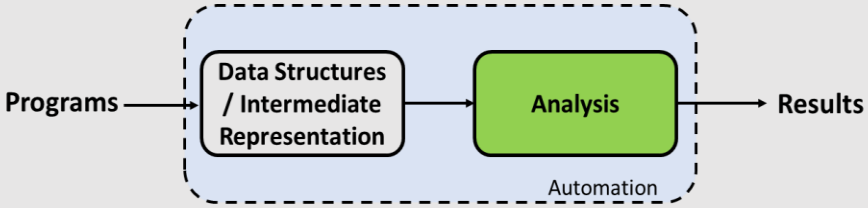
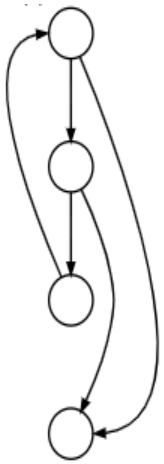
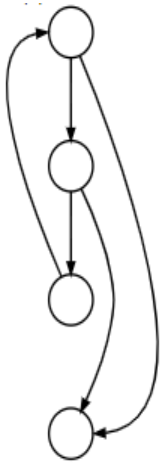




Program Analysis

Venkatesh Vinayakarao

venkateshv@cmi.ac.in
Mar – Apr, 2018
Chennai Mathematical Institute



"… we expect that the use of pointer analysis will become universal in modern programming tools, lending the analysis huge practical importance for future programming tasks."

– Yannis Smaragdakis and George Balatsouras, Foundations and Trends in Programming Languages, Vol 2, No. 1, 2015.

Quick Review – Program Representations

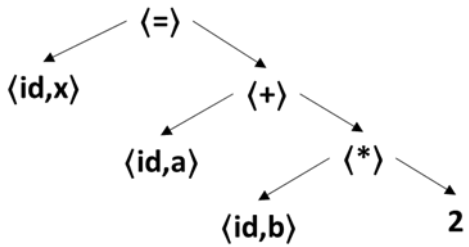
Lexemes

x = a + 2*b

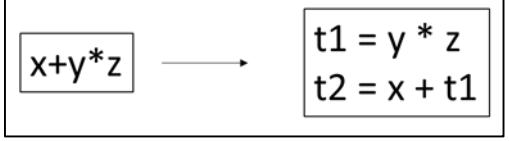
Tokens

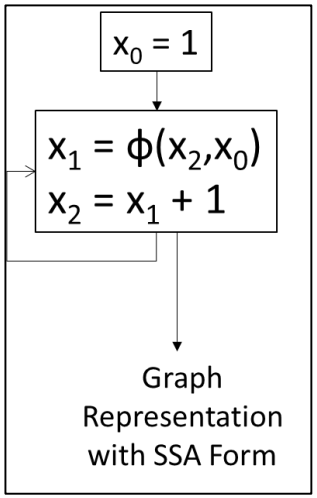
[(identifier, x), (operator, =), (identifier, a), (operator, +), (identifier, b), (operator, *), (literal, 2), (separator, ;)]

