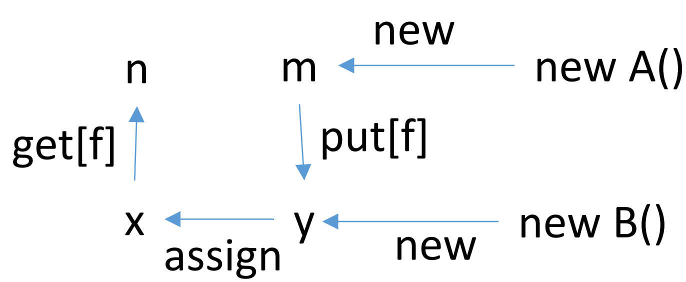
软件分析

其他过程内指针分析
过程间指针分析

熊英飞
北京大学

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;
```

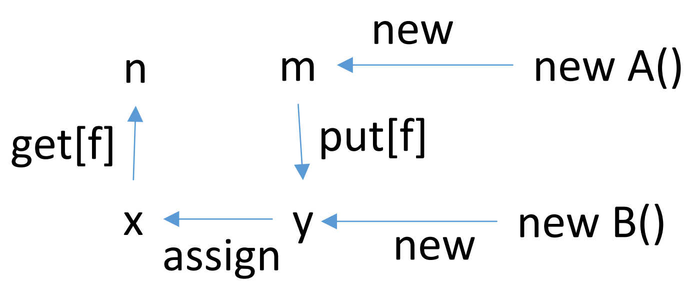


图上的每条边f同时存在反向边f

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;
```



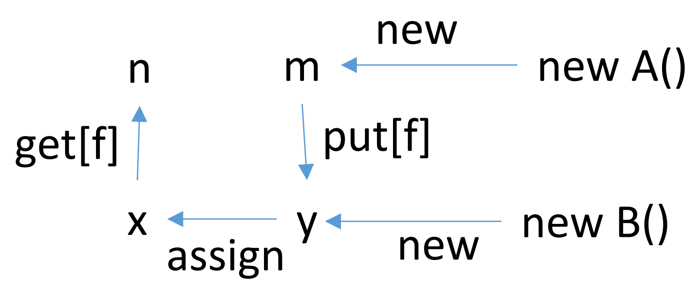
图上的每条边f同时存在反向边$\underline{\text{f}}$

FlowTo= new (assign | put[f] Alias get[f])*
PointsTo = ($\underline{\text{assign}}$ | $\underline{\text{get[f]}}$ Alias $\underline{\text{put[f]}}$)* $\underline{\text{new}}$
Alias = PointsTo FlowTo

能否不定义Alias关系？比如：
FlowTo = new FlowTo'
FlowTo' = put[f] FlowTo' get[f]
| FlowTo' FlowTo' | assign | ϵ

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$\underline{f}$

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

PointsTo = ($\underline{\text{assign}}$ | $\underline{\text{get[f]}}$ Alias $\underline{\text{put[f]}}$)* $\underline{\text{new}}$

Alias = PointsTo FlowTo

> 能否把assign统一定义在Alias内部？如：
>
> FlowTo= new (Alias | put[f] Alias get[f])*
>
> PointsTo = (Alias | $\underline{\text{get[f]}}$ Alias $\underline{\text{put[f]}}$)* $\underline{\text{new}}$
>
> Alias = PointsTo FlowTo | assign | $\underline{\text{assign}}$

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
$x \xrightarrow{\text{PointsTo}} m$	$m \in \mathbf{x}$
$m \xrightarrow{\text{FlowsTo}} x$	$m \in \mathbf{x}$
$x \xrightarrow{\text{Alias}} y$	$\mathbf{x} \cap \mathbf{y} \neq \emptyset$
$\exists y.\, y \xrightarrow{\text{PointsTo}} n \wedge y \xrightarrow{\text{puts[f] PointsTo}} m$	$n \in \mathbf{m.f}$

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

复习：Anderson指向分析算法

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

- 产生约束
  - $\boldsymbol{o} \supseteq \{v\}$
  - $\boldsymbol{w} \supseteq \{w\}$
  - $\boldsymbol{q} \supseteq \{p\}$
  - $\boldsymbol{q} \supseteq \{r\}$
  - $\forall v \in \boldsymbol{q}. \boldsymbol{v} \supseteq \boldsymbol{p}$
  - $\forall v \in \boldsymbol{q}. \boldsymbol{w} \supseteq \boldsymbol{v}$
  - $\boldsymbol{p} \supseteq \boldsymbol{o}$

