EXERCISE 20

CALL TARGET ANALYSIS REVIEW

Write your name and answer the following on a piece of paper

Draw the call graph of the following program:

```
class SupClass{
public:
    virtual int fun(SupClass * in){
        in->fun();
    }
};

class SubA : public SupClass {
    virtual int fun(SupClass * in){
        in->fun();
    }
};
class SubB : public SupClass {
    virtual int fun(SupClass * in){
        in->fun();
    }
};
int main(){
    SupClass * s = new SubA();
    s->fun();
}
```

EXERCISE 20 SOLUTION

CALL TARGET ANALYSIS REVIEW

ADMINISTRIVIA AND ANNOUNCEMENTS

POINTS-TO ANALYSIS

EECS 677: Software Security Evaluation

Drew Davidson

LAST TIME: CALL TARGET ANALYSIS

REVIEW: LAST LECTURE

DETERMINE WHERE A (POSSIBLE INDIRECT) CALL MIGHT GO

Simplistic

– Class Hierarchy Analysis

More Precise (but incomplete)

– Rapid Type Analysis (RTA and it's elaborations)

Even more precise (but expensive)

- Value Type Analysis (VTA)

```
class SupClass{
public:
    virtual int fun(SupClass * in){
        in->fun();
    }
};

class SubA : public SupClass {
    virtual int fun(SupClass * in){
        in->fun();
    }
};
class SubB : public SupClass {
    virtual int fun(SupClass * in){
        in->fun();
    }
};
int main(){
    SupClass * s = new SubA();
    s->fun();
}
```

LAST TIME: CALL TARGET ANALYSIS

REVIEW: LAST LECTURE

INDIRECT CALLS COME FROM…

Dynamic Dispatch

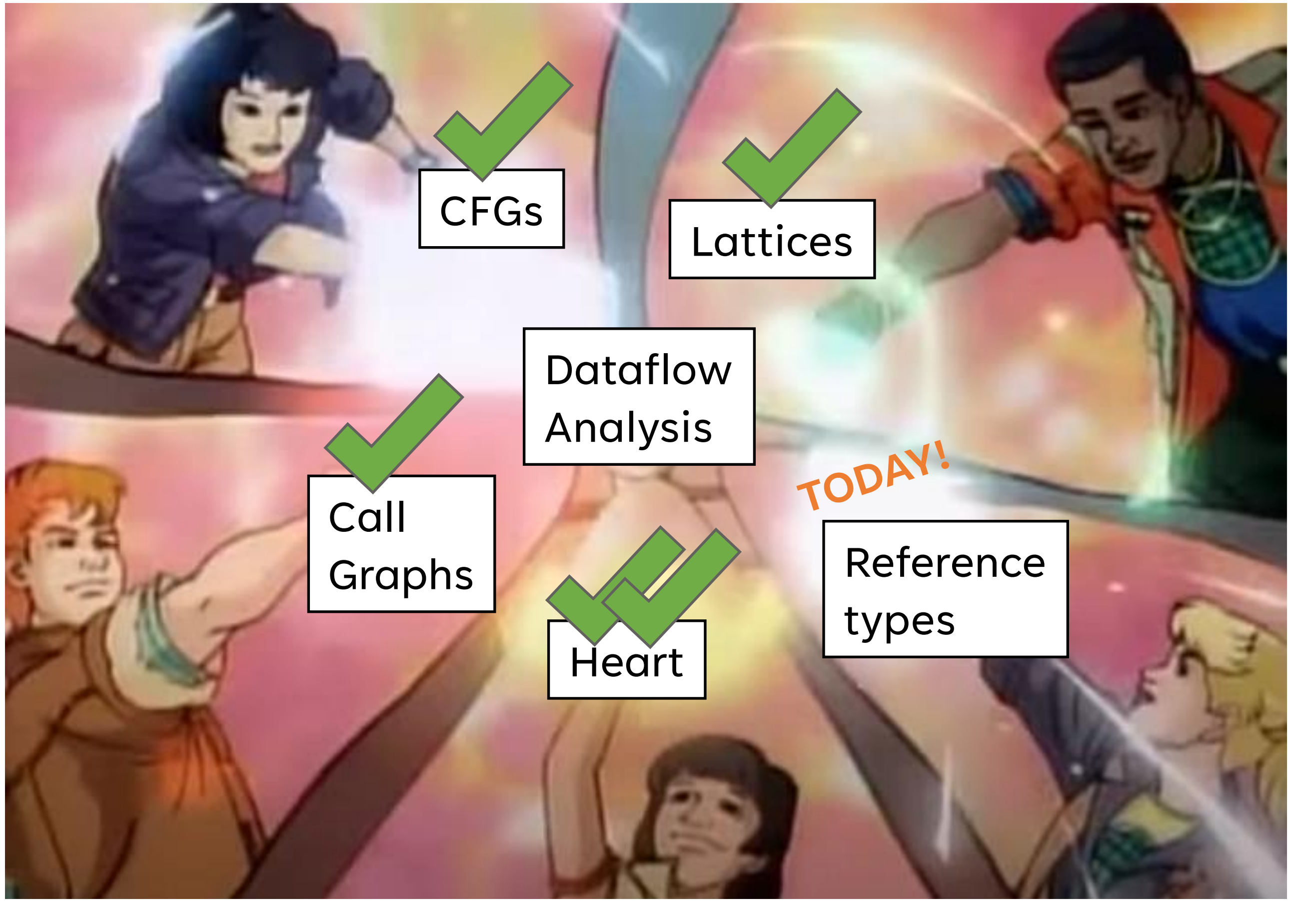
– Virtual functions / Superclass inheritance

