



软件分析

# 指针分析

熊英飞  
北京大学  
2016



# 指向分析

- 每个指针变量可能指向的内存位置
- 通常是其他很多分析的基础
  
- 本节课先考虑流非敏感指向分析
- 不考虑在堆上分配的内存，不考虑struct、数组等结构，不考虑指针运算（如 $*(p+1)$ ）
  - 内存位置==局部和全局变量在栈上的地址



# 指向分析——例子

```
o=&v;  
q=&p;  
if (a > b) {  
    p=*q;  
    p=o; }  
*q=&w;
```

- 指向分析结果
  - $p = \{v, w\}$ ;
  - $q = \{p\}$ ;
  - $o = \{v\}$ ;
- 问题：如何设计一个指向分析算法？



# 复习：方程求解

- 数据流分析的传递函数和 $\sqcap$ 操作定义了一组方程
  - $D_{v_1} = F_{v_1}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
  - $D_{v_2} = F_{v_2}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
  - ...
  - $D_{v_n} = F_{v_n}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
- 其中
  - $F_{v_1}(D_{v_1}, D_{v_2}, \dots, D_{v_n}) = f_{v_1}(I)$
  - $F_{v_i}(D_{v_1}, D_{v_2}, \dots, D_{v_n}) = f_{v_i}(\sqcap_{j \in \text{pred}(i)} D_{v_j})$
- 数据流分析即为求解该方程的最大解
  - 传递函数和 $\sqcap$ 操作表达了该分析的安全性条件，所以该方程的解都是安全的
  - 最大解是最有用的解



# 从不等式到方程组

- 有一个有用的解不等式的unification算法
  - 不等式
    - $D_{v_1} \sqsubseteq F_{v_1}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
    - $D_{v_2} \sqsubseteq F_{v_2}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
    - ...
    - $D_{v_n} \sqsubseteq F_{v_n}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
  - 可以通过转换成如下方程组求解
    - $D_{v_1} = D_{v_1} \sqcap F_{v_1}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
    - $D_{v_2} = D_{v_2} \sqcap F_{v_2}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
    - ...
    - $D_{v_n} = D_{v_n} \sqcap F_{v_n}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$



# Anderson指向分析算法

赋值语句	约束
$a = \&b$	$a \supseteq \{b\}$
$a = b$	$a \supseteq b$
$a = *b$	$\forall v \in b. a \supseteq v$
$*a = b$	$\forall v \in a. v \supseteq b$

其他语句可以转换成这四种基本形式

$*a = **b;$    $c = *b;$   
 $d = *c;$   
 $*a = d;$



# Anderson指向分析算法-例

```
o=&v;  
q=&p;  
if (a > b) {  
    p=*q;  
    p=o; }  
*q=&w;
```

- 产生约束
  - $o \supseteq \{v\}$
  - $q \supseteq \{p\}$
  - $\forall v \in q. p \supseteq v$
  - $p \supseteq o$
  - $\forall v \in q. v \supseteq \{w\}$
- 如何求解这些约束



# 约束求解方法—通用框架

- 将约束
  - $\mathbf{o} \supseteq \{v\}$
  - $\mathbf{q} \supseteq \{p\}$
  - $\forall v \in \mathbf{q}. \mathbf{p} \supseteq v$
  - $\mathbf{p} \supseteq \mathbf{o}$
  - $\forall v \in \mathbf{q}. v \supseteq \{w\}$
- 转换成标准形式
  - $\mathbf{p} = \mathbf{p} \cup \mathbf{o} \cup (\cup_{v \in \mathbf{q}} v) \cup (p \in \mathbf{q} ? \{w\} : \emptyset)$
  - $\mathbf{q} = \mathbf{q} \cup \{p\} \cup (q \in \mathbf{q} ? \{w\} : \emptyset)$
  - $\mathbf{o} = \mathbf{o} \cup \{v\} \cup (o \in \mathbf{q} ? \{w\} : \emptyset)$
- 等号右边都是递增函数



# 求解方程组

- $p = p \cup o \cup (\cup_{v \in q} v) \cup (p \in q ? \{w\} : \emptyset)$
- $q = q \cup \{p\} \cup (q \in q ? \{w\} : \emptyset)$
- $o = o \cup \{v\} \cup (o \in q ? \{w\} : \emptyset)$

$$\begin{aligned} p &= \{ \} \\ q &= \{ \} \\ o &= \{ \} \end{aligned}$$



$$\begin{aligned} p &= \{ \} \\ q &= \{p\} \\ o &= \{v\} \end{aligned}$$



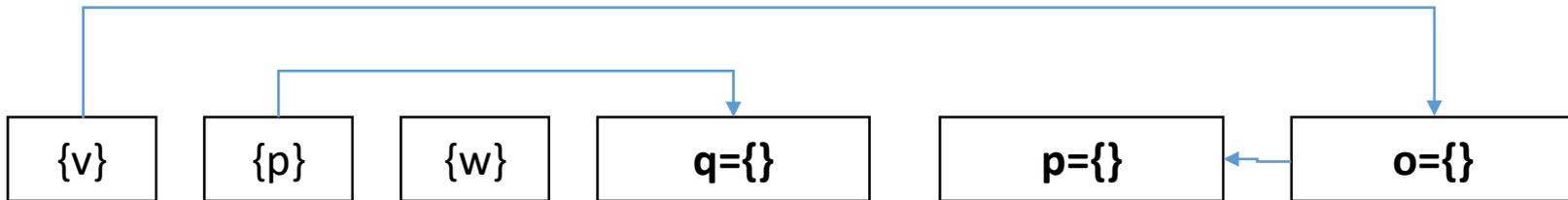
$$\begin{aligned} p &= \{v, w\} \\ q &= \{p\} \\ o &= \{v\} \end{aligned}$$