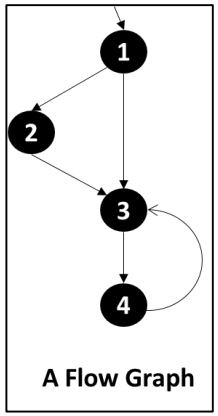
Abstract Syntax Tree

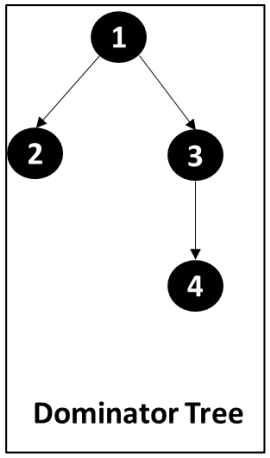


Three Address Code

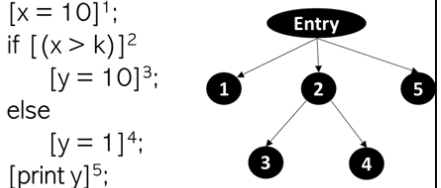


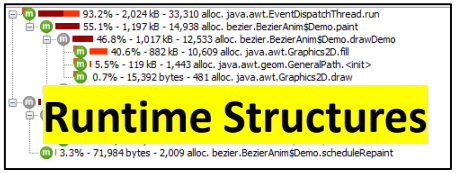






Control Dependence

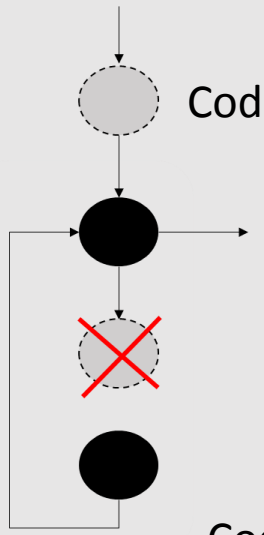




Tools

Eclipse JDT
Soot
JProfiler
Z3

Quick Review – Popular Compiler Optimizations



Loop Invariant Code Motion

```
int a = 30;
int b = 9 - (a / 5);
int c;

c = b * 4;
if (c > 10) {
  c = c - 10;
}
return c * (60 / a);
```

=

```
return 4;
```

Constant Folding and Propagation

$$prog: I_{fixed} \times I_{dynamic} \rightarrow O$$

$$\downarrow$$

$$prog^{*}: I_{dynamic} \rightarrow O$$

Partial Evaluation

Before	After
y = x / 8	y = x >> 3
y = x * 64	y = x << 6
y = x * 2	y = x << 1

Strength Reduction

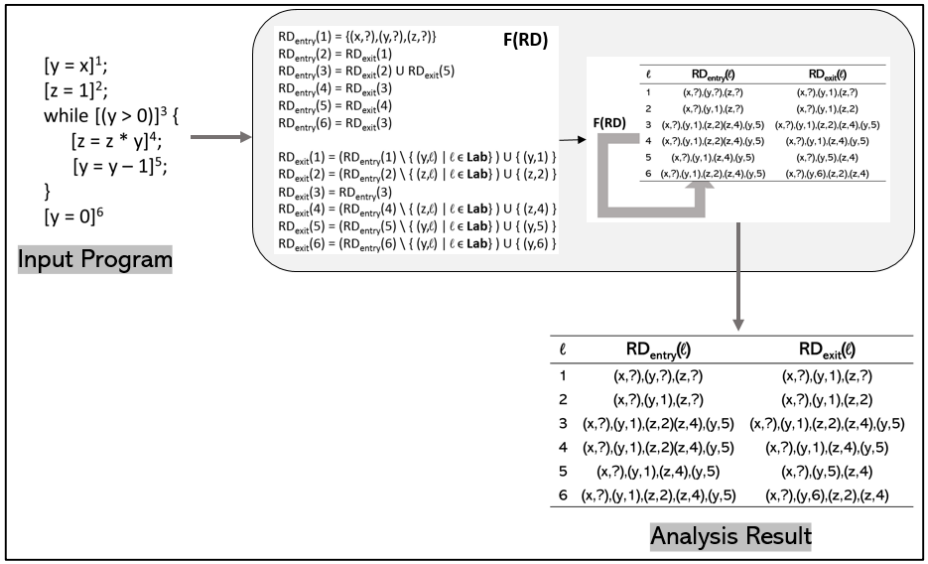
Peephole Optimization

```
float x,y,a,b,c,d;
…
x = (a/b)*c;
y = (a/b)*d;
```

Common Subexpressions

Quick Review

Data Flow as a System of Equations



Classic Four Analyses

$VB_{entry}(\ell)$	$(VB_{exit}(\ell) \setminus kill_{VB}(B^\ell)) \cup gen_{VB}(B^\ell)$	Backward Analyses
$VB_{exit}(\ell)$	$\emptyset$ if $\ell \in final(S_*)$ $\bigcap \{VB_{entry}(\ell') \mid (\ell', \ell) \in flow^R(S_*)\}$ otherwise	
$LV_{entry}(\ell)$	$(LV_{exit}(\ell) \setminus kill_{LV}(B^\ell)) \cup gen_{LV}(B^\ell)$	
$LV_{exit}(\ell)$	$\emptyset$ if $\ell \in final(S_*)$ $\bigcup \{LV_{entry}(\ell') \mid (\ell', \ell) \in flow^R(S_*)\}$ otherwise	
$RD_{entry}(\ell)$	$\{(x,?) \mid x \in Var_*\}$ if $\ell = init(S_*)$ $\bigcup \{RD_{exit}(\ell') \mid (\ell', \ell) \in flow(S_*)\}$ otherwise	Forward Analyses
$RD_{exit}(\ell)$	$(RD_{entry}(\ell) \setminus kill_{RD}(B^\ell)) \cup gen_{RD}(B^\ell)$	
$AE_{entry}(\ell)$	$\emptyset$ if $\ell = init(S_*)$ $\bigcap \{AE_{exit}(\ell') \mid (\ell', \ell) \in flow(S_*)\}$ otherwise	
$AE_{exit}(\ell)$	$(AE_{entry}(\ell) \setminus kill_{AE}(B^\ell)) \cup gen_{AE}(B^\ell)$	