First-class functions

– Function pointers

```
int foo(char a) { return 1; }
int bar(char a) { return 2; }

int main(int argc)
{
    int (*fun_ptr)(char) = &foo;

    if (argc == 2){
        fun_ptr = &bar;
    }

    (*fun_ptr)('!');

}
```

CLASS PROGRESS



LECTURE OUTLINE

- Pointers
- Andersen’s Analysis
- Steensgard’s Analysis

REFERENCE TYPES

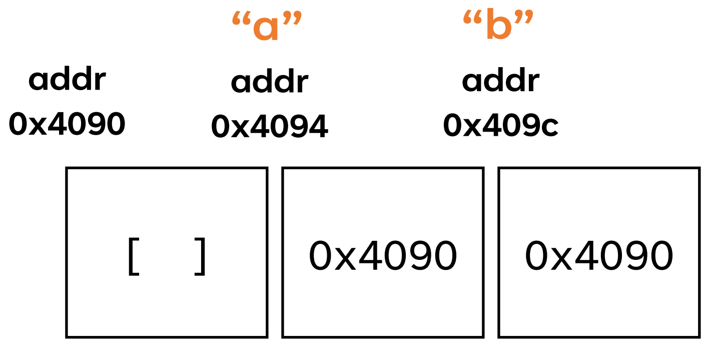
THINKING ABOUT POINTERS

MULTIPLE NAMES BOUND TO THE SAME "LOCATION"

We will sometimes say…

X and Y are "aliases"

X and Y "refer to the same value"



REFERENCE TYPES

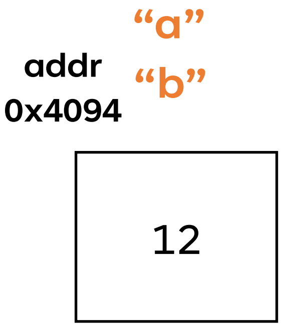
THINKING ABOUT POINTERS

MULTIPLE NAMES BOUND TO THE SAME LOCATION

We will sometimes say…

X and Y are “aliases”

X and Y “refer to the same value”



HUGELY IMPORTANCE FOR DATAFLOW ANALYSIS

Cause a data leak through an alias

Change control flow through an alias

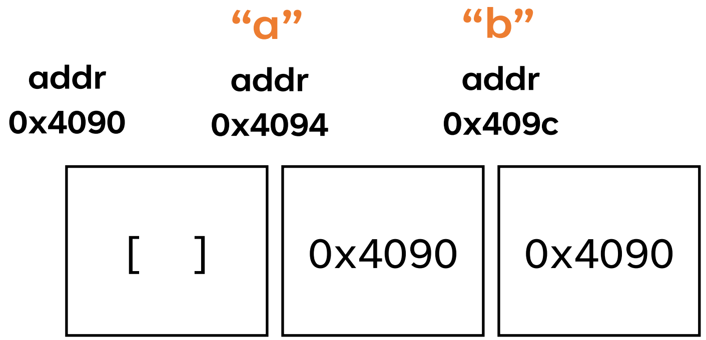
```
a = SOURCE ()
SINK(b)
```

NOT JUST A C LANGUAGE THING!