q={}

p={}

o={}

w={}

r={}

复习：Anderson指向分析算法

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

- 产生约束
  - $\boldsymbol{o} \supseteq \{v\}$
  - $\boldsymbol{w} \supseteq \{w\}$
  - $\boldsymbol{q} \supseteq \{p\}$
  - $\boldsymbol{q} \supseteq \{r\}$
  - $\forall v \in \boldsymbol{q}. \boldsymbol{v} \supseteq \boldsymbol{p}$
  - $\forall v \in \boldsymbol{q}. \boldsymbol{w} \supseteq \boldsymbol{v}$
  - $\boldsymbol{p} \supseteq \boldsymbol{o}$

q={pr}

p={}

o={v}

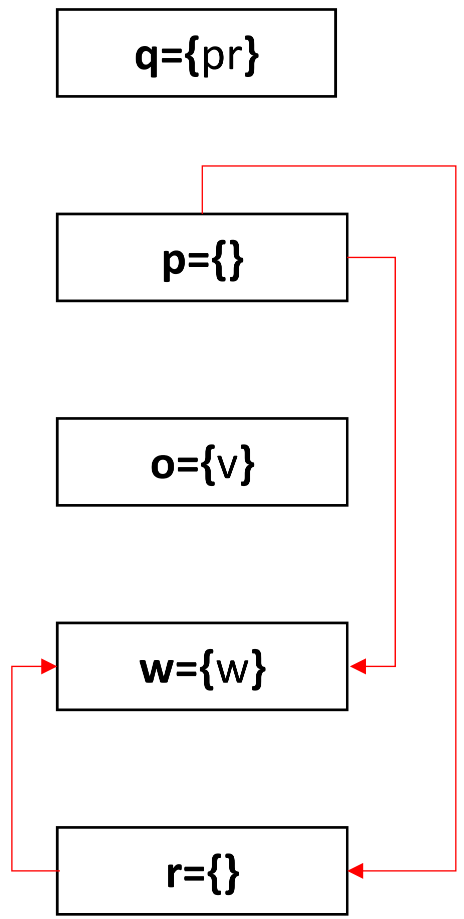
w={w}

r={}

复习：Anderson指向分析算法

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

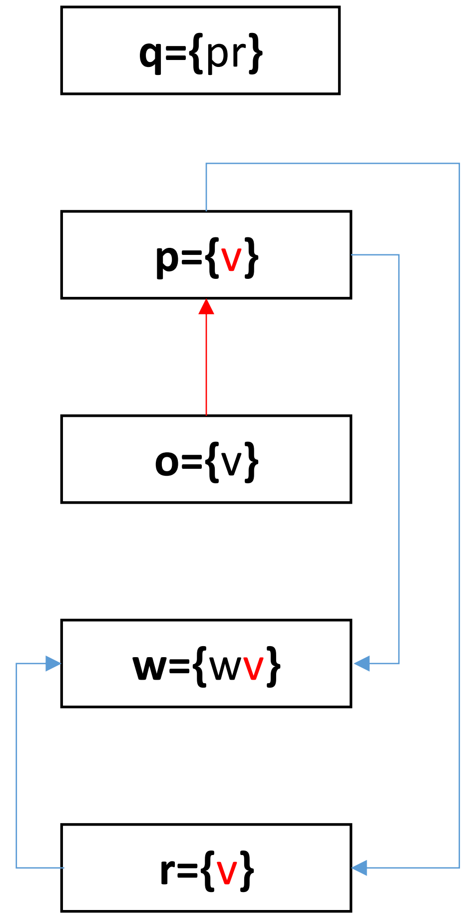
- 产生约束
  - $\boldsymbol{o} \supseteq \{v\}$
  - $\boldsymbol{w} \supseteq \{w\}$
  - $\boldsymbol{q} \supseteq \{p\}$
  - $\boldsymbol{q} \supseteq \{r\}$
  - $\forall v \in \boldsymbol{q}. \boldsymbol{v} \supseteq \boldsymbol{p}$
  - $\forall v \in \boldsymbol{q}. \boldsymbol{w} \supseteq \boldsymbol{v}$
  - $\boldsymbol{p} \supseteq \boldsymbol{o}$



复习：Anderson指向分析算法

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

- 产生约束
  - $\boldsymbol{o} \supseteq \{v\}$
  - $\boldsymbol{w} \supseteq \{w\}$
  - $\boldsymbol{q} \supseteq \{p\}$
  - $\boldsymbol{q} \supseteq \{r\}$
  - $\forall v \in \boldsymbol{q}. \boldsymbol{v} \supseteq \boldsymbol{p}$
  - $\forall v \in \boldsymbol{q}. \boldsymbol{w} \supseteq \boldsymbol{v}$
  - $\boldsymbol{p} \supseteq \boldsymbol{o}$

