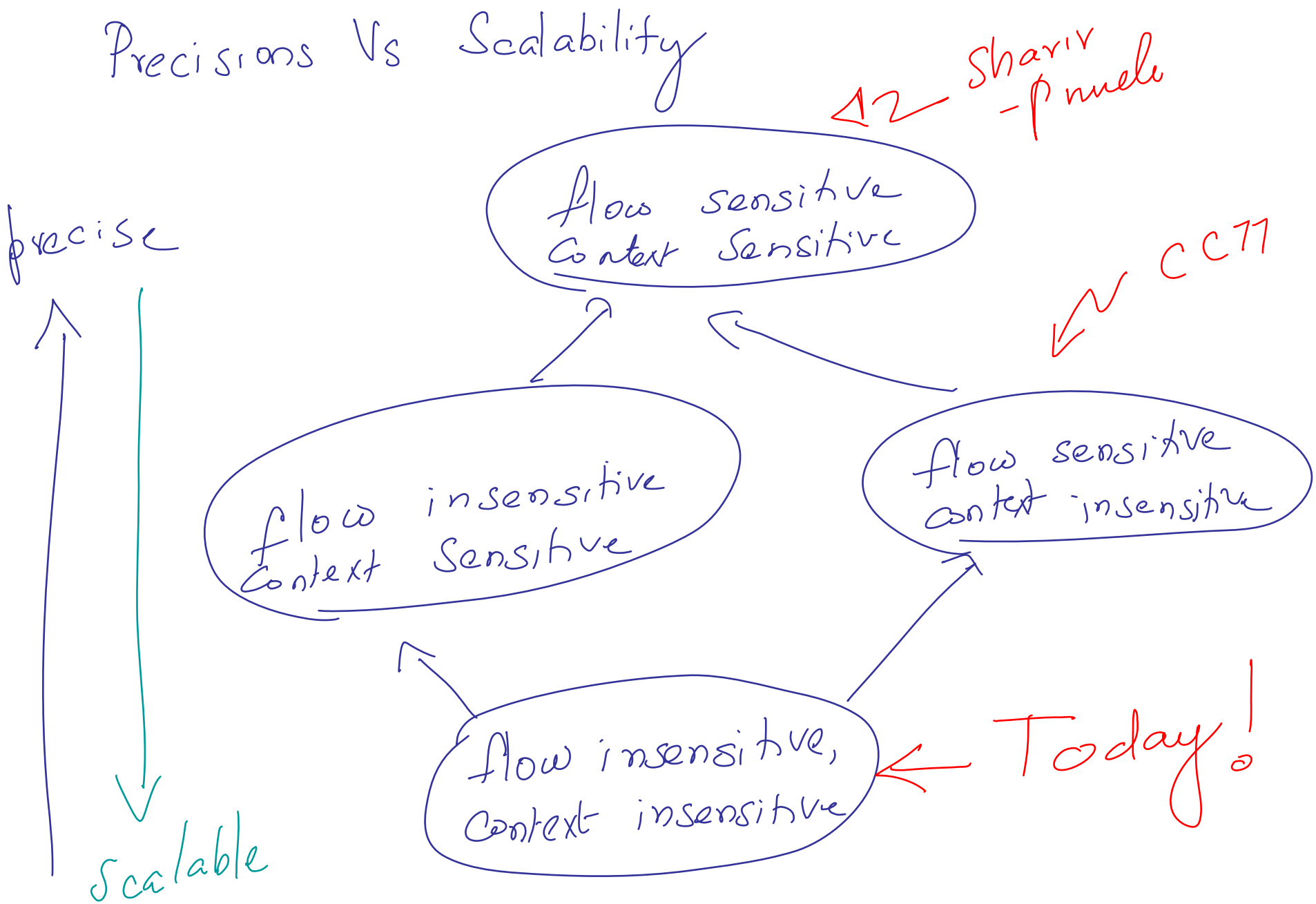
Program analysis & Verification

Lecture 7:
Flow-insensitive analysis

Sriram Rajamani
MSR India

Flow sensitivity: Taking order of statements into account

Context sensitivity: Taking calling contexts into account


Precisions Vs Scalability
Sharir-Pnueli
precise
flow sensitive
Context Sensitive
CCT
flow insensitive
Context Sensitive
flow sensitive
context insensitive
flow insensitive,
Context insensitive
Today!
Scalable