# 约束求解方法—直接计算

- $o \supseteq \{v\}$
- $q \supseteq \{p\}$
- $\forall v \in q. p \supseteq v$
- $p \supseteq o$
- $\forall v \in q. v \supseteq \{w\}$



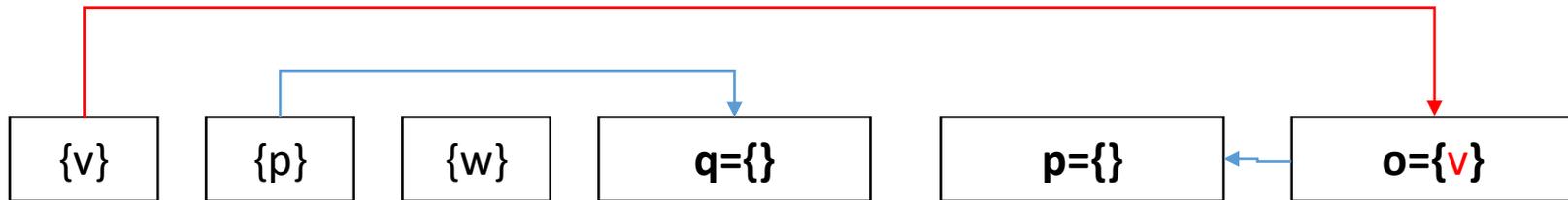
$$\forall v \in q. p \supseteq v$$

$$\forall v \in q. v \supseteq \{w\}$$



# 约束求解方法—直接计算

- $o \supseteq \{v\}$
- $q \supseteq \{p\}$
- $\forall v \in q. p \supseteq v$
- $p \supseteq o$
- $\forall v \in q. v \supseteq \{w\}$



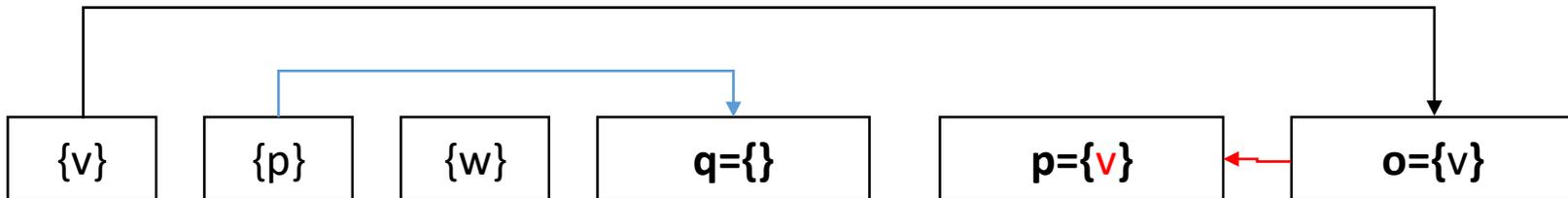
$$\forall v \in q. p \supseteq v$$

$$\forall v \in q. v \supseteq \{w\}$$



# 约束求解方法—直接计算

- $o \supseteq \{v\}$
- $q \supseteq \{p\}$
- $\forall v \in q. p \supseteq v$
- $p \supseteq o$
- $\forall v \in q. v \supseteq \{w\}$



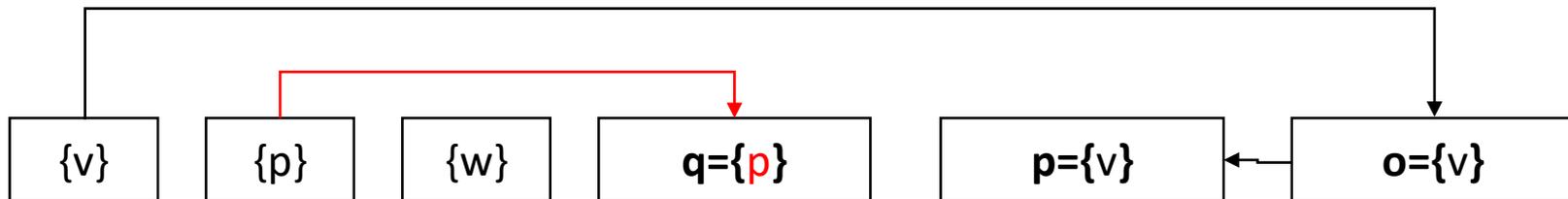
$$\forall v \in q. p \supseteq v$$

$$\forall v \in q. v \supseteq \{w\}$$



# 约束求解方法—直接计算

- $o \supseteq \{v\}$
- $q \supseteq \{p\}$
- $\forall v \in q. p \supseteq v$
- $p \supseteq o$
- $\forall v \in q. v \supseteq \{w\}$