Data Flow Analysis Framework

A Data Flow Analysis framework (P, ^, F) consists of:

- A bounded semilattice (P, ^)
  - Bounded since we need finite length ascending chain.
- A monotone transfer function
  - x <= y implies f(x) <= f(y)

forward or reverse flow.

init(S*) or final(S*)

union or intersection

$$Analysis_*(\ell) = \bigsqcup \{Analysis_*(\ell') \mid (\ell', \ell) \in F\} \sqcup \iota_E^\ell$$

$$\text{where } \iota_E^\ell = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases}$$

$$Analysis_*(\ell) = f_\ell(Analysis_*(\ell))$$

Soot

Points-To Analysis

The Analysis

- A pointer variable x **points to** y at a program point ℓ, if it holds the address of variable y at ℓ.

```
int  main() {
    int y, *x;
    y = 1;
    x = &y;
    return 0;
}
```

points-to relation that may hold in the whole program: {x→y}

Pointers in C

```
int main()
{
    int y;
    y = 1;
    printf("%d",y);
    return 0;
}
```

Prints “1”

```
int main()
{
    int y;
    y = 1;

    int *p;
    p = &y;

    printf("%d",*p);
    return 0;
}
```

Prints “1”

```
int main()
{
    int y;
    y = 1;

    int *p;
    p = &y;

    int *x;
    x = p;

    printf("%d",*x);
    return 0;
}
```

Prints “1”

Pointer Preliminaries

- Pointers in C

```
int  main() {
    int y, *x;
    y = 1;
    x = &y;
    return 0;
}
```

x points-to y
*x is 1

```
int  main() {
    int y, *x;
    y = 1;
    x = &y;
    *x = 2;
    return 0;
}
```

y = 2

```
int  main() {
    int y, *x, *p;
    y = 1;
    x = &y;
    p = x;
    *p = 2;
    return 0;
}
```

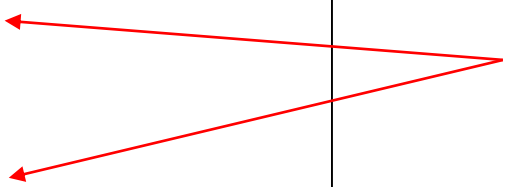
Copying a pointer
y = 2

```
int  main() {
    int z, y, *x;
    y = 1;
    x = &y;
    z = *x;
    return 0;
}
```

Dereferencing a pointer
z = 1

Points-To Analysis

```
int  main() {
      int z=1, y=1, *p;
      if (…some condition…) {
            p = &y;
      } else {
            p = &z;
      }
      *p = 2;
      printf(“%d”, y);
      return 0;
}
```



p may point-to
y or z.

Points-To Relation

```
int main(){
    int a, *x, **p;
    x=&a;
    p=&x;
    return 0;
}
```

{p→x, x→a}

or simply,

{(p,x),(x,a)}

points-to relation

Flow Sensitivity

		Flow Insensitive	Flow Sensitive
1	int x, y;		
2	int *p;	p → {x,y}	3: p → {x} 5: p → {y}
3	p = &x;		
4	x = 3;	Order of statements does not matter	Order of statements matter. Store results for each program point
5	p = &y;		

Program Analysis and Specialization for the C Programming Language, PhD Thesis, Andersen.