Today:

Flow insensitive, context insensitive analysis

Case study: POINTER analysis

Two major approaches:

UNIFICATION based: Steensgard's analysis

INCLUSION based: Andersen's analysis.

today's topic →

Programming language:

$$
\begin{aligned}
S ::= \quad & x = y \\
| \quad & x = \&y \\
| \quad & x = *y \\
| \quad & *x = y \\
| \quad & x = \text{allocate}(y)
\end{aligned}
$$

Programming language:

$$
\begin{aligned}
S ::= \; & x = y \\
| \; & x = \&y \\
| \; & x = *y \\
| \; & *x = y \\
| \; & x = \text{allocate}(y)
\end{aligned}
$$

Goal:

For each variable infer a "type" that captures all possible "storage shapes" that the variable can point to

Programming language:

$$
\begin{aligned}
S ::= \quad & x = y \\
\mid \quad & x = \&y \\
\mid \quad & x = *y \\
\mid \quad & *x = y \\
\mid \quad & x = \text{allocate}(y)
\end{aligned}
$$

Goal:

For each variable infer a "type" that captures all possible "storage shapes" that the variable can point to

Notes:

(1) No control flow: why?

(2) No operations: will add later

(3) No procedures: will add later

$$S ::= x = y \mid x = \&y \mid x = *y \mid *x = y \mid x = \text{allocate}(y)$$

Types

$$\alpha ::= \bot \mid \text{ref}(\alpha)$$

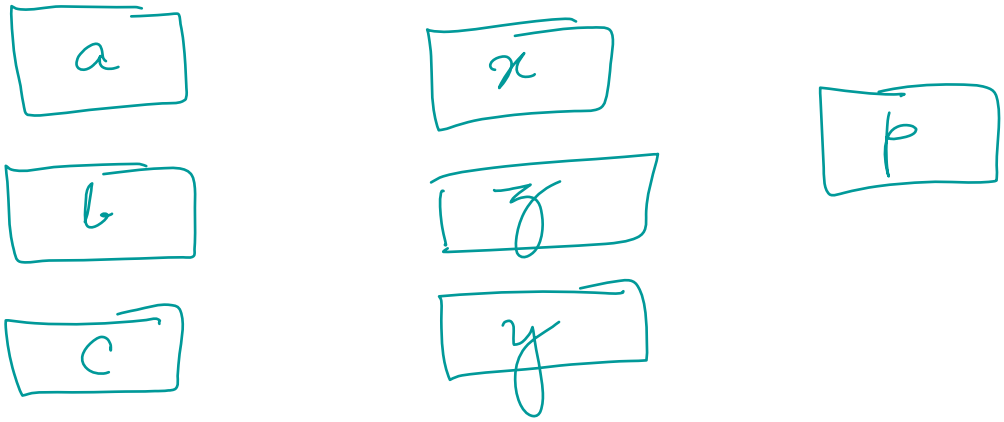
Idea:

(1) For each variable the type "points to" a separate location.

(2) For each assignment "unify" the types on either side of the assignment

```
a = &x;
b = &y;
if (p) then
    y = &z;
else
    y = &x;
fi

c = &y;
```