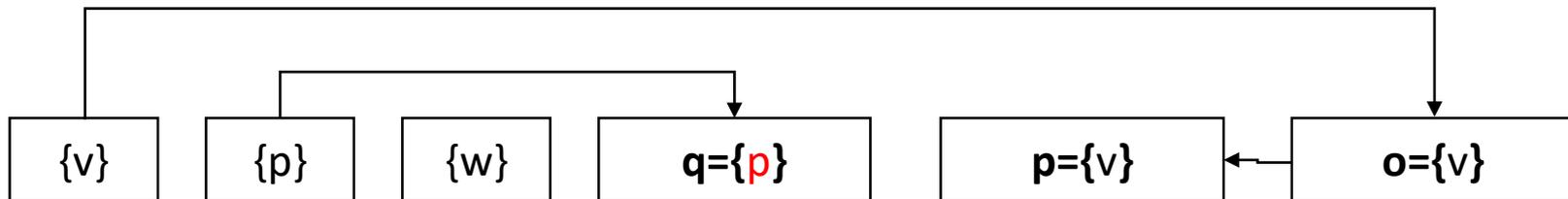
$$\forall v \in q. p \supseteq v$$

$$\forall v \in q. v \supseteq \{w\}$$



# 约束求解方法—直接计算

- $o \supseteq \{v\}$
- $q \supseteq \{p\}$
- $\forall v \in q. p \supseteq v$
- $p \supseteq o$
- $\forall v \in q. v \supseteq \{w\}$



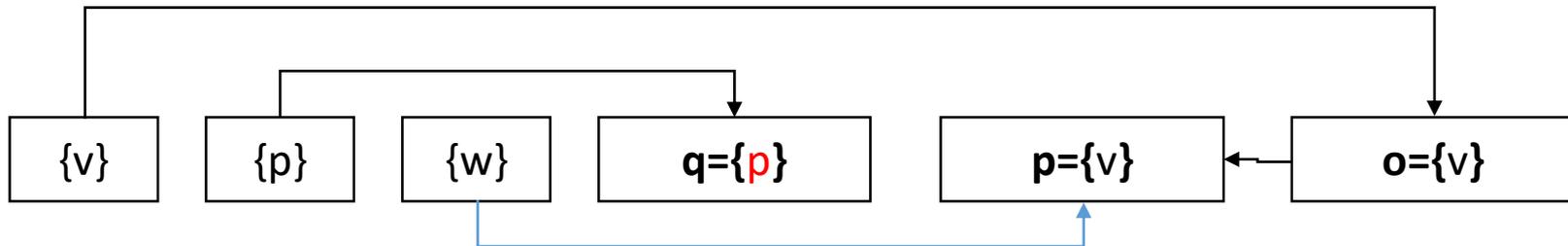
$$\forall v \in q. p \supseteq v$$

$$\forall v \in q. v \supseteq \{w\}$$



# 约束求解方法—直接计算

- $o \supseteq \{v\}$
- $q \supseteq \{p\}$
- $\forall v \in q. p \supseteq v$
- $p \supseteq o$
- $\forall v \in q. v \supseteq \{w\}$



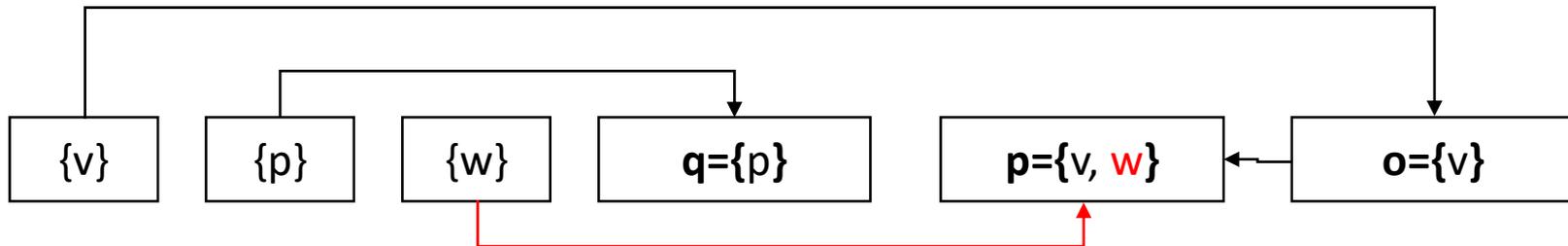
$$\forall v \in q. p \supseteq v$$

$$\forall v \in q. v \supseteq \{w\}$$



# 约束求解方法—直接计算

- $o \supseteq \{v\}$
- $q \supseteq \{p\}$
- $\forall v \in q. p \supseteq v$
- $p \supseteq o$
- $\forall v \in q. v \supseteq \{w\}$



$$\forall v \in q. p \supseteq v$$

$$\forall v \in q. v \supseteq \{w\}$$



# 复杂度分析

- 对于每条边来说，前驱集合新增元素的时候该边将被激活，激活后执行时间为 $O(m)$ ，其中 $m$ 为新增的元素数量
  - 应用均摊分析，每条边传递的总复杂度为 $O(n)$ ，其中 $n$ 为结点数数量
- 边的数量为 $O(n^2)$
- 总复杂度为 $O(n^3)$



# 进一步优化

- 强连通子图中的每个集合必然相等
- 动态检测图中的强连通子图，并且合并成一个集合
- 参考文献：
  - The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code, Hardekopf and Lin, PLDI 2007



# 流敏感的指针分析算法

- 能否通过SSA直接将流敏感转换成流非敏感分析？
  - 不能
- 如何把Anderson算法转换成数据流分析？
  - 半格集合是什么？
    - 指针变量到内存位置集合的映射
  - 交汇操作是什么？
    - 对应内存位置集合取并
  - 四种基本操作对应的转换函数是什么？



# 流敏感的指针分析算法

赋值语句	转换函数
$a = \&b$	$a_{out} := \{b\}$
$a = b$	$a_{out} := b_{in}$
$a = *b$	$a_{out} := \bigcup_{\forall v \in b} v_{in}$
$*a = b$	$\begin{cases} \forall v \in a. v_{out} := b_{in} &  a  = 1 \\ \forall v \in a. v_{out} := v_{in} \cup b_{in} &  a  > 1 \end{cases}$





# 流敏感的指针分析算法

- 传统流敏感的指针分析算法很慢
- 最新工作采用部分SSA来对流敏感进行加速，可以应用到百万量级的代码
- 参考论文：
  - Hardekopf B, Lin C. Flow-sensitive pointer analysis for millions of lines of code. CGO 2011:289-298.