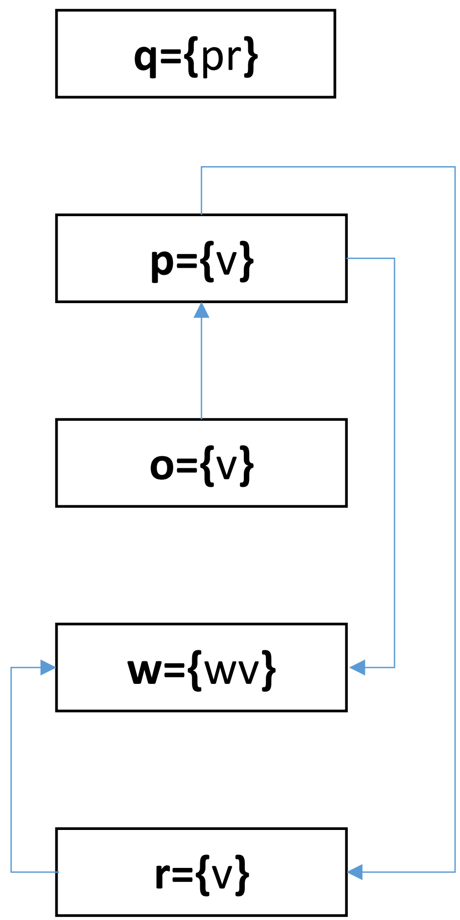


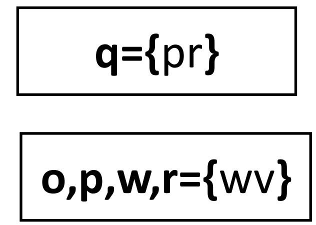
Steensgaard指向分析算法

- Anderson算法的开销主要来自于顺着边传递地址
  - 复杂度为$O(n^3)$
- 取消一部分传递能显著提升效率
  - The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code, Hardekopf and Lin, PLDI 2007
- 能否通过牺牲精度来彻底取消这个传递?
- Steensgaard指向分析算法
  - 通过合并节点避免传递
  - 复杂度为$O\big(n\alpha(n)\big)$，接近线性时间

Steensgaard指向分析结果

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





如果有可能需要将元素同时添加到两个集合中，则将两个集合合并。

有两种合并：

1. 有边可达的节点合并成同一个。
   a. 即超集关系换成等价关系
2. 被同一个指针指向的节点合并成同一个
   a. 因为对该指针的读写会导致相关节点合并

Steensgaard指向分析算法

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

- Anderson约束
  - $\boldsymbol{o} \supseteq \{v\}$
  - $\boldsymbol{w} \supseteq \{w\}$
  - $\boldsymbol{q} \supseteq \{p\}$
  - $\boldsymbol{q} \supseteq \{r\}$
  - $\forall v \in \boldsymbol{q}.\boldsymbol{v} \supseteq \boldsymbol{p}$
  - $\forall v \in \boldsymbol{q}.\boldsymbol{w} \supseteq \boldsymbol{v}$
  - $\boldsymbol{p} \supseteq \boldsymbol{o}$

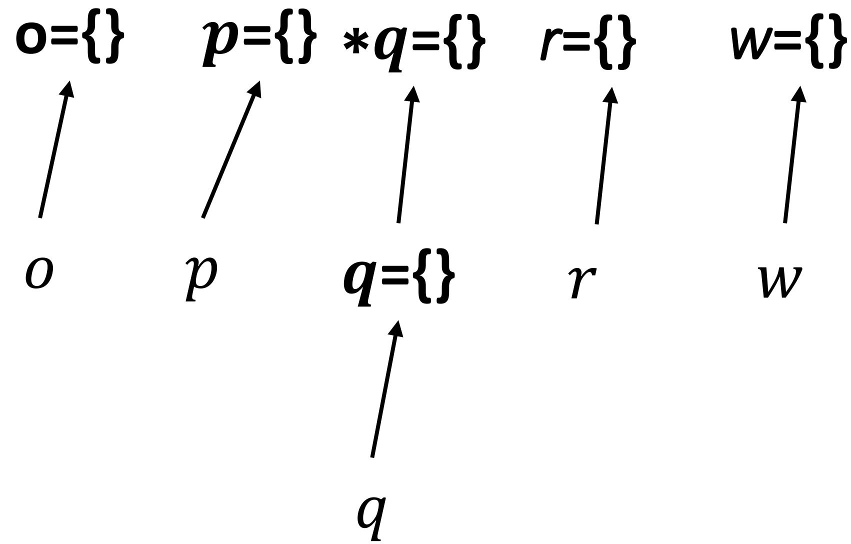
- Steensgaard约束
  - $\mathrm{v} \in \mathbf{o}$
  - $\mathrm{w} \in \mathbf{w}$
  - $\mathrm{p} \in \mathbf{q}$
  - $\mathrm{r} \in \mathbf{q}$
  - ***q=p**
  - **w=*q**
  - **p=o**
  - $\forall y. \forall x \in \boldsymbol{y}. \boldsymbol{x} =_* \boldsymbol{y}$