$a: \alpha_1 = \perp$

$b: \alpha_2 = \perp$

$c: \alpha_3 = \perp$

$x: \alpha_4 = \perp$

$y: \alpha_5 = \perp$

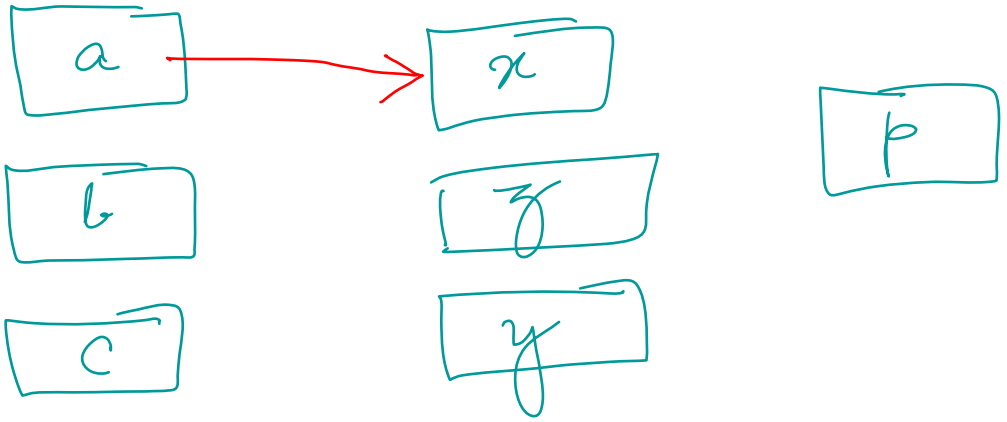
$z: \alpha_6 = \perp$

$p: \alpha_7 = \perp$

Note: for each variable the "box" represents its type

```
a = &x;
b = &y;
if (p) then
    y = &z;
else
    y = &x;
fi

c = &y;
```



$a : \alpha_1 = ref(\alpha_4)$

$b : \alpha_2 = \bot$

$c : \alpha_3 = \bot$

$x : \alpha_4 = \bot$

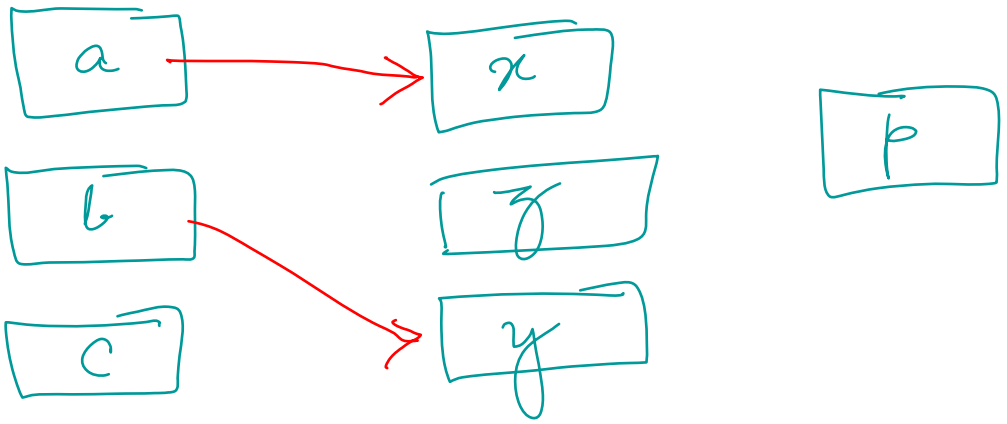
$y : \alpha_5 = \bot$

$z : \alpha_6 = \bot$

$p : \alpha_7 = \bot$

Note: for each variable the "box" represents its type

```
a = &x;
b = &y;
if (p) then
    y = &z;
else
    y = &x;
fi
c = &y;
```



$a : \alpha_1 = ref(\alpha_4)$

$b : \alpha_2 = ref(\alpha_5)$

$c : \alpha_3 = \bot$

$x : \alpha_4 = \bot$

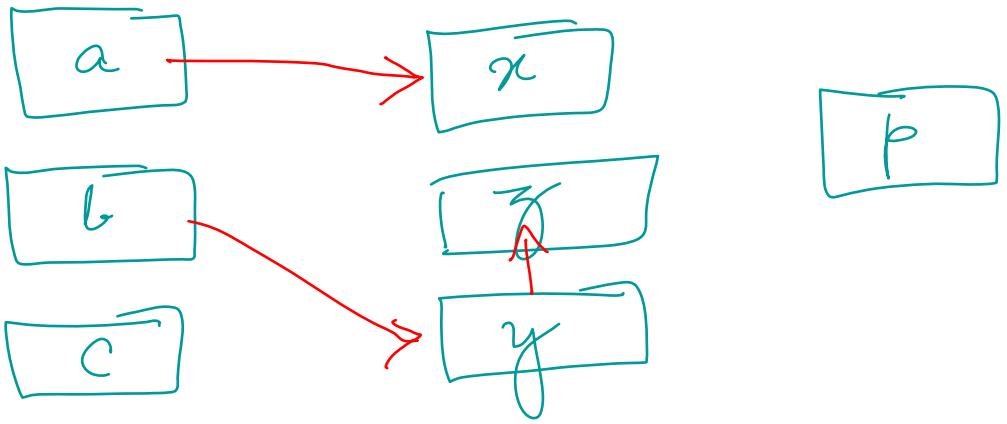
$y : \alpha_5 = \bot$

$z : \alpha_6 = \bot$

$p : \alpha_7 = \bot$

Note: for each variable the "box" represents its type

```
a = &x;
b = &y;
if (p) then
    y = &z;
else
    y = &x;
fi
c = &y;
```