Flow-Insensitive Analysis

```
int x, y, *p, **q;
p = &x;
q = &p;
*q = &y;
```

Flow-Insensitive Analysis

```
int x, y, *p, **q;
p = &x;
q = &p;
*q = &y;
```

{p→{x,y}, q→{p}}

Context Sensitivity

```
1   int x, y, *p;
2
3    int main() {
4    f();
5    p = &x;
6    g();
7   }
```

```
8    int f() {
9          p = &y;
10         g();
11  }
```

```
12  int g() {
13        printf("%d", *p);
14  }
```

What values may p point to at line 13?

Context Insensitive	Context Sensitive
$p_{13} \rightarrow \{x,y\}$	10: $p_{13} \rightarrow \{y\}$ 6: $p_{13} \rightarrow \{x\}$

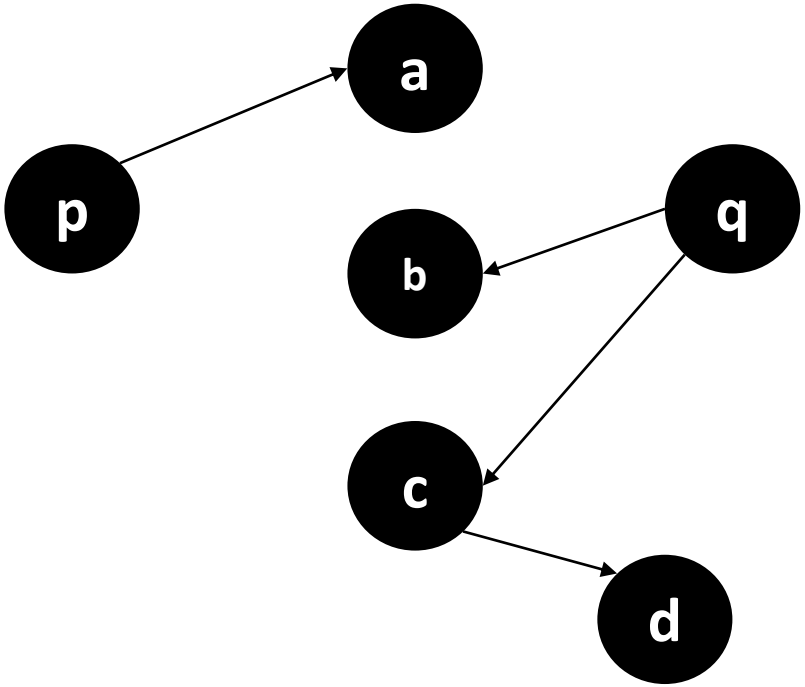
Calling Context

Anderson Style Analysis

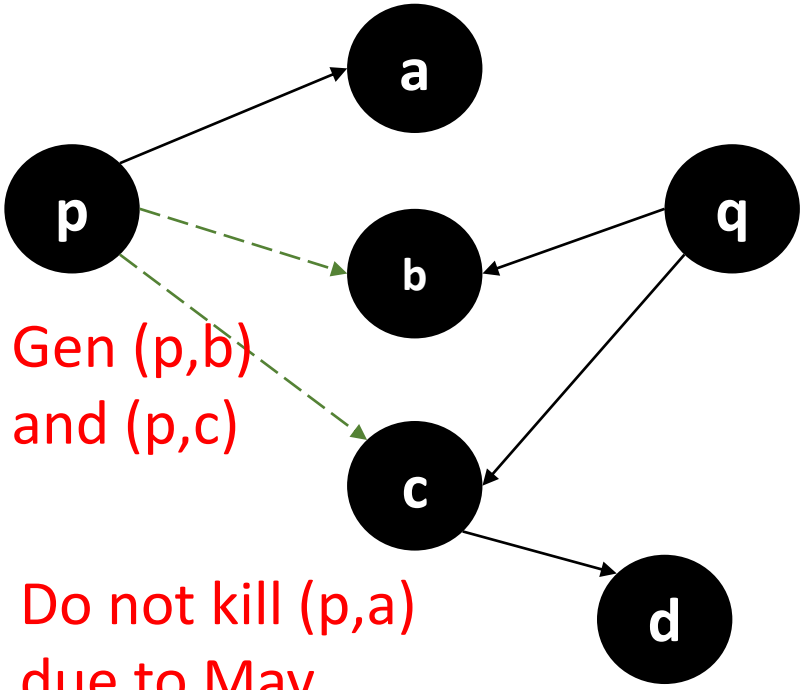
- Context Insensitive, Flow Insensitive.
- May Analysis
- Builds a points-to graph for the given code.
- Assumes that all pointer operations can be expressed in terms of:
  - p = q
  - *p = q
  - p = &q
  - p = *q