通用约束，用于传递相等关系

- 赋值使得左右两边的集合相等
- 最后一条约束使得相等指针的后继也相等
- 因为集合相等，所以可以合并成一个集合

合并操作执行方法

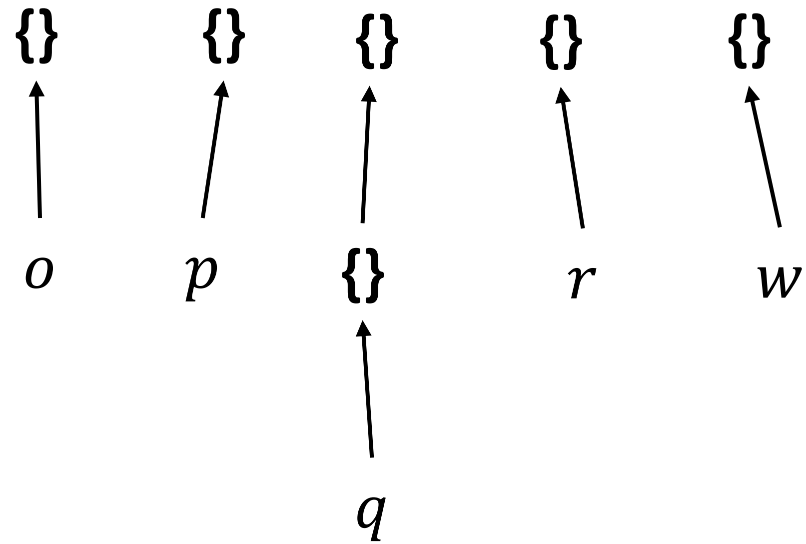
- v ∈ **o**
- w ∈ **w**
- p ∈ **q**
- r ∈ **q**
- ***q=p**
- **w**=***q**
- **p=o**
- $\forall y.\forall x \in \boldsymbol{y}.\boldsymbol{x} =* \boldsymbol{y}$



- 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

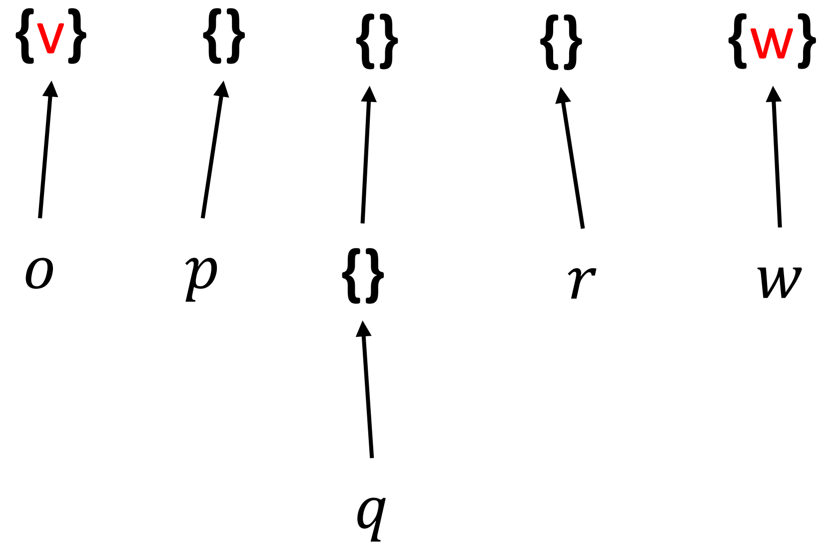
- v ∈ **o**
- w ∈ **w**
- p ∈ **q**
- r ∈ **q**
- ***q=p**
- **w=*q**
- **p=o**
- $\forall y. \forall x \in \boldsymbol{y}. \boldsymbol{x} =* \boldsymbol{y}$



