Pointer Analysis

Rupesh Nasre.

CS6843 Program Analysis
IIT Madras
Jan 2015

Outline

- Introduction
- Pointer analysis as a DFA problem
- Design decisions
- Andersen's analysis, Steensgaard's analysis
- Pointer analysis as a graph problem
  - Optimizations
- Applications
- Parallelization
  - Constraint based
  - Replication based
  - Graph rewrite rules

Applications

- Dead-code elimination
- Common subexpression elimination
- Parallelization
- Escape analysis

Dead Code Elimination

```
a = s1.arr;
b = s2.ptr;
q = &a[ii];
p = &b[jj];

if (p == q) {
        x = 10;
        y = 100;
} else {
        x = 20;
        y = 30;
}
```

To check the condition, we need to test if

- p == q
- a + ii * *typesize* == b + jj * *typesize*
- s1.arr + ii * *typesize* == s2.ptr + jj * *typesize*

This needs to be tested statically

Common Subexpression Elimination

```
q = s1.arr;
p = s1.ptr;

if (p + i == q + j) {
        x = 10;
        y = 100;
} else {
        x = 20;
        y = 30;
}
```

To identify if the expression is common

- p + ii == q + jj
- s1.arr + ii * *typesize_ii* == s1.ptr + jj * *typesize_jj*

This needs to be computed statically

Parallelization

```
f() {
        *p = 10;
}
g() {
        *q = 20;
}
main() {
        ...
        f();
        g();
}
```

To identify if the functions are parallelizable, check if

- !alias(*p, *q)

Escape Analysis

```
f() {
      *p = 10;
}
```

To identify if the definition escapes function f, check

- if p points-to any global / heap variable
- pointsto(p, x) where x ∈ globals or x ∈ heap-allocated

Parallel Pointer Analysis

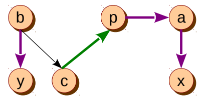
- putta-cc-2012 slides

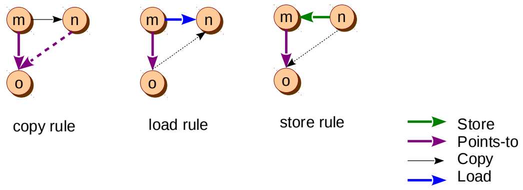
Pointer Analysis as Graph Rewrite Rules

- Initially: Constraint-based: pointers and associated points-to sets
- Later: Graph problem: pointers as nodes, subset relation forms edges, points-to set with each node
- Now: Graph rewrite rules: variables as nodes, all relations form edges, points-to set defined using edges

Graph Rewrite Rules

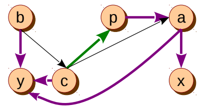
Program	Andersen's
a = &x;	a → {x, y}
b = &y;	b → {y}
p = &a;	c → {y}
c = b;	p → {a}
*p = c;	

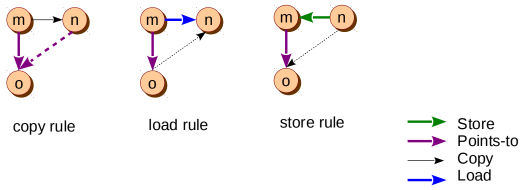




Graph Rewrite Rules

Program	Andersen's
a = &x;	a → {x, y}
b = &y;	b → {y}
p = &a;	c → {y}
c = b;	p → {a}
*p = c;	