# 堆上分配的内存

- `a=malloc();`
- `malloc()`语句每次执行创建一个内存位置
- 无法静态的知道`malloc`语句被执行多少次
  - 为什么?
    - 停机问题可以转换成求每个语句的执行次数
  - 造成什么影响?
    - 无法定义出有限半格
- 应用Widening
  - 每个`malloc()`创建一个抽象内存位置



# Struct

```
Struct Node {  
    int value;  
    Node* next;  
    Node* prev;  
};
```

```
a = malloc();  
a->next = b;  
a->prev = c;
```

- 如何处理结构体的指针分析？
- 域非敏感Field-Insensitive分析
- 域敏感Field-sensitive分析
- 猜一猜应该怎么做？

# 域非敏感Field-Insensitive分析



```
Struct Node {  
    int value;  
    Node* next;  
    Node* prev;  
};
```

```
a = malloc();
```

```
a->next = b;
```

```
a->prev = c;
```

- 把所有struct中的所有fields当成一个对象
- 原程序变为
  - a'=malloc();
  - a'=b;
  - a'=c;
  - 其中a'代表a, a->next, a->prev
- 分析结果
  - a, a->next, a->prev都有可能指向malloc(), b和c



# 域敏感Field-sensitive分析

```
Struct Node {  
    int value;  
    Node* next;  
    Node* prev;  
};
```

```
a = malloc();  
a->next = b;  
a->prev = c;
```

- 对于Node类型的内存位置x，添加两个指针变量
  - x.next
  - x.prev
- 对于任何Node类型的内存位置x，拆分成四个内存位置
  - x
  - x.value
  - x.next
  - x.prev
- a->next = b转换成
  - $\forall x \in a, x.next \supseteq b$



# Java上的指向分析

- Java上的指向分析可以看成是C上的子集

Java	C
<code>A a = new A();</code>	<code>A* a = malloc(sizeof(a));</code>
<code>a.next = b</code>	<code>a-&gt;next = b</code>
<code>b = a.next</code>	<code>b = a-&gt;next</code>

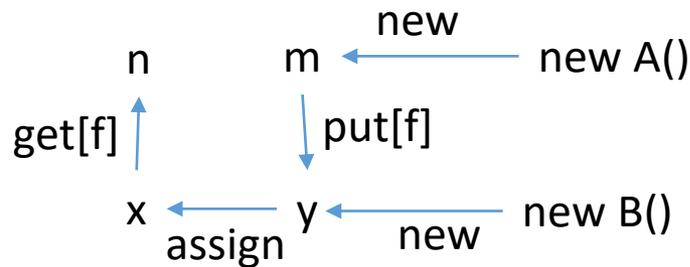


# 基于CFL可达性的域敏感分析

```

y = new B();
m=new A();
x=y;
y.f=m;
n=x.f;

```



图上的每条边f同时存在反向边f

```

FlowTo= new (assign | put[f] Alias get[f])*
PointsTo = (assign | get[f] Alias put[f])* new
Alias = PointsTo FlowTo

```

# 基于CFL和基于Anderson算法的域敏感分析等价性



基于CFL	基于Anderson算法
$\begin{array}{c} \text{PointsTo} \\ x \longrightarrow m \end{array}$	$m \in x$
$\begin{array}{c} \text{FlowsTo} \\ m \longrightarrow x \end{array}$	$m \in x$
$\begin{array}{c} \text{Alias} \\ x \longrightarrow y \end{array}$	$x \cap y \neq \emptyset$
$\exists y. y \xrightarrow{\text{PointsTo}} m \wedge y \xrightarrow{\text{puts}[f] \text{ PointsTo}} n$	$n \in m.f$

归纳证明 以上各行左右的等价性

- 从左边推出右边：在CFL的路径长度上做归纳
- 从右边推出左边：在集合的元素个数上做归纳



# 数组和指针运算

- 从本质上来讲都需要区分数组中的元素和分析下标的值
  - $p[i]$ ,  $*(p+i)$
- 大多数框架提供的指针分析算法不支持数组和指针运算
  - 一个数组被当成一个结点



# Steensgaard指向分析算法

- Anderson算法的复杂度为 $O(n^3)$
- Steensgaard指向分析通过牺牲精确性来达到效率
- 分析复杂度为 $O(n\alpha(n))$ ，接近线性时间。
  - $n$ 为程序中的语句数量。
  - $\alpha$ 为阿克曼函数的逆
  - $\alpha(2^{132}) < 4$



# Steensgaard指向分析算法

- Anderson算法执行速度较慢的一个重要原因是边数就达到 $O(n^2)$ 。
- 边数较多的原因是因为\*p的间接访问会导致动态创建边
- Steensgaard算法通过不断合并同类项来保证间接访问可以一步完成，不用创建边



