软件分析

流非敏感分析
指向分析

熊英飞
北京大学

2015

程序分析的分类-敏感性

- 一般而言，抽象过程中考虑的信息越多，程序分析的精度就越高，但分析的速度就越慢
- 程序分析中考虑的信息通常用敏感性来表示
  - 流敏感性flow-sensitivity
  - 路径敏感性path-sensitivity
  - 上下文敏感性context-sensitivity
  - 字段敏感性field-sensitivity

术语-流敏感(flow-sensitivity)

- 流非敏感分析（flow-insensitive analysis）：如果把程序中语句随意交换位置（即：改变控制流），如果分析结果始终不变，则该分析为流非敏感分析。
- 流敏感分析（flow-sensitive analysis）：其他情况

- 数据流分析通常为流敏感的

流非敏感分析示例

- 流非敏感的符号分析
  - 收集程序中的所有赋值语句
  - 使用一个变量符号值的集合
  - 反复计算所有赋值语句更新该集合
- 流非敏感的活跃变量分析
  - 对于整个程序产生一个集合，只要程序中有读取变量v的语句，就将其加入集合

时间空间复杂度

- 活跃变量分析：语句数为n，程序中变量个数为m，使用bitvector表示集合
- 流非敏感的活跃变量：每条语句的操作时间为O(m)，因此时间复杂度上界为O(nm)，空间复杂度上界为O(m)
- 流敏感的活跃变量分析：格的高度为O(m)，转移函数、交汇运算和比较运算都是O(m)，时间复杂度上界为O(nm^2)，空间复杂度上界为O(nm)

- 对于特定分析，流非敏感分析能到达很快的处理速度和可接受的精度（如基于SSA的指针分析）

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

Microsoft Research 微软亚洲研究院

另一种视角：方程求解

- 数据流分析的传递函数和⊓操作定义了一组方程
  - $D_{v_1} = F_{v_1}(D_{v_1}, D_{v_2}, \ldots, D_{v_n})$
  - $D_{v_2} = F_{v_2}(D_{v_1}, D_{v_2}, \ldots, D_{v_n})$
  - …
  - $D_{v_n} = F_{v_n}(D_{v_1}, D_{v_2}, \ldots, D_{v_n})$
- 其中
  - $F_{v_i}(D_{v_1}, D_{v_2}, \ldots, D_{v_n}) = f_{v_i}(I)$
  - $F_{v_i}(D_{v_1}, D_{v_2}, \ldots, D_{v_n}) = f_{v_i}\left(\sqcap_{j \in pred(i)} D_{v_j}\right)$
- 数据流分析即为求解该方程的最大解
  - 传递函数和⊓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

方程组求解算法

- 在数理逻辑学中，该类算法称为Unification算法
  - 参考：http://en.wikipedia.org/wiki/Unification_(computer_science)
- 对于单调函数和有限格，标准的Unification算法就是我们学到的数据流分析算法
  - 从$(I, \top, \top, \ldots, \top)$开始反复应用$F_{v_1}$到$F_{v_n}$，直到达到不动点
  - 增量优化：每次只执行受到影响的F_{v_i}

方程组求解视角的意义

- 如果我们能把分析的安全性表达为数据的约束，我们就把原问题转换成寻找合适的unification算法求解方程组的问题
- 根据分析需要，我们可以用合适的unification算法获得最大解或者最小解
- 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
    - $D_{v_1} = D_{v_1} \sqcap F_{v_1}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
    - $D_{v_2} = D_{v_2} \sqcap F_{v_2}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
    - …
    - $D_{v_n} = D_{v_n} \sqcap F_{v_n}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
- 很多问题可以转成上述形式的不等式

Anderson指向分析算法

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	$*a \supseteq b$	$\forall v \in pts(a).\ pts(v) \supseteq pts(b)$

注意：$\boldsymbol{a}$等价于pts(a)，a等价于loc(a)