- 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

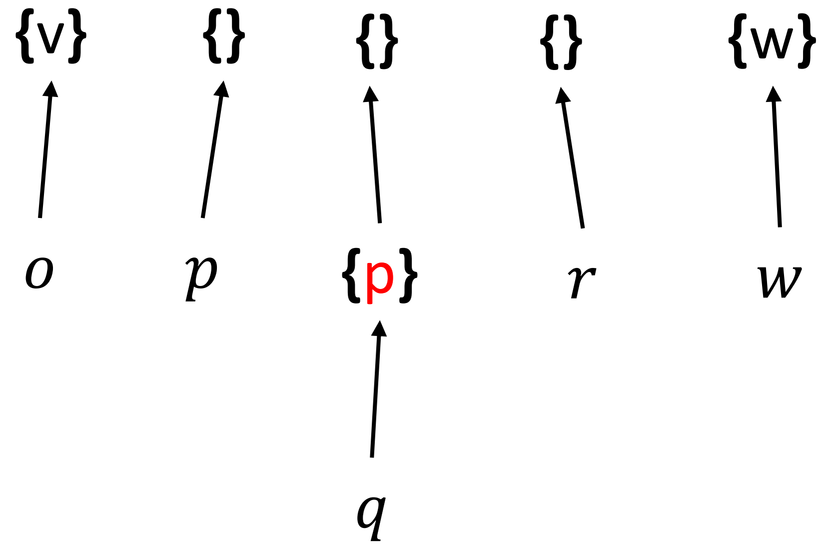
- $v \in \mathbf{o}$
- $w \in \mathbf{w}$
- $p \in \mathbf{q}$
- $r \in \mathbf{q}$
- ***q=p**
- **w=*q**
- **p=o**
- $\forall y. \forall x \in \boldsymbol{y}. \boldsymbol{x} =* \boldsymbol{y}$



- 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

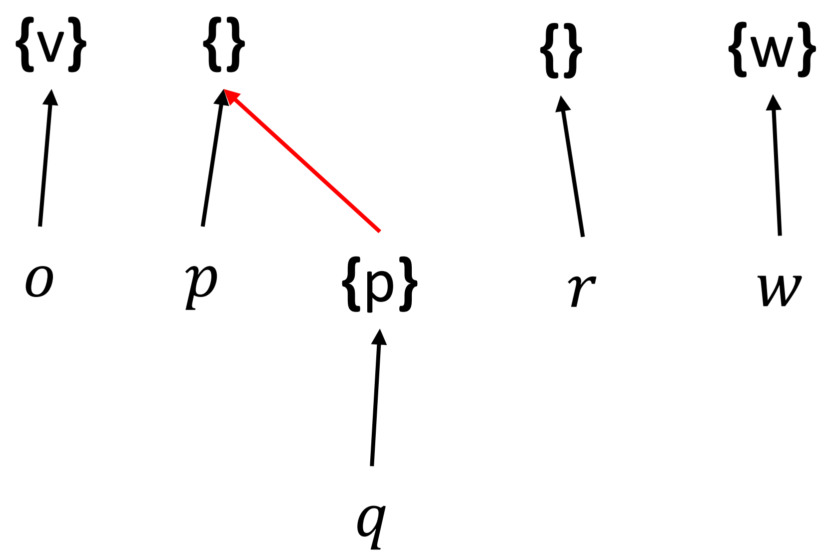
- v ∈ **o**
- w ∈ **w**
- p ∈ **q**
- r ∈ **q**
- ***q=p**
- **w=*q**
- **p=o**
- $\forall y. \forall x \in \boldsymbol{y}. \boldsymbol{x} =* \boldsymbol{y}$



- 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

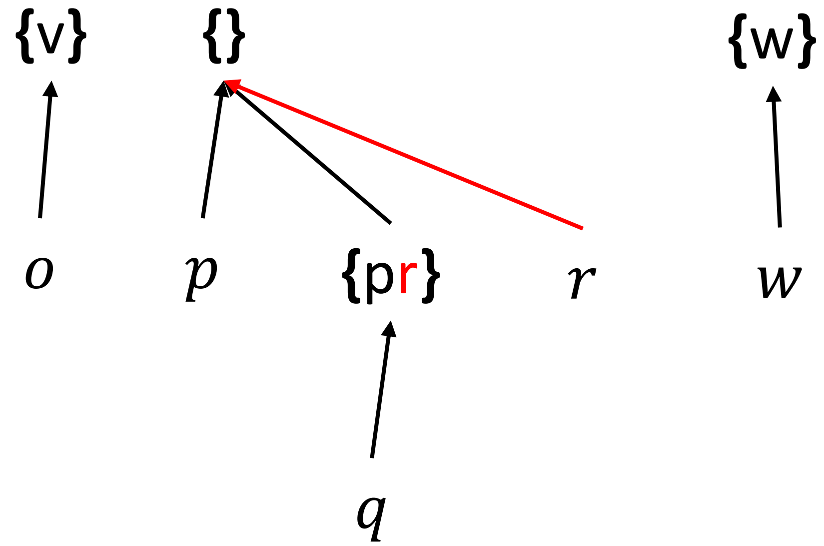
- v ∈ **o**
- w ∈ **w**
- p ∈ **q**
- r ∈ **q**
- ***q=p**
- **w=*q**
- **p=o**
- $\forall y. \forall x \in \boldsymbol{y}. \boldsymbol{x} =* \boldsymbol{y}$