# Steensgaard指向分析算法

```
o=&v;  
q=&p;  
if (a > b) {  
    p=*q;  
    p=o; }  
*q=&w;
```

- 产生约束

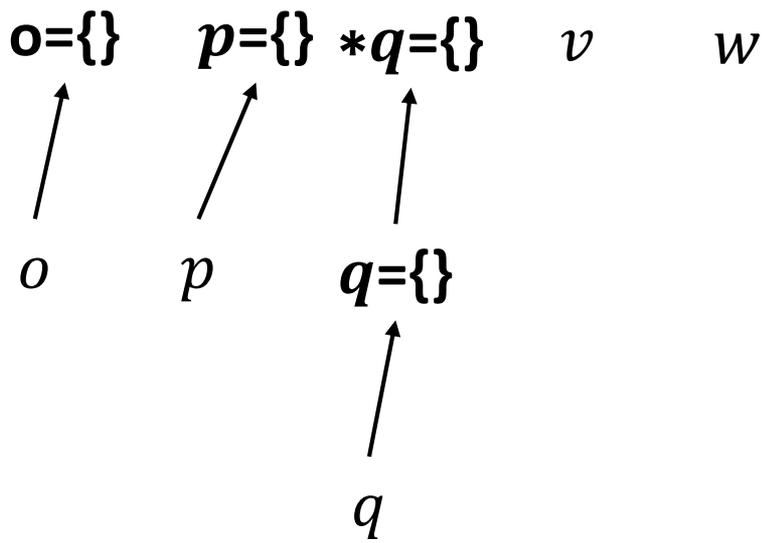
- $v \in o$
- $p \in q$
- $p = *q$
- $p = o$
- $w \in *q$
- $\forall y. \forall x \in y. x = *y$