$a : \alpha_1 = ref(\alpha_4)$

$b : \alpha_2 = ref(\alpha_5)$

$c : \alpha_3 = \bot$

$x : \alpha_4 = \bot$

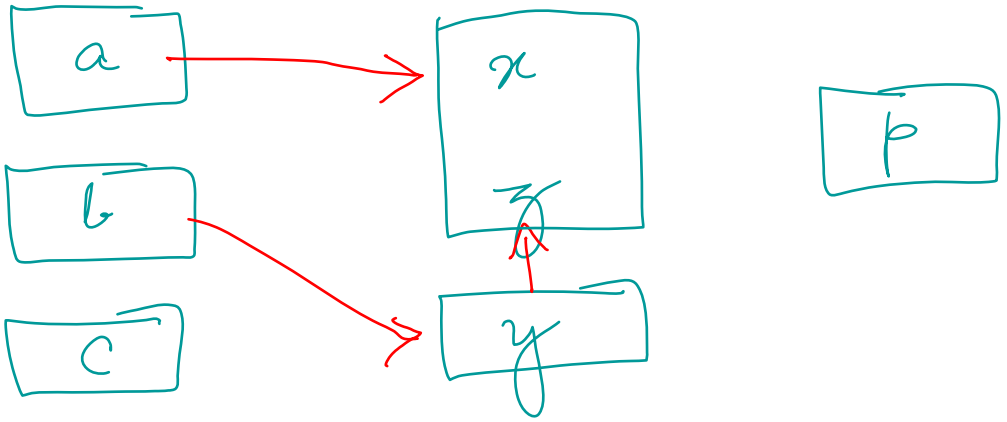
$y : \alpha_5 = ref(\alpha_6)$

$z : \alpha_6 = \bot$

$p : \alpha_7 = \bot$

Note: for each variable the "box" represents its type

```
a = &x;
b = &y;
if (p) then
    y = &z
else
    y = &x;
fi
c = &y;
```



$a : \alpha_1 = ref(\alpha_4)$

$b : \alpha_2 = ref(\alpha_5)$

$c : \alpha_3 = \bot$

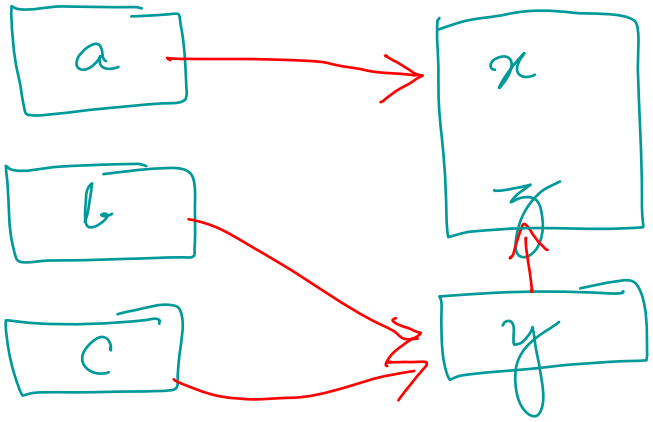
$x, z : \alpha_{4,6} = \bot$

$y : \alpha_5 = ref(\alpha_{4,6})$

$p : \alpha_7 = \bot$

Note: for each variable the "box" represents its type

```
a = &x;
b = &y;
if (p) then
    y = &z
else
    y = &x;
fi
c = &y;
```



$a : \alpha_1 = ref(\alpha_4)$

$b : \alpha_2 = ref(\alpha_5)$

$c : \alpha_3 = ref(\alpha_5)$

$x, z : \alpha_{4,6} = \bot$

$y : \alpha_5 = ref(\alpha_{4,6})$

$p : \alpha_7 = \bot$

Note: for each variable the "box" represents its type

$$S ::= x = y \mid x = \&y \mid x = *y \mid *x = y \mid x = \text{allocate}(y)$$

Types

$$\alpha ::= \bot \mid \text{ref}(\alpha)$$

Idea:

(1) For each variable the type "points to" a separate location.