Andersen's Points-To Analysis

$\text{Entry}[p = q]^{\ell}$



$\text{Exit}[p = q]^{\ell}$

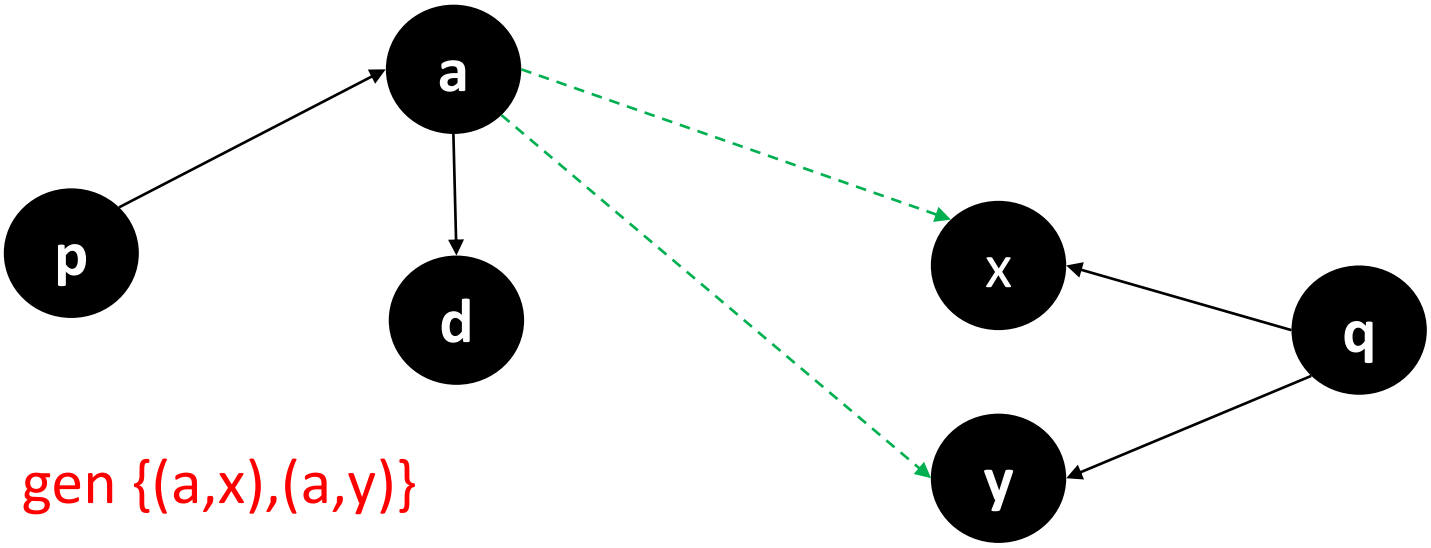


Gen (p,b) and (p,c)