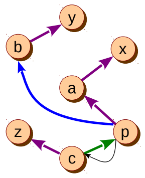


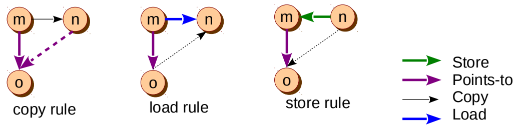


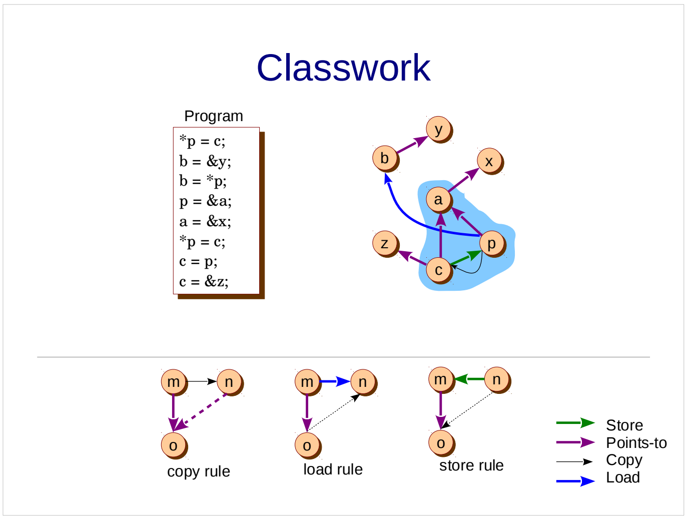
Classwork

Program

```
*p = c;
b = &y;
b = *p;
p = &a;
a = &x;
*p = c;
c = p;
c = &z;
```



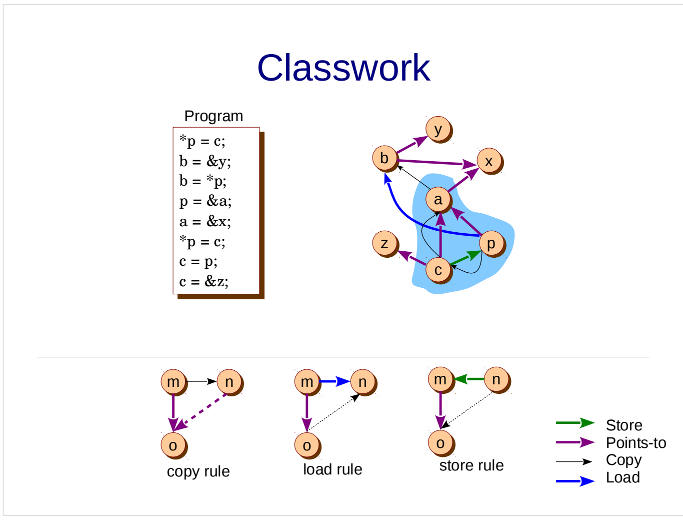




Classwork

Program

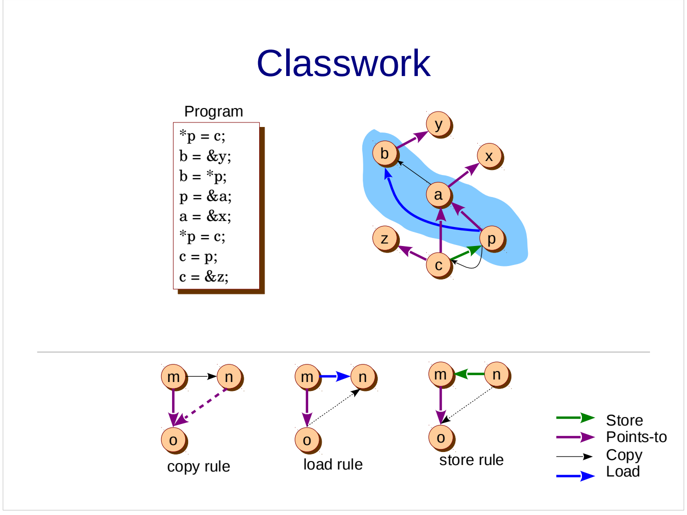
```
*p = c;
b = &y;
b = *p;
p = &a;
a = &x;
*p = c;
c = p;
c = &z;
```



Classwork

Program

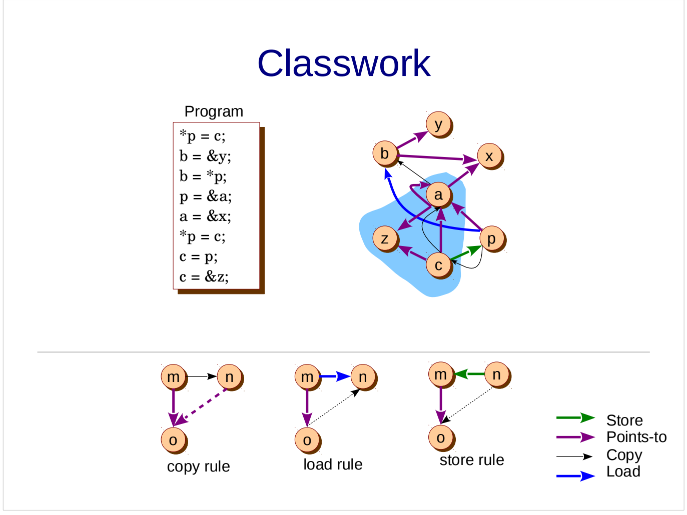
```
*p = c;
b = &y;
b = *p;
p = &a;
a = &x;
*p = c;
c = p;
c = &z;
```