THINKING ABOUT POINTERS

PYTHON

```
1: a = []
2: b = a
3: b.append(1)
4: print(a)
```

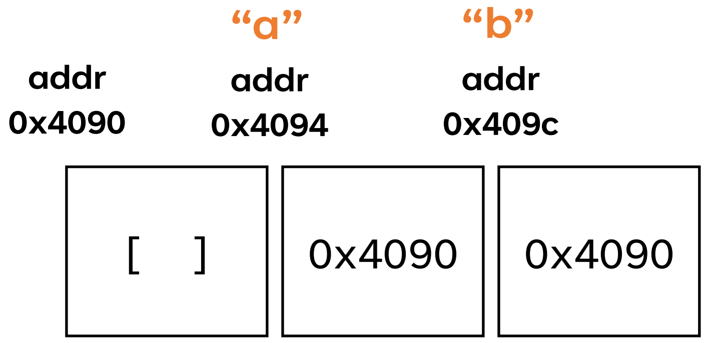


Line 4 prints “[1]”
Even though there is no a.append !

POINTERS: A SPECIAL REFERENCE TYPE

THINKING ABOUT POINTERS

```c
int main(int argc)
{
        int a = 1;
        int * b = &a;
        int * c;
        int * d;
        c = &a;
        *c = 2;
        *d = 3;
}
```



POINTS-TO ANALYSIS

THINKING ABOUT POINTERS

THE FORMAL ANALYSIS TO DETERMINE IF BINDINGS POINT TO THE SAME LOCATION

```
int main(int argc)
{
        int a = 1;
        int * b = &a;
        int * c;
        int * d;
        c = &a;
        *c = 2;
        *d = 3;
}
```

Who points to who?

I know what you're thinking...

"THIS GUY HAS ONE TRICK" – YOU, MAYBE

THINKING ABOUT POINTERS

Another flow-sensitive lattice saturation algorithm?!

No!

MAY-POINT VS MUST-POINT
THINKING ABOUT POINTERS

May-Point(p)

The set of locations to which p **might** refer

Must-Point(p)

The set of locations to which p **must** refer

LECTURE OUTLINE

- May-point v Must-point
- Andersen’s Analysis
- Steensgard’s Analysis

SUBSET CONSTRAINTS

ANDERSEN'S ANALYSIS

A Flow-insensitive Algorithm

Each statement adds a constraint over the points-to sets

End up with a (solvable) system of constraints

Program

```
p = &a;
q = p;
p = &b;
r = p;
```

SUBSET CONSTRAINTS

ANDERSEN'S ANALYSIS

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

SOLVING SUBSET CONSTRAINTS

ANDERSEN'S ANALYSIS

Apply Constraint Rules until Saturation

Each statement adds a constraint over the points-to sets

End up with a (solvable) system of constraints

Program	**Constraints**	**Initial**	**Final**
p = &a;	$p \supseteq \{a\}$	pts(p) = ∅	pts(p) = {a,b}
q = p;	$q \supseteq p$	pts(q) = ∅	pts(q) = {a,b}
p = &b;	$p \supseteq \{b\}$	pts(r) = ∅	pts(r) = {a,b}
r = p;	$r \supseteq p$	pts(a) = ∅	pts(a) = ∅
		pts(b) = ∅	pts(b) = ∅