Do not kill (p,a) due to May analysis

Andersen's Points-To Analysis

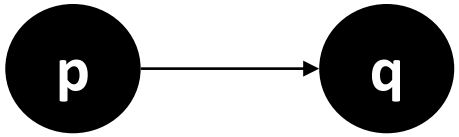
*p = q



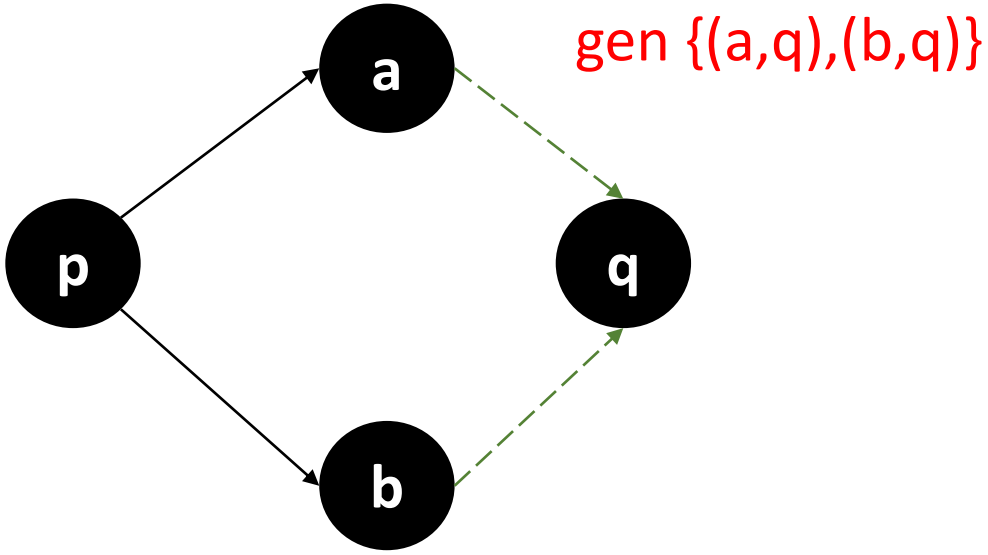
gen {(a,x),(a,y)}

Andersen's Points-To Analysis

p = &q

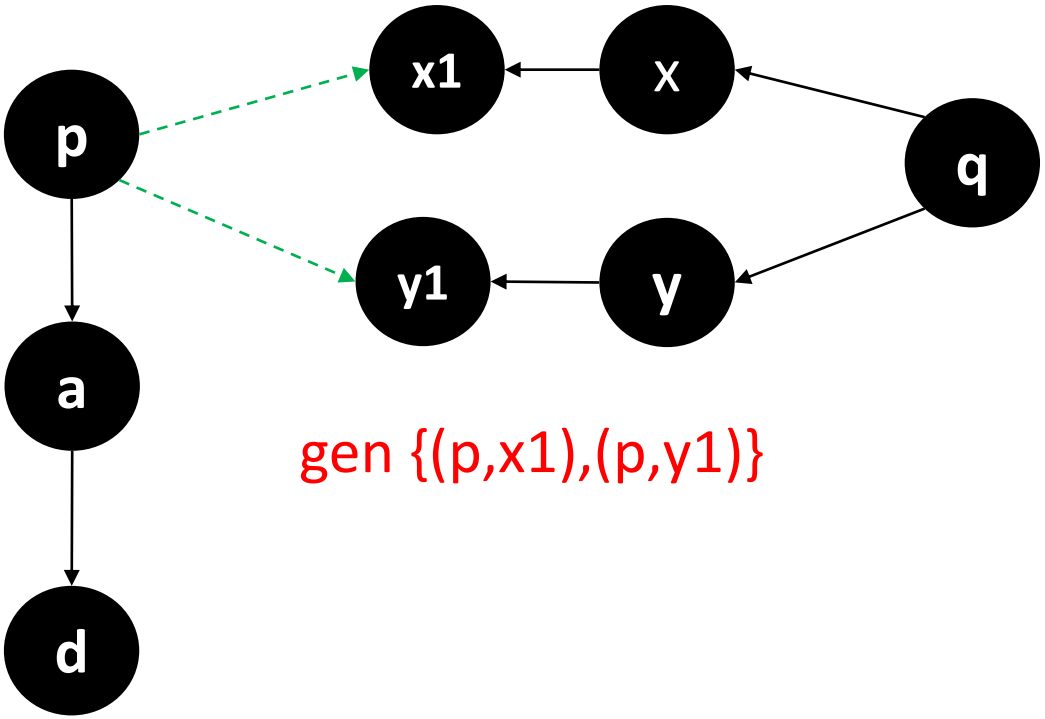


*p = &q



Andersen's Points-To Analysis

p = *q



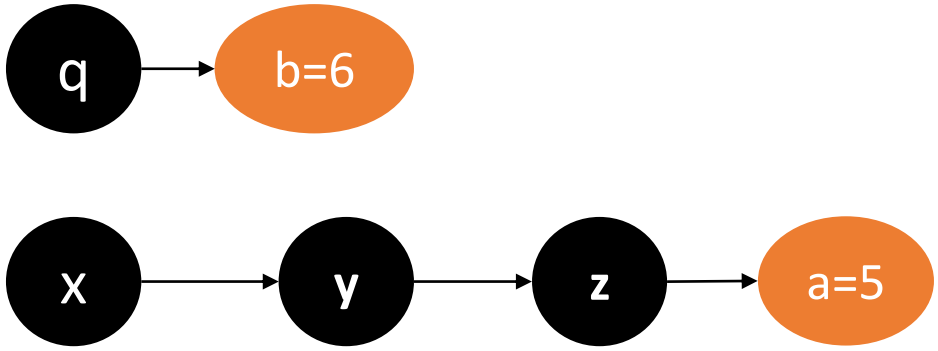
gen {(p,x1),(p,y1)}

Program Normalization

- Programs can be normalized to contain only the following statements if there are more levels of indirections. (Assumption)

```
p = &q
p = q
p = *q
*p = q
```

Program Normalization



How can you normalize the following code If the program is already in the above state?

```
p = **x;
*p = *q;
```

Program Normalization

```
1  void main() {
2      int a = 5, b = 6;
3      int **x, *y, *z, *p, *q;
4      z = &a;
5      y = &z;
6      x = &y;
7
8      q = &b;
9
10     //p = **x;
11     //*p = *q;
12
13     int *temp;
14     temp = *x;