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

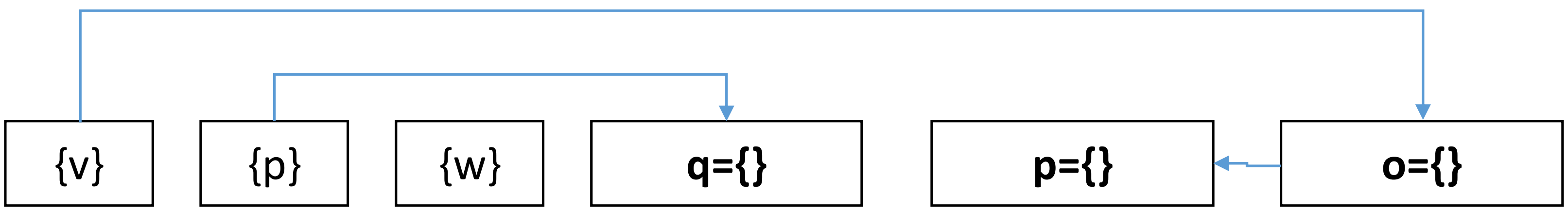
约束求解方法—通用框架

- 将约束
  - $\boldsymbol{o} \supseteq \{v\}$
  - $\boldsymbol{q} \supseteq \{p\}$
  - $\forall v \in \boldsymbol{q}.\boldsymbol{p} \supseteq \boldsymbol{v}$
  - $\boldsymbol{p} \supseteq \boldsymbol{o}$
  - $\forall v \in \boldsymbol{q}.\boldsymbol{v} \supseteq \{w\}$
- 转换成标准形式
  - $\boldsymbol{p} = \boldsymbol{p} \cup \boldsymbol{o} \cup \left(\bigcup_{v\in\boldsymbol{q}} \boldsymbol{v}\right) \cup (p \in \boldsymbol{q}\ ?\ \{w\} : \emptyset)$
  - $\boldsymbol{q} = \boldsymbol{q} \cup \{p\} \cup (q \in \boldsymbol{q}\ ?\ \{w\} : \emptyset)$
  - $\boldsymbol{o} = \boldsymbol{o} \cup \{v\} \cup (o \in \boldsymbol{q}\ ?\ \{w\} : \emptyset)$