- 赋值使得左右两边的集合相等
- 因为集合相等，所以只需要一个集合来表示
- 每个等号约束都是集合的合并



# 合并操作执行方法

- $v \in o$
- $p \in q$
- $p = *q$
- $p = o$
- $w \in *q$

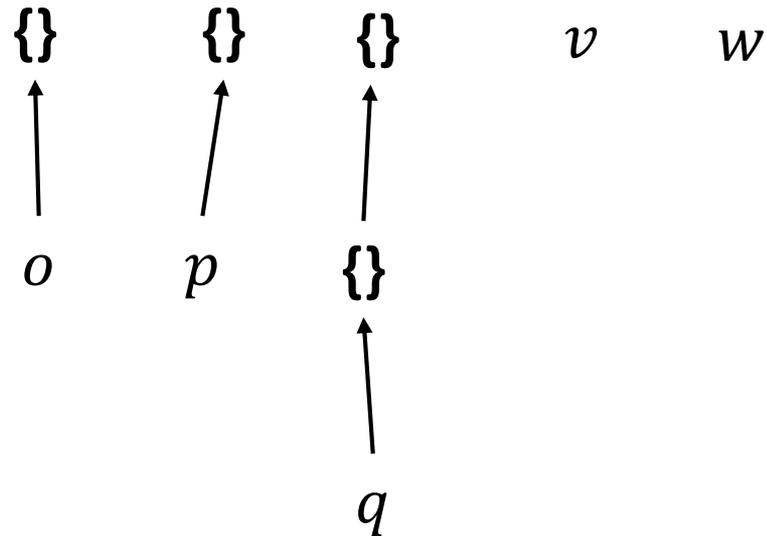


- 边表示指向关系
- 每个元素只能有一个后继
- 如果合并导致多于一个后继，就合并后继



# 合并操作执行方法

- $v \in o$
- $p \in q$
- $p =^* q$
- $p = o$
- $w \in^* q$

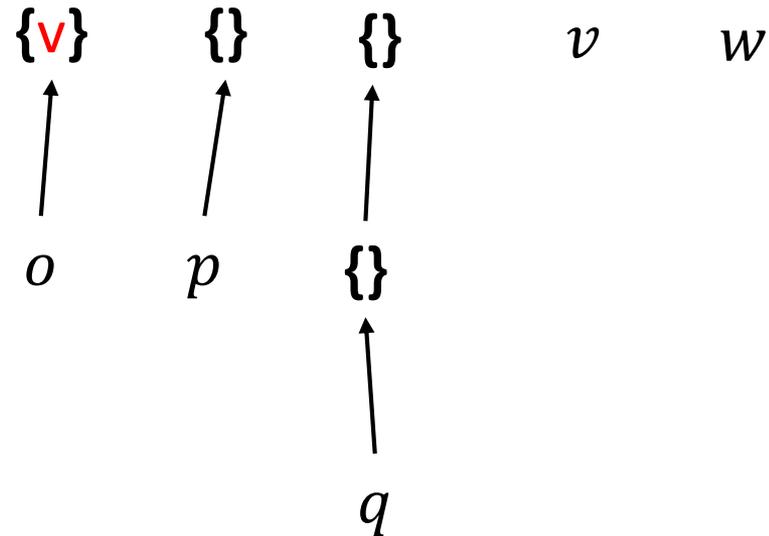


- 边表示指向关系
- 每个元素只能有一个后继
- 如果合并导致多于一个后继，就合并后继



# 合并操作执行方法

- $v \in o$
- $p \in q$
- $p = *q$
- $p = o$
- $w \in *q$

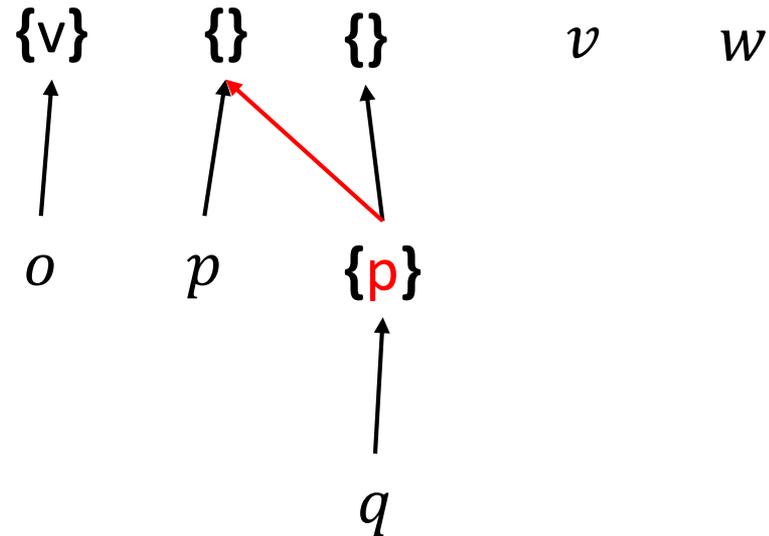


对于集合和元素的合并，添加元素到集合中，并添加元素的后继为集合的后继



# 合并操作执行方法

- $v \in o$
- $p \in q$
- $p = *q$
- $p = o$
- $w \in *q$

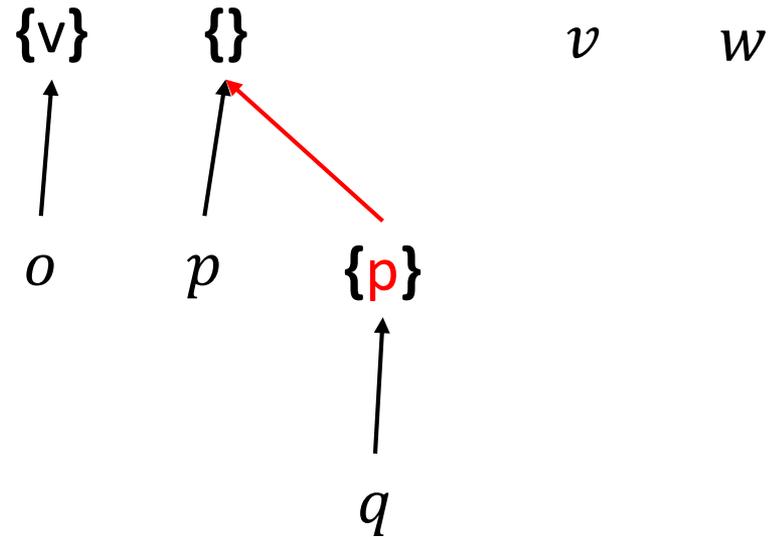


对于集合和元素的合并，添加元素到集合中，并添加元素的后继为集合的后继



# 合并操作执行方法

- $v \in o$
- $p \in q$
- $p =^* q$
- $p = o$
- $w \in^* q$

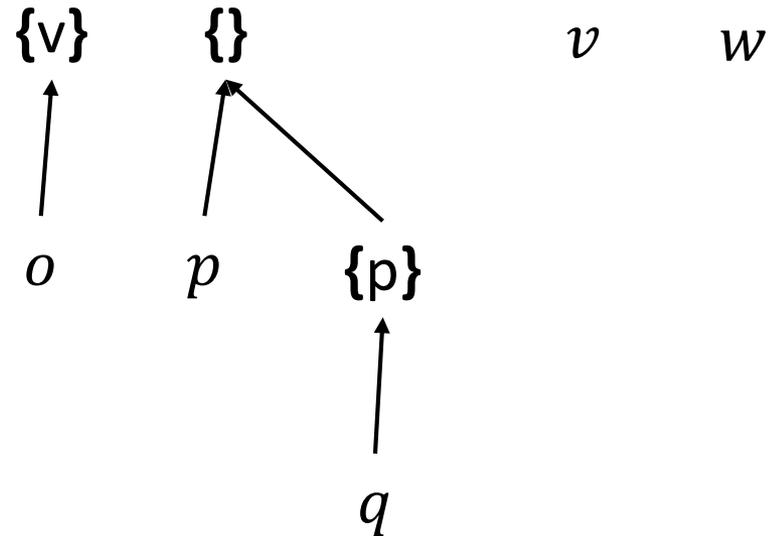


继续合并后继



# 合并操作执行方法

- $v \in o$
- $p \in q$
- **$p = *q$**
- $p = o$
- $w \in *q$

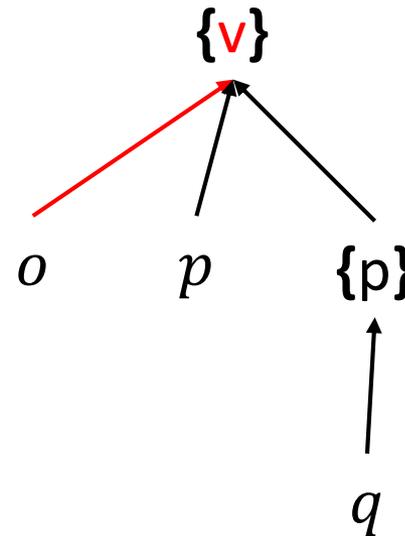


对于集合的合并，直接合并两个集合



# 合并操作执行方法

- $v \in o$
- $p \in q$
- $p =^* q$
- **$p = o$**
- $w \in^* q$



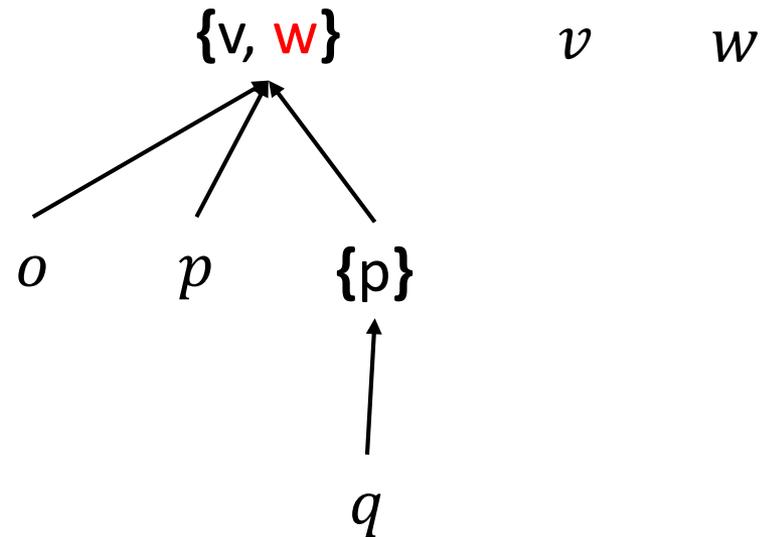
$v$        $w$

对于集合的合并，直接合并两个集合



# 合并操作执行方法

- $v \in o$
- $p \in q$
- $p = *q$
- $p = o$
- $w \in *q$



- 返回
  - $p = \{v, w\}$
  - $q = \{p\}$
  - $o = \{v, w\}$  //不精确



# 复杂度分析

- 节点个数为 $O(n)$
- 每次合并会减少一个节点，所以总合并次数是 $O(n)$
- 每次合并的时间开销包括
  - 集合的合并开销
  - 解析\*p等指针引用找到合适集合的开销
- 通过选择合适的数据结构（union-find structure），可以做到 $O(1)$ 时间的合并和 $O(\alpha(n))$ 的查找



# 术语

- Inclusion-based
  - 指类似Anderson方式的指针分析算法
- Unification-based
  - 指类似Steensgaard方式的指针分析算法



# 别名分析

- 给定两个变量  $a, b$ ，判断这两个变量是否指向相同的内存位置，返回以下结果之一
  - $a, b$  是 must aliases: 始终指向同样的位置
  - $a, b$  是 must-not aliases: 始终不指向同样的位置
  - $a, b$  是 may aliases: 可能指向同样的位置，也可能不指向
- 别名分析结果可以从指向分析导出
  - 如果  $a=b$  且  $|a|=1$ ，则  $a$  和  $b$  为 must aliases