- 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

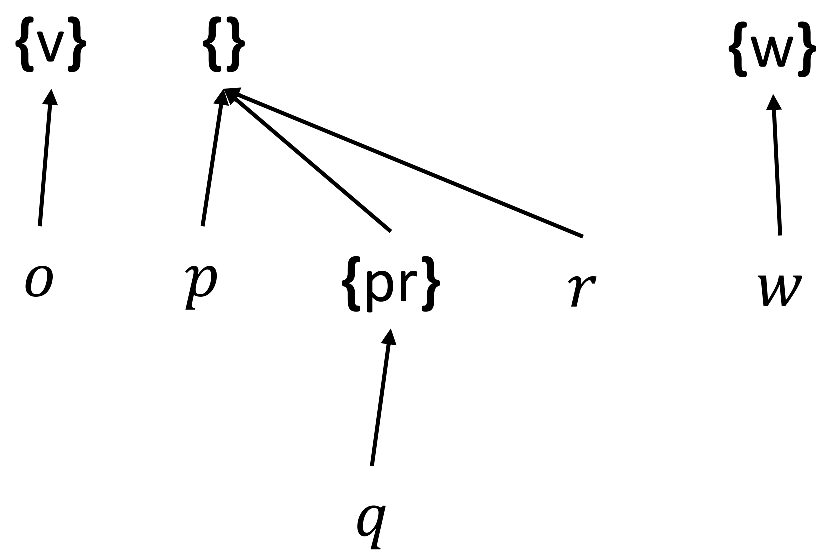
- v ∈ **o**
- w ∈ **w**
- p ∈ **q**
- r ∈ **q**
- ***q=p**
- **w=*q**
- **p=o**
- $\forall y. \forall x \in \boldsymbol{y}. \boldsymbol{x} =* \boldsymbol{y}$



- 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

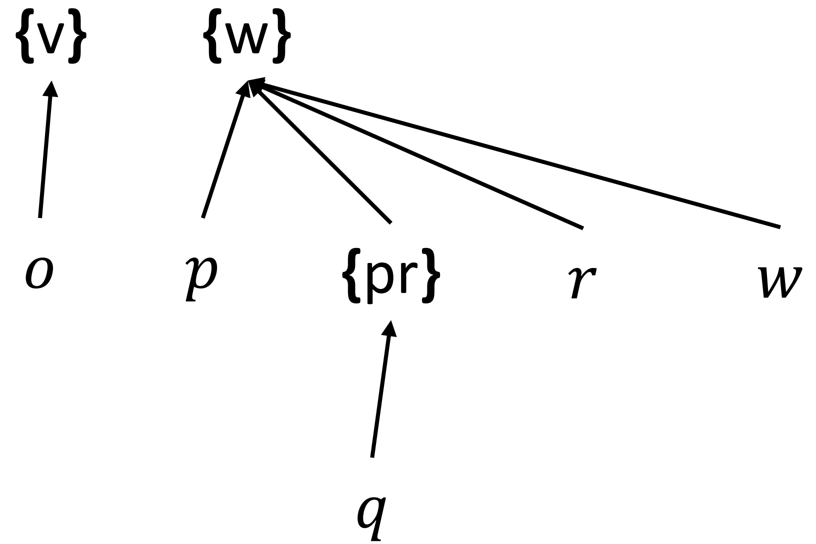
- v ∈ **o**
- w ∈ **w**
- p ∈ **q**
- r ∈ **q**
- ***q=p***
- **w=*q**
- **p=o**
- $\forall y. \forall x \in \boldsymbol{y}. \boldsymbol{x} =* \boldsymbol{y}$



- 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

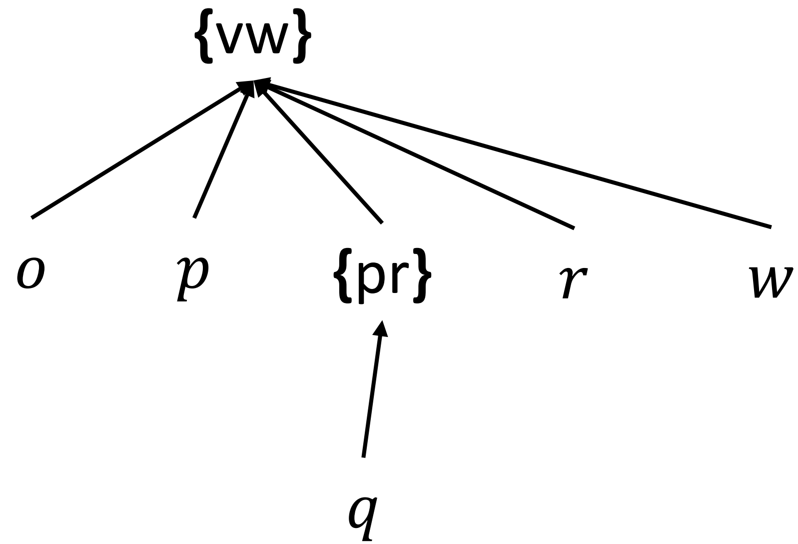
- $v \in \mathbf{o}$
- $w \in \mathbf{w}$
- $p \in \mathbf{q}$
- $r \in \mathbf{q}$
- ***q=p**
- **w=*q**
- **p=o**
- $\forall y. \forall x \in \boldsymbol{y}. \boldsymbol{x} =* \boldsymbol{y}$