ANOTHER EXAMPLE

ANDERSEN'S ANALYSIS

A Flow-insensitive Algorithm

Each statement adds a constraint over the points-to sets

End up with a (solvable) system of constraints

Program	**Constraints**	**Initial**	**Final**
p = &a	$p \supseteq \{a\}$	pts(p) = { a }	pts(p) = { a }
q = &b	$q \supseteq \{b\}$	pts(q) = { b }	pts(q) = { b }
*p = q;	$*p \supseteq q$	pts(r) = { c }	pts(r) = { c }
r = &c;	$r \supseteq \{c\}$	pts(s) = ∅	pts(s) = { a }
s = p;	$s \supseteq p$	pts(t) = ∅	pts(t) = { b, c }
t = *p;	$t \supseteq *p$	pts(a) = ∅	pts(a) = { b, c }
*s = r;	$*s \supseteq r$	pts(b) = ∅	pts(b) = ∅
		pts(c) = ∅	pts(c) = ∅

SOLVING CONSTRAINTS AS REACHABILITY

ANDERSEN'S ANALYSIS

Graph closure on the subset relation

Assgmt.	Constraint	Meaning	Edge
a = &b	$a \supseteq \{b\}$	$b \in pts(a)$	no edge
a = b	$a \supseteq b$	$pts(a) \supseteq pts(b)$	$b \rightarrow a$
a = *b	$a \supseteq *b$	$\forall v \in pts(b).\ pts(a) \supseteq pts(v)$	no edge
*a = b	$*a \supseteq b$	$\forall v \in pts(a).\ pts(v) \supseteq pts(b)$	no edge

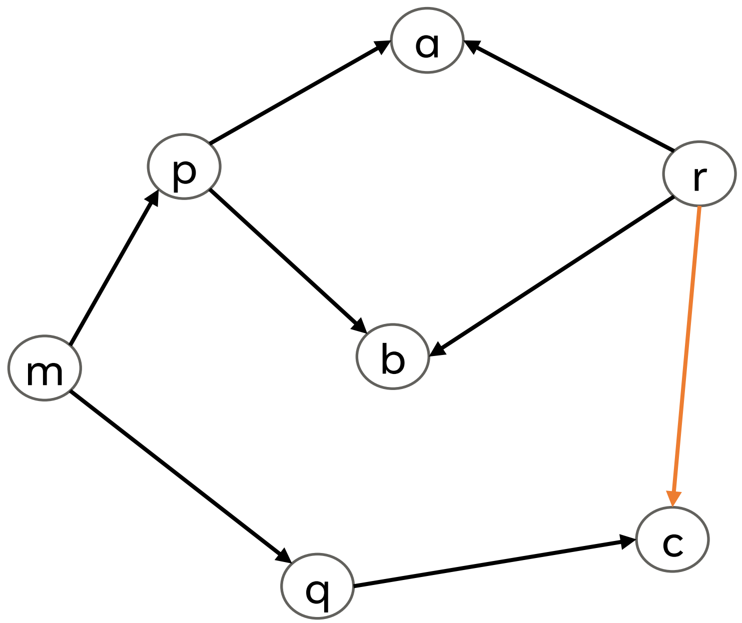
ANDERSEN'S ALGORITHM: REACHABILITY

REVIEW: LAST LECTURE

REACHABILITY FORMULATION

Step 1: List pointer-related operations
Step 2: Saturate points-to graph
Step 3: Compute node reachability

Program	Constraints
p = &a	p ⊇ {a}
p = &b	p ⊇ {b}
m =&p;	m ⊇ {p}
r = *m;	r ⊇ *m
q = &c;	q ⊇ {c}
m = &q	m ⊇ {q}



Assignment	Constraint	Meaning
a = &b	a ⊇ {b}	loc(b) ∈ pts(a)
a = b	a ⊇ b	pts(a) ⊇ pts(b)
a = *b	a ⊇ *b	∀v∈pts(b). pts(a) ⊇ pts(v)
*a = b	*a ⊇ b	∀v∈pts(a). pts(v) ⊇ pts(b)

Initial	Final
pts(a) = { }	pts(a) = { }
pts(b) = { }	pts(b) = { }
pts(m) = { }	pts(m) = { p, q }
pts(p) = { }	pts(p) = { a, b }
pts(q) = { }	pts(q) = { c }
pts(r) = { }	pts(r) = { a, b, c }

OVERHEAD

ANDERSEN’S ANALYSIS

Worst Case: Cubic Time

That’s not great!