15     p = *temp;
16     temp = *q;
17     *p = temp;
18
19     printf("%d", *p);
20 }
```

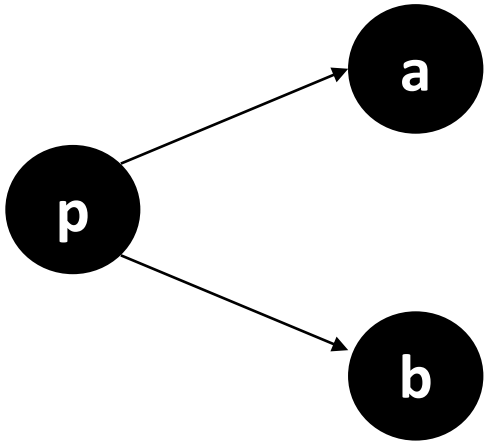
Worst Case Complexity is $O(n^3)$

- Graph can be constructed in one pass of the program with n variables.
- Statements may force the algorithm to visit n^2 nodes.

Program Analysis and Specialization for the C Programming Language, PhD Thesis, Andersen.

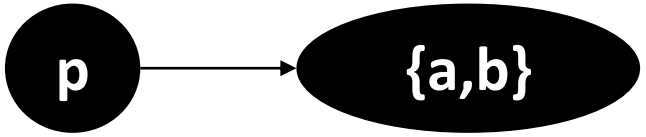
Steensgard's Analysis

- Start assuming all variables are distinct.
- Merge when you find multiple outgoing edges from a node. (so that there would be only one outgoing edge after merge)



Andersen-Style

Defining Equivalence Classes

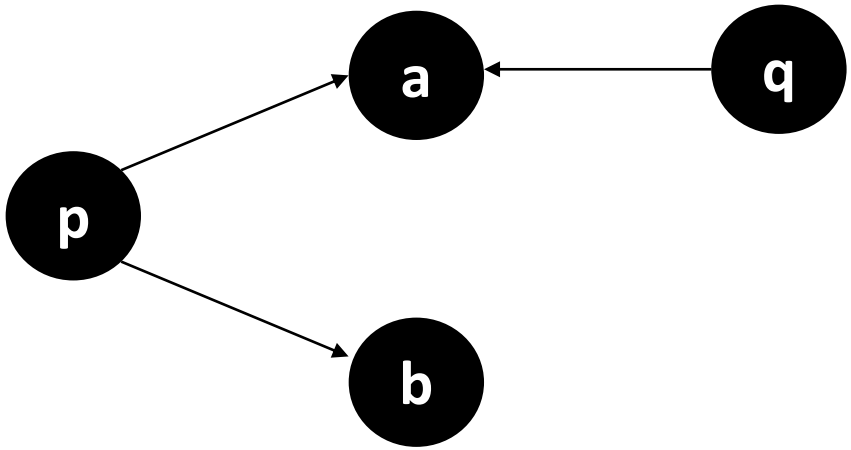


Steensgard-Style

Approximation for multiple outgoing edges

[Points-to analysis in almost linear time](), **Bjarne Steensgaard, POPL '96.**

Steensgard's Analysis

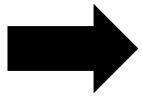


Inaccurate but fast. (Worst Case: O(n))