  - 如果  $a \cap b = \emptyset$ ，则  $a$  和  $b$  为 must-not aliases
  - 否则  $a$  和  $b$  为 may aliases
- 别名分析本身有更精确的算法，但可伸缩性不高，在实践中较少使用



# 上下文敏感的指针分析

- 能否做精确的上下文敏感的指针分析？
- 域敏感的指针分析或者考虑二级指针的分析：不能
- 简单理论理解
  - 上下文无关性是一个上下文无关属性
    - 必须用下推自动机表示
  - 域敏感性也是一个上下文无关属性
  - 两个上下文无关属性的交集不一定是上下文无关属性
- Tom Reps等人2000年证明这是一个不可判定问题



# 解决方法

- 降低上下文敏感性：把被调方法根据上下文克隆n次
- 降低域敏感性：把域展开n次



# 域展开一次

```
Struct Node {  
    int value;  
    Node* next;  
};  
a = malloc();  
a->next = b;
```

- 对于每个Node\*的变量a，创建两个指针变量
  - a
  - a->next
- a->next=b产生
  - a->next  $\supseteq$  b
  - a->next  $\supseteq$  b->next
- a=b->next产生
  - a  $\supseteq$  b->next
  - a->next  $\supseteq$  b->next

约束中不含全程量词，可以用IFDS转成图并加上括号。



# 域展开两次

```
Struct Node {  
    int value;  
    Node* next;  
};
```

```
a = malloc();  
a->next = b;
```

- 对于每个Node\*的变量a，创建两个指针变量
  - a
  - a->next
  - a->next->next
- a->next=b产生
  - a->next  $\supseteq$  b
  - a->next->next  $\supseteq$  b->next
  - a->next->next  $\supseteq$  b->next->next
- a=b->next产生
  - a  $\supseteq$  b->next
  - a->next  $\supseteq$  b->next->next
  - a->next->next  $\supseteq$  b->next->next



# 过程间分析-函数指针

```
interface I {  
    void m();  
}  
class A implements I {  
    void m() { x = 1; }  
}  
class B implements I {  
    void m() { x = 2; }  
}  
static void main() {  
    I i = new A();  
    i.m();  
}
```

如何设计分析算法得出程序执行结束后的x所有可能的值？



# 控制流分析

- 确定函数调用目标的分析叫做控制流分析
- 控制流分析是may analysis
  - 为什么不是must analysis?
- 控制流分析 vs 数据流分析
  - 控制流分析确定程序控制的流向
  - 数据流分析确定程序中数据的流向
  - 数据流分析在控制流图上完成，因此控制流分析是数据流分析的基础



# Class Hierarchy Analysis

```
interface I {  
    void m(); }  
class A implements I {  
    void m() { x = 1; } }  
class B implements I {  
    void m() { x = 2; } }  
static void main() {  
    I i = new A();  
    i.m(); }  
class C { void m() {} }
```