Optimization:

Cycle elimination

Detect and collapse SCCs in the points-to relation

LECTURE OUTLINE

- May-point v Must-point
- Andersen’s Analysis
- Steensgard’s Analysis

AN ALTERNATIVE APPROACH

STEENSGARD'S ANALYSIS

Aim for Near-Linear-time points-to Analysis

Going to require us to reduce our search-space somewhat

Intuition: Equality constraints

Do away with the notion of subsets

STEENGARD’S ALGORITHM

AN EFFICIENT OVER-APPROXIMATION

IN PRACTICE

Step 1

List pointer-related operations

Step 2

Induce set of ~~subset~~ equality constraints

Step 3

Solve system of constraints

REACHABILITY FORMULATION

Step 1

List pointer-related operations

Step 2

Saturate 1-out points-to graph

Step 3

Compute node reachability

Andersen’s

Assignment	Constraint	Meaning
a = &b	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
a = b	$a \supseteq b$	$pts(a) \supseteq pts(b)$
a = *b	$a \supseteq *b$	$\forall v \in pts(b).\ pts(a) \supseteq pts(v)$
*a = b	$*a \supseteq b$	$\forall v \in pts(a).\ pts(v) \supseteq pts(b)$

Steengaard’s

Assignment	Constraint	Meaning
a = &b	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
a = b	$a = b$	$pts(a) = pts(b)$
a = *b	$a = *b$	$\forall v \in pts(b).\ pts(a) = pts(v)$
*a = b	$*a = b$	$\forall v \in pts(a).\ pts(v) = pts(b)$