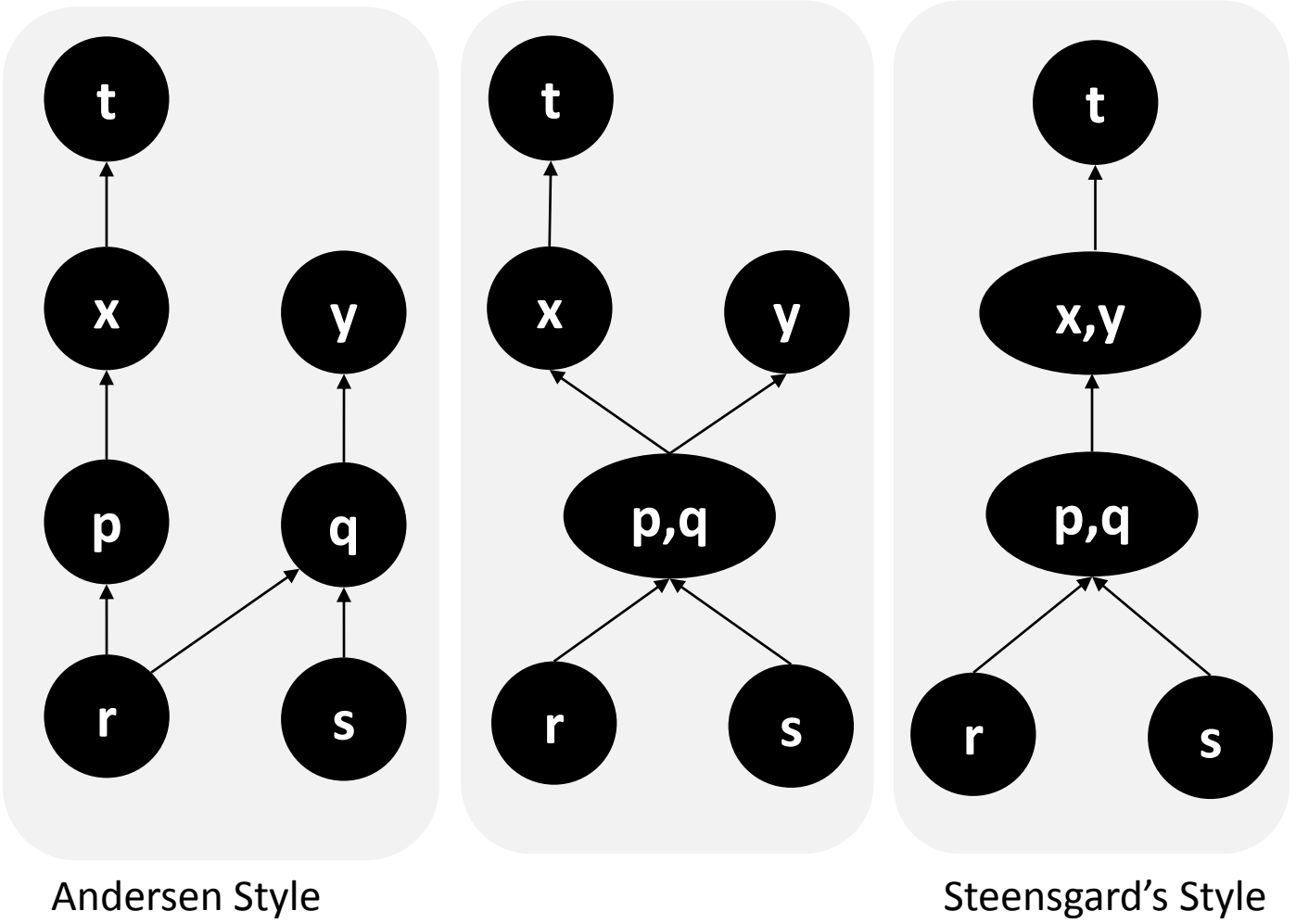
We claim, "p may point to a or b" and "q may point to a or b"

Comparison

p = &x
r = &p
q = &y
s = &q
r = s
*p = &t



Points-to Graphs



Andersen Style

Steensgard's Style