- 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 ∈ **o**
- w ∈ **w**
- p ∈ **q**
- r ∈ **q**
- ***q=p**
- **w=*q**
- **p=o**
- $\forall y. \forall x \in \boldsymbol{y}. \boldsymbol{x} =* \boldsymbol{y}$



- 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

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$O(n)$
- 通过并查集实现合并，单次合并的开销为$O(\alpha(n))$

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

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

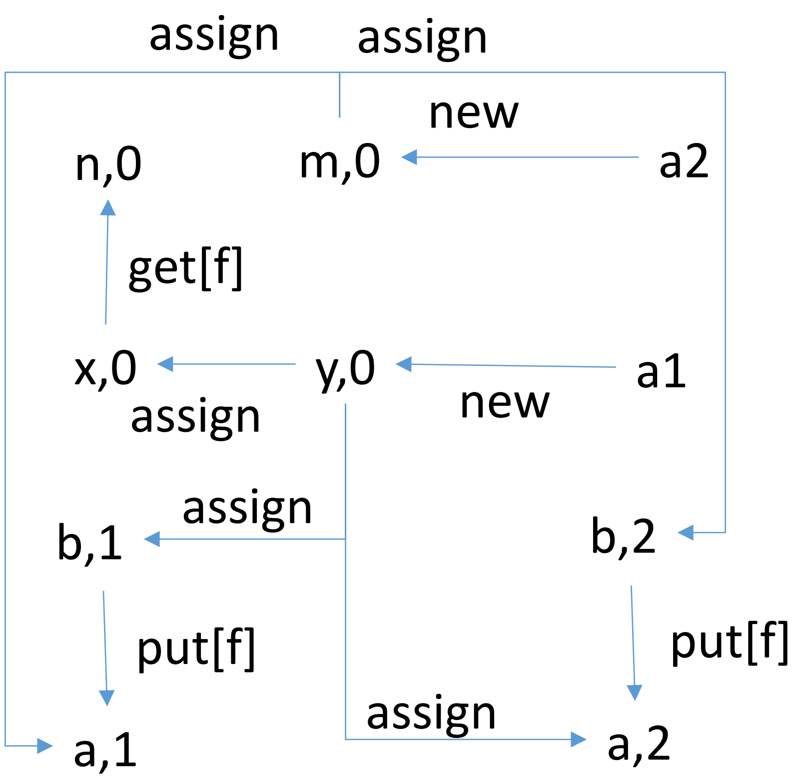
解决方法

- 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

降低上下文敏感性

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

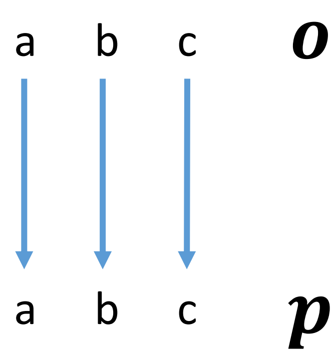
```
Main(): //0
y = new A();//a1
m=new A();//a2
SetF(m, y); //1
x=y;
SetF(y, m); //2
n=x.f;

SetF(a, b):
a.f=b;
```



 如果不克隆方法，会认为n指向a1

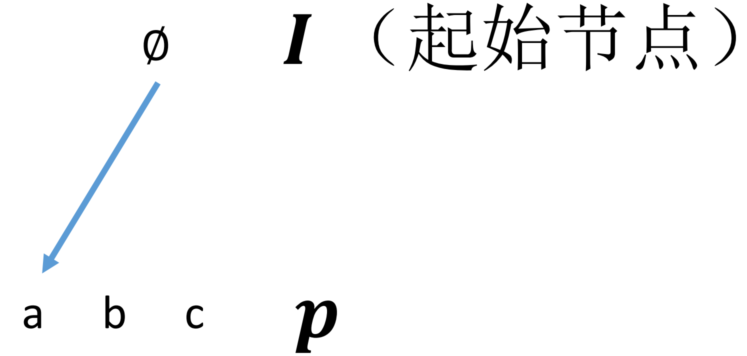
降低域敏感性

- 思路：转换成CFL可达性问题来进行精确的上下文敏感分析
- 对于约束$\boldsymbol{p} \supseteq \boldsymbol{o}$
- 化为方程$\boldsymbol{p} = \boldsymbol{p} \cup \boldsymbol{o}$
- 即如下可达性图