EQUALITY CONSTRAINTS

STEENSGARD’S ANALYSIS

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

EQUALITY CONSTRAINTS

STEENSGARD'S ANALYSIS

p = &a

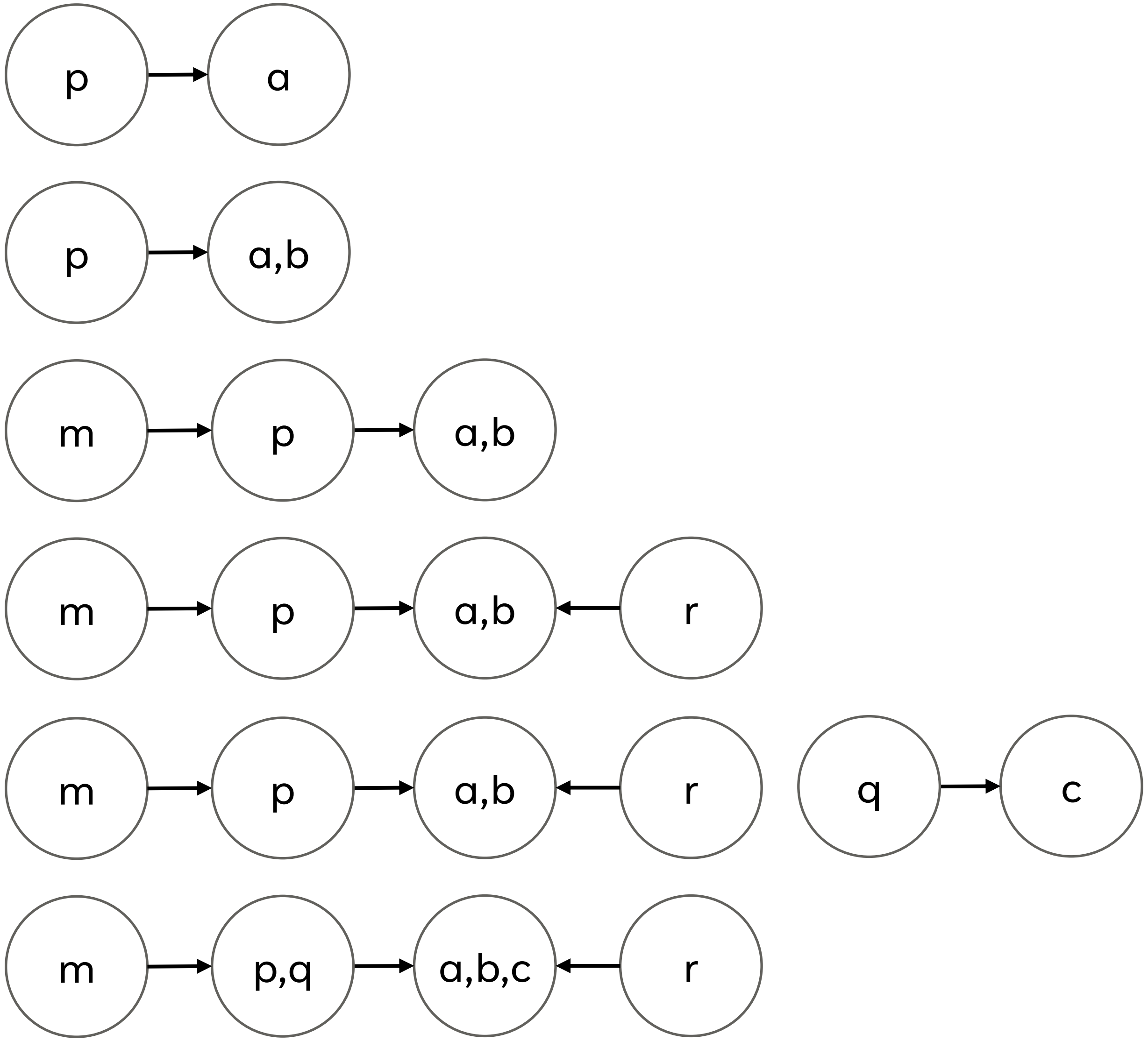
p = &b

m = &p

r = *m

q = &c

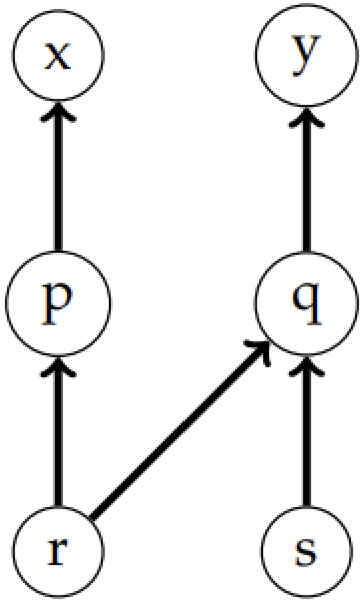
m = &q



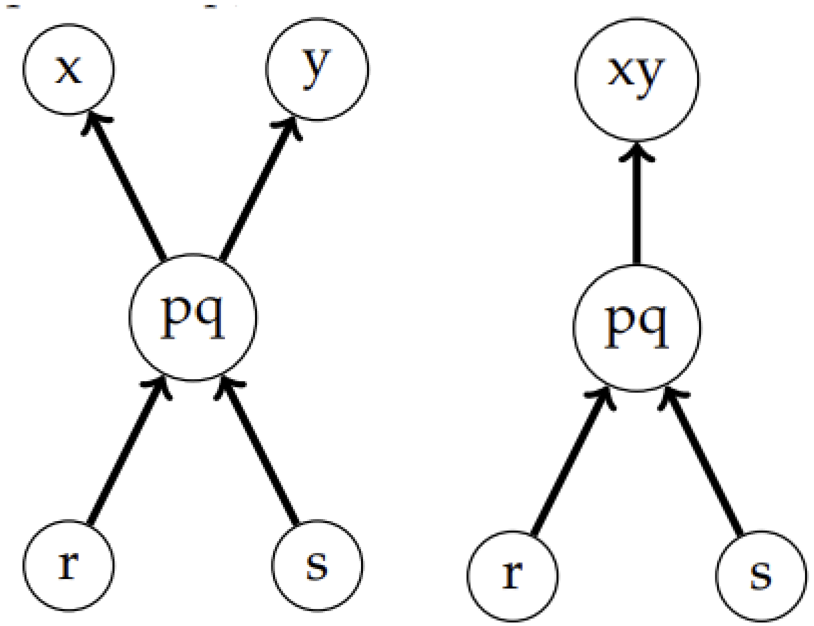
EQUALITY CONSTRAINTS
STEENSGARD'S ANALYSIS

$$
\begin{aligned}
1 :&\quad p := \&x\\
2 :&\quad r := \&p\\
3 :&\quad q := \&y\\
4 :&\quad s := \&q\\
5 :&\quad r := s
\end{aligned}
$$

Andersen's



Steensgard's



WRAP-UP