Classwork

Program

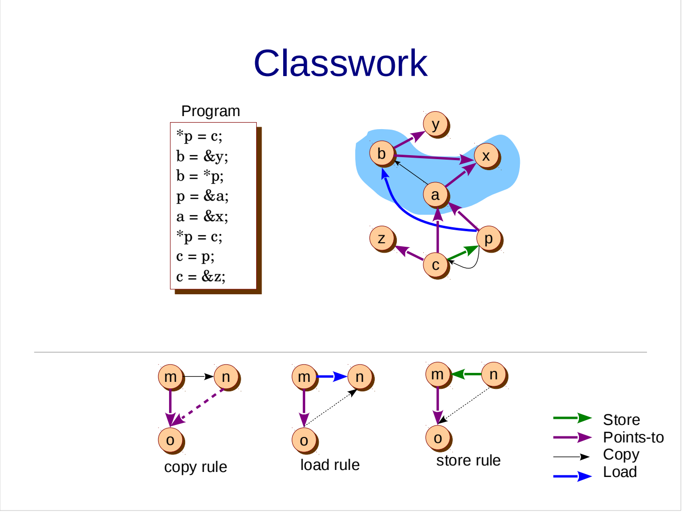
```
*p = c;
b = &y;
b = *p;
p = &a;
a = &x;
*p = c;
c = p;
c = &z;
```



Classwork

Program

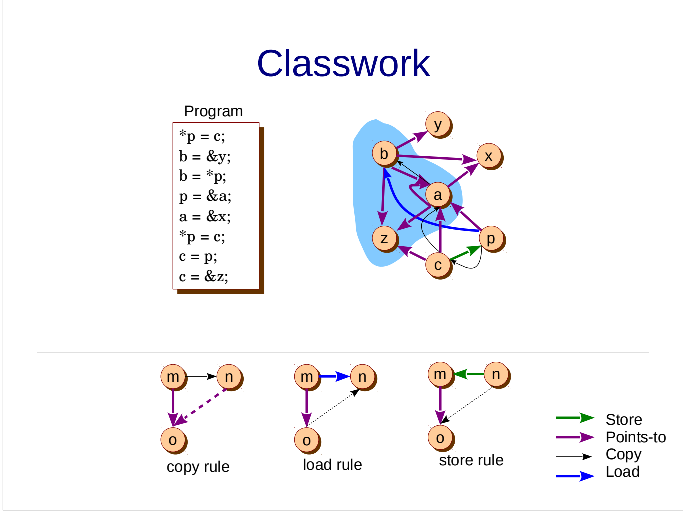
```
*p = c;
b = &y;
b = *p;
p = &a;
a = &x;
*p = c;
c = p;
c = &z;
```



Classwork

Program

```
*p = c;
b = &y;
b = *p;
p = &a;
a = &x;
*p = c;
c = p;
c = &z;
```



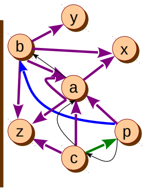
Classwork

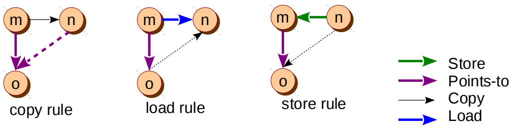
Program

```
*p = c;
b = &y;
b = *p;
p = &a;
a = &x;
*p = c;
c = p;
c = &z;
```

Classwork

Program	Andersen's
*p = c;	a→{a, x, z}
b = &y;	b→{a, x, y, z}
b = *p;	c→{a, z}
p = &a;	p→{a}
a = &x;	
*p = c;	
c = p;	
c = &z;	



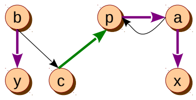


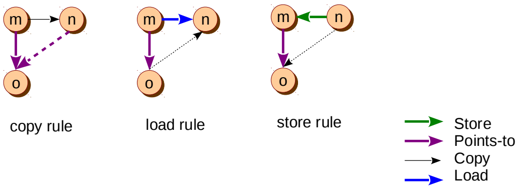
Parallel Graph Rewrite Rules

- *Open:* How to order rule evaluation?
- *Open:* How to combine rules for better efficiency?

Parallel Graph Rewrite Rules

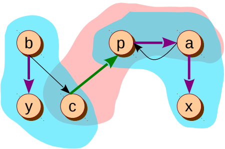
Program	Andersen's
a = &x;	a → {x, y}
b = &y;	b → {y}
p = &a;	c → {y}
c = b;	p → {a, x, y}
*p = c;	x → {y}
p = a;	y → {y}

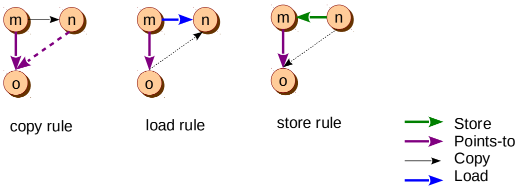




Parallel Graph Rewrite Rules

Program	Andersen's
a = &x;	a → {x, y}
b = &y;	b → {y}
p = &a;	c → {y}
c = b;	p → {a, x, y}
*p = c;	x → {y}
p = a;	y → {y}




m
n
o
copy rule
m
n
o
load rule
m
n
o
store rule
Store
Points-to
Copy
Load