(2) For each assignment "unify" the types on either side of the assignment

Typing rules

$$\frac{A \vdash x : \alpha \quad A \vdash y : \alpha}{A \vdash WT(x = y)}$$

$$\frac{A \vdash x : ref(\alpha) \quad A \vdash y : \alpha}{A \vdash WT(x = \&y)}$$

$$\frac{A \vdash x : ref(\alpha) \quad A \vdash y : \alpha}{A \vdash WT(*x = y)}$$

$$\frac{A \vdash x : \alpha \quad A \vdash y : ref(\alpha)}{A \vdash WT(x = *y)}$$

$$\frac{A \vdash x : ref(\_)}{A \vdash x = allocate(y)}$$

Note:

Typing rules specify entire algm

1. Precisely!
2. Succinctly!
3. Completely!

Complexity:

$$O(N * \alpha(N, N))$$

where α is the inverse ackerman function

Notes: (1) $\alpha(N, N)$ is the cost of 1 union-find operation

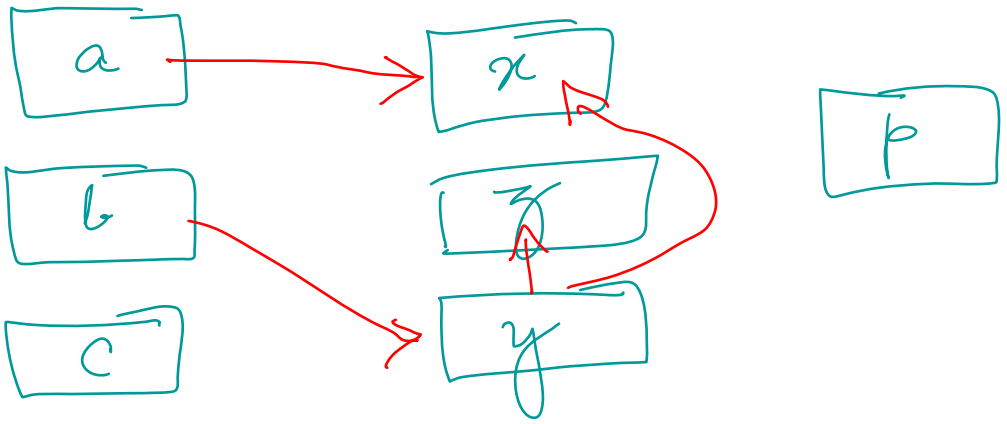
(2) At most N such operations need to be done. (Why?)

Inclusion based analysis

$$\frac{A \vdash x : \alpha \quad A \vdash y : \beta \quad \beta \subseteq \alpha}{A \vdash WT(x = y)}$$

Recall...

```
a = &x;
b = &y;
if (p) then
   y = &z
else
   y = &x;
fi
c = &y;
```



Complexity: ??

"Points-to analysis in almost linear time"
Bjarne Steensgard, POPL 96

← our presentation is a simplified version of this paper.

Two major enhancements in the paper:

E1. Unification is avoided with non-pointer types using more sophisticated typing and "delayed" unification

E2. Procedure calls are handled
- Assignments from: actuals → formals
  return vars → LHS at call

E3. Function pointers are handled
- Function types are used for functions (and they are handled by unification as well)

Recall:

Our typing rule for $x = y$

$$\frac{A \vdash x : \alpha \qquad A \vdash y : \alpha}{A \vdash WT(x = y)}$$

Consider the example:

$a = 4$

$x = a$

$y = a$

Our rule forces x, y to have the same type even if a never holds a pointer

Recall:

Our typing rule for $x = y$

$$\frac{A \vdash x : \alpha \qquad A \vdash y : \alpha}{A \vdash WT(x = y)}$$

Consider the example:

$a = 4$

$x = a$

$y = a$

Our rule forces x, y to have the same type even if a never holds a pointer

Steensgard's rule:

$$\frac{A \vdash x : ref(\alpha_1) \qquad A \vdash y : ref(\alpha_2) \qquad \alpha_2 \trianglelefteq \alpha_1}{A \vdash WT(x = y)}$$

$$t_1 \trianglelefteq t_2 \triangleq (t_1 = \underline{1}) \vee (t_1 = t_2)$$

Note: to implement this rule, one needs "delayed" joins.

```
void * id (void *p) {
    void * retVal;
    retVal = p;
    return (retVal);
}

x = id(&a)
y = id(&b)
```

Q: What is the result of Steensgaard's analysis on this program?

```
void * id (void *p) {
    void * retVal;
    retVal = p;
    return (retVal);
}
```

x = id(&a)

y = id(&b)

Bad flows:

y ← &a

x ← &b

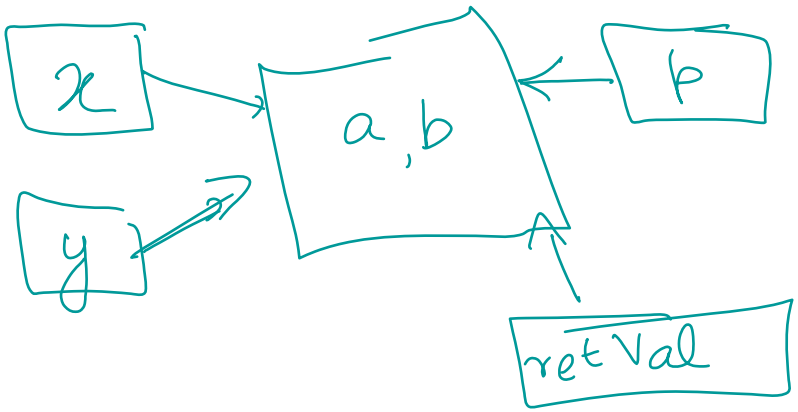
Need context sensitivity to avoid this!

p = &a

p = &b

x = retVal

y = retVal



Assignment questions

1. Consider the following program

$x = fun(a, b) \rightarrow r.$ $\quad r = a$

$y = fun(a, b) \rightarrow r.$ $\quad r = b$

```
z = &x;
z = &y;
```

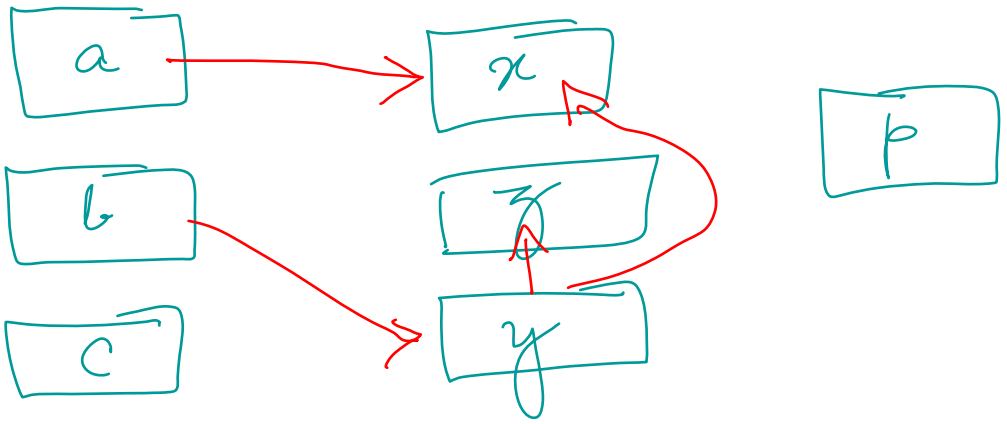
Q1: What are the types inferred for x and y using Steensgaards' analysis?

Q2: What is the complexity of inclusion based pointer analyses?

$$\frac{A \vdash x : \alpha \quad A \vdash y : \beta \quad \beta \subseteq \alpha}{A \vdash WT(x = y)}$$

Recall...

```
a = &x;
b = &y;
if (p) then
   y = &z
else
   y = &x;
fi
c = &y;
```



Complexity: ??