初始值

- 对于约束$\boldsymbol{p} \supseteq \{\mathrm{a}\}$
- 化为方程$\boldsymbol{p} = \boldsymbol{p} \cup \{\mathrm{a}\}$
- 即如下可达性图



降低域敏感性

- 但是，原分析中还有全称量词
  - $\forall x \in \boldsymbol{a},\ \ \boldsymbol{x.next} \supseteq \boldsymbol{b}$
- 这类约束无法直接转换成CFL可达性的图表示
  - 需要在图上动态加边
- 通过降低域敏感性来去掉全称量词

复习：域非敏感分析

```
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
- 约束中不会出现全称量词

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=b产生
  - **a ⊇ b**
  - **a->next⊇b->next**
  - **b->next⊇a->next**
- a->next=b产生
  - **a->next ⊇ b**
  - **a->next⊇b->next**
  - **b->next⊇a->next**
- a=b->next产生
  - **a ⊇ b->next**
  - **a->next⊇b->next**
  - **b->next⊇a->next**

为什么两个方向都有？

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 ⊇ b**
  - **a->next->next⊇b->next**
  - **a->next->next⊆b->next**
  - **a->next->next⊇b->next->next**
  - **a->next->next⊆b->next->next**
- a=b->next产生
  - **a ⊇ b->next**
  - **a->next⊇b->next->next**
  - **a->next⊆b->next->next**
  - **a->next->next⊇b->next->next**
  - **a->next->next⊆b->next->next**

域敏感性 vs 上下文敏感性

- 目前学术界还缺乏对两者权衡的详细比较
  - 降低上下文敏感性和降低域敏感性的研究各自独立发展，需要进一步工作进行统一
- 降低上下文敏感性的工作
  - 通常基于CFL可达性的指向分析，通过引入新的表示和算法来尽量精确的进行上下文敏感分析
  - 如：用正则文法来模拟上下文敏感性所需的上下文无关文法
  - 参考：Manu Sridharan, Rastislav Bodík: Refinement-based context-sensitive points-to analysis for Java. PLDI 2006: 387-400
- 降低域敏感性的工作
  - 通常基于Anderson指向分析，通过引入新的表示和算法来尽量精确的进行域敏感分析
  - 如：引入*表示任意长的访问序列
  - 参考：Johannes Lerch, Johannes Späth, Eric Bodden, Mira Mezini:
  - Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths (T). ASE 2015: 619-629
- 主流框架中更多采用精确域敏感性，用克隆实现上下文敏感性的方法

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

分析结果i.m()调用A.m()

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λ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
  - $\forall y \in \mathbf{f\#y}. \forall m \in \mathrm{targets}(y, g),$
    $\mathbf{f\#x} \supseteq \mathbf{m\#ret}$
    $\mathbf{m\#this} \supseteq \mathrm{filter}\ (\mathbf{f\#y}, \mathrm{declared}(m))$
    $\mathbf{m\#a} \supseteq \mathbf{f\#a}$
    $\mathbf{m\#b} \supseteq \mathbf{f\#b}$
- 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** ⊒{3}
- $\forall i$ ∈ **main#i**, $\forall m$ ∈ targets(i, f),
  - **main#i⊒m#ret**
  - $\mathbf{m\#this} \supseteq \text{filter}(\mathbf{main\#i}, \text{declared}(\text{m}))$
- $\forall i$ ∈ **main#i**, $\forall m$ ∈ targets(i, f),
  - **main#x⊒m#ret**
  - $\mathbf{mthis} \supseteq \text{filter}\left(\mathbf{main\#i}, \text{declared}(\text{m})\right)$
- **A.f#ret** ⊒{1}
- **B.f#ret** ⊒{2}
-
- 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是上下文敏感而流非敏感的

作业

给定下面的程序，假设域展开一次，请画出可达性图。

```
Struct Node {
  int value;
  Node* next;
};
a = malloc(); //1
b = malloc(); //2
a->next = b;
b->next = malloc(); //3
```

参考文献

- 基于CFA的指向分析：
  - Thomas W. Reps: Program Analysis via Graph Reachability. ILPS 1997: 5-19
- Steensgaard分析：
  - Lecture Notes: Pointer Analysis
    - Jonathan Aldrich
    - https://www.cs.cmu.edu/~aldrich/courses/15-819O-13sp/resources/pointer.pdf
- 控制流分析：
  - Lecture Notes: Object-Oriented Call Graph Construction
    - Jonathan Aldrich
    - https://www.cs.cmu.edu/~aldrich/courses/15-819O-13sp/resources/callgraph.pdf
- 基于展开域的指针分析：
  - Neil D. Jones, Steven S. Muchnick: Flow Analysis and Optimization of Lisp-Like Structures. POPL 1979: 244-256