- 根据i的类型确定m可能的目标
- 在这个例子中，i.m可能的目标为
  - A.m()
  - B.m()
- 不可能的目标为
  - C.m()
- 分析结果为 $x=\{1,2\}$
- 优点：简单快速
- 缺点：非常不精确，特别是有Object.equals()这类调用的时候



# Rapid Type Analysis

```
interface I {  
    void m(); }  
class A implements I {  
    void m() { x = 1; } }  
class B implements I {  
    void m() { x = 2; } }  
static void main() {  
    I i = new A();  
    i.m(); }  
class C { void m() {  
    new B().m();  
} }  
}}
```

- 只考虑那些在程序中创建了的对象
- 可以有效过滤library中的大量没有使用的类



# Rapid Type Analysis

```
interface I {  
    void m(); }  
class A implements I {  
    void m() { x = 1; } }  
class B implements I {  
    void m() { x = 2; } }  
static void main() {  
    I i = new A();  
    i.m(); }  
class C { void m() {  
    new B().m();  
} }  
}}
```

- 三个集合
  - 程序中可能被调用的方法集合Methods，初始包括main
  - 程序中所有的方法调用和对应目标Calls→Methods
  - 程序中所有可能被使用的类Classes
- **Methods**中每增加一个方法
  - 将该方法中所有创建的类型加到Classes
  - 将该方法中所有的调用加入到Call，目标初始为根据当前Classes集合类型匹配的方法
- **Classes**中每增加一个类
  - 针对每一次调用，如果类型匹配，把该类中对应的方法加入到Calls→Methods
  - 把方法加入到Methods当中



# Rapid Type Analysis

```
interface I {  
    void m(); }  
class A implements I {  
    void m() { x = 1; } }  
class B implements I {  
    void m() { x = 2; } }  
static void main() {  
    I i = new A();  
    I j = new B();  
    i.m(); }  
class C { void m() {  
    new B().m();  
} }  
}}
```

- 分析速度非常快
- 精度仍然有限
- 在左边的例子中，得出*i.m*的目标包括*A.m*和*B.m*
- 如何进一步分析出精确的结果？



# 精确的控制流分析CFA

- 该算法没有名字，通常直接称为CFA (control flow analysis)
- CFA和指针分析需要一起完成
  - 指针分析确定调用对象
  - 调用对象确定新的指向关系
- 原始算法定义在 $\lambda$ 演算上
- 这里介绍算法的面向对象版本



# CFA-算法

```

interface I {
  I m(); }
class A implements I {
  I m() { return new B(); } }
class B implements I {
  I m() { return new A(); } }
static void main() {
  I i = new A();
  if (...) i = i.m();
  I x = i.m();
}

```

- 首先每个方法的参数和返回值都变成图上的点
  - 注意this指针是默认参数
- 对于方法调用
 

```
f() { ...
  x = y.g(a, b)
  ... }
```
- 生成约束
  - $\forall y \in f\#y. \forall m \in \text{targets}(y, g),$   
 $f\#x \supseteq m\#ret$   
 $m\#this \supseteq \text{filter}(f\#y, \text{declared}(m))$   
 $m\#a \supseteq f\#a$   
 $m\#b \supseteq f\#b$
- 约束求解方法和Anderson指针分析算法类似

根据调用对象和方法名确定被调用方法

方法的声明类

保留符合特定类型的对象



# CFA-计算示例

```
interface I {  
  I f(); }  
class A implements I {  
  I f() { return new B1(); } }  
class B implements I {  
  I f() { return new A2(); } }  
static void main() {  
  I i = new A3();  
  if (...) i = i.f();  
  I x = i.f();  
}
```

- **main#i**  $\supseteq \{3\}$
- $\forall i \in \mathbf{main\#i}, \forall m \in \mathbf{targets(i, f)}$ ,
  - **main#i**  $\supseteq$  **m#ret**
  - **m#this**  $\supseteq$  filter(**main#i**, declared(m))
- **A.f#ret**  $\supseteq \{1\}$
- **B.f#ret**  $\supseteq \{2\}$
- $\forall m \in \mathbf{targets(main\#i, f)}$ ,
  - **main#x**  $\supseteq$  **m#ret**
  - **m#this**  $\supseteq$  filter (**main#i**, declared(m))
- 求解结果
  - **main#i** = {1, 2, 3}
  - **main#x** = {1, 2}



# CFA

- 以上CFA算法是否是上下文敏感的？
- 不是，因为每个方法只记录了一份信息，没有区分上下文
- 用克隆的方法处理上下文敏感性
- 基于克隆方法的CFA也被称为m/k-CFA
  - 上下文不敏感的CFA称为0-CFA



# 流敏感vs上下文敏感

- 当不能同时做到两种精度时，优先考虑哪个？
  - 通常认为，在C语言等传统命令式语言中流敏感性比较重要
  - 在Java、C++等面向对象语言中上下文敏感性比较重要
  - 主流指针分析算法通常时上下文敏感而流非敏感的



# 实践中的指针分析算法

- 大多数代码分析框架都提供指针分析
- 除非研究指针分析本身，很少需要自己搭建指针分析
- 但是需要了解各种不同的分析算法对精度和速度的影响，以便选择合适的指针分析算法



# 作业

- SOOT本身带有指针分析PADDLE和SPARK，查找资料并回答以下问题
  - 这两个算法是Anderson风格， Steensgaard风格，还是两者都不是？
  - 这两个算法是否是flow-sensitive的？
  - 这两个算法是否是field-sensitive的？
  - 这两个算法是否是Context-sensitive的？ 是什么意义上的Context-sensitivity？
  - 这两个算法是如何进行控制流分析的？