  - 等号右边都是递增函数

约束求解方法—增量算法

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

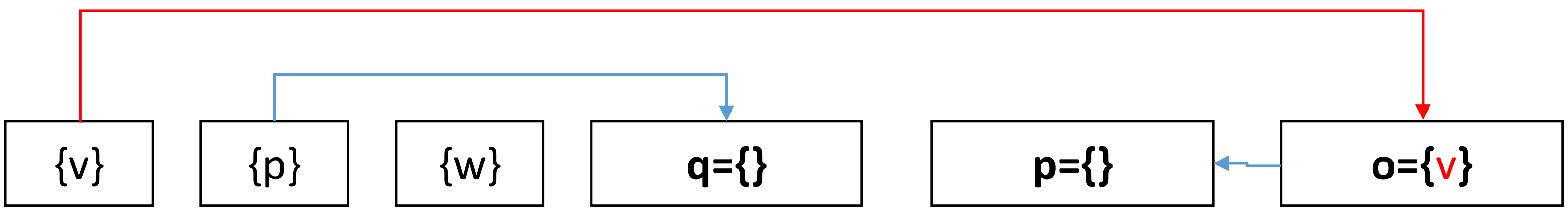


$$\forall v \in \boldsymbol{q}.\boldsymbol{p} \supseteq \boldsymbol{v}$$

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

约束求解方法—增量算法

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

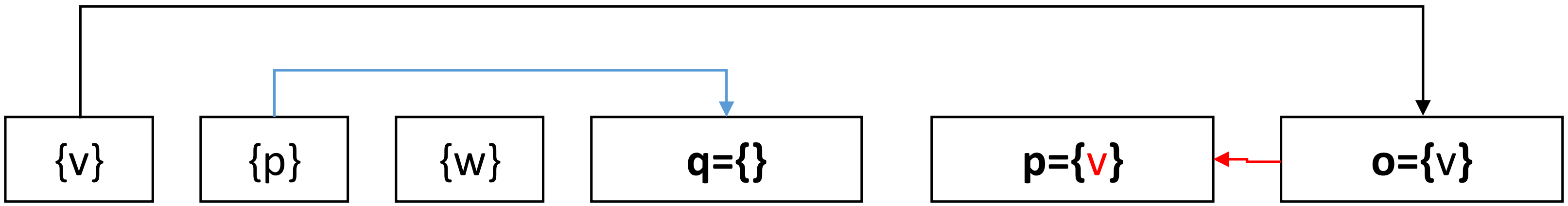


$$\forall v \in \boldsymbol{q}.\boldsymbol{p} \supseteq \boldsymbol{v}$$

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

约束求解方法—增量算法

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

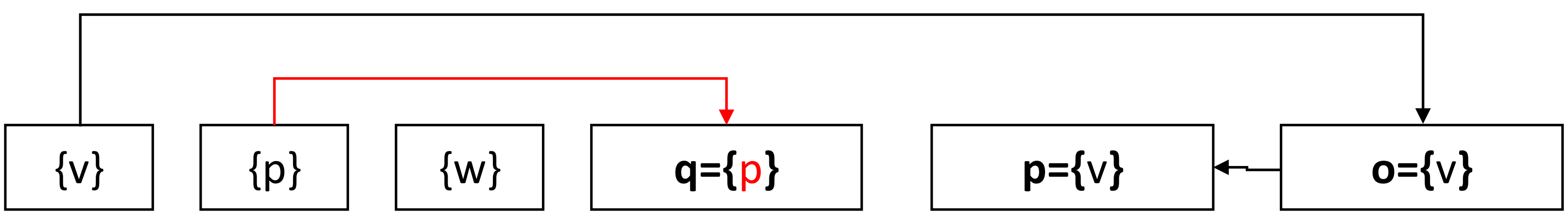


$\forall v \in \boldsymbol{q}.\boldsymbol{p} \supseteq \boldsymbol{v}$

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

约束求解方法—增量算法

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

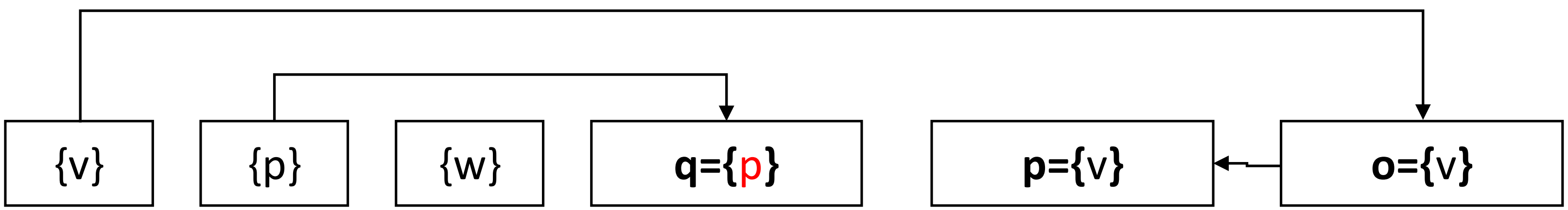


$$\forall v \in \boldsymbol{q}.\boldsymbol{p} \supseteq \boldsymbol{v}$$

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

约束求解方法—增量算法

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

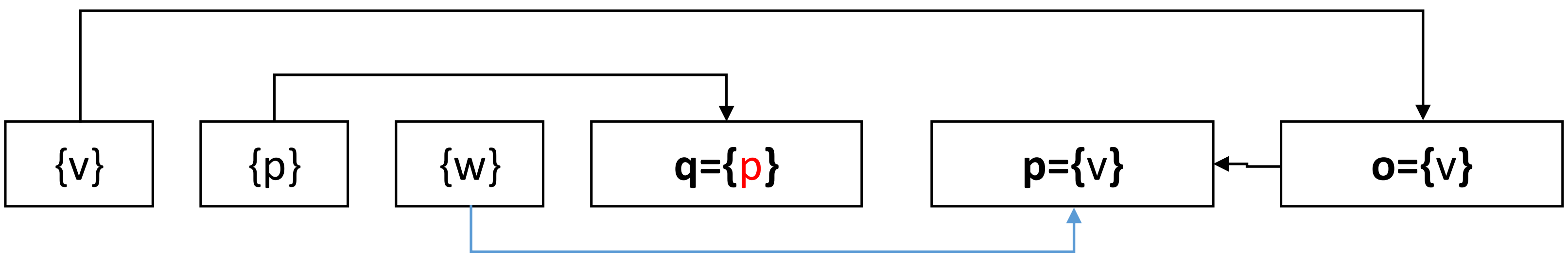


$\forall v \in \boldsymbol{q}.\boldsymbol{p} \supseteq \boldsymbol{v}$

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

约束求解方法—增量算法

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

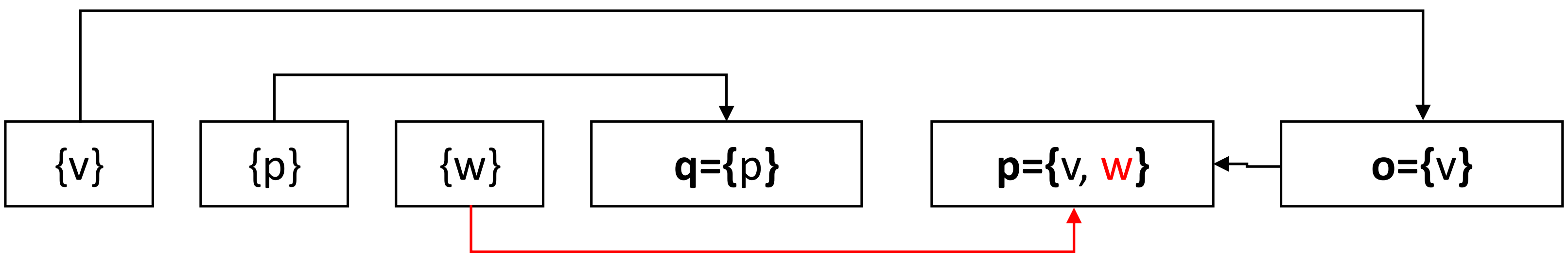


$\forall v \in \boldsymbol{q}.\boldsymbol{p} \supseteq \boldsymbol{v}$

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

约束求解方法—增量算法

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



$$\forall v \in \boldsymbol{q}.\boldsymbol{p} \supseteq \boldsymbol{v}$$

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

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
- 参考论文：
  - The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code, Hardekopf and Lin, PLDI 2007

堆上分配的内存

- a=malloc();
- malloc()语句每次执行创造一个内存位置
- 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
- 猜一猜应该如何做？

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Nodel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
  - $\forall x \in \boldsymbol{a},\ \ x.next \supseteq b$

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$O\big(n\alpha(n)\big)$，接近线性时间。
  - n为程序中的语句数量。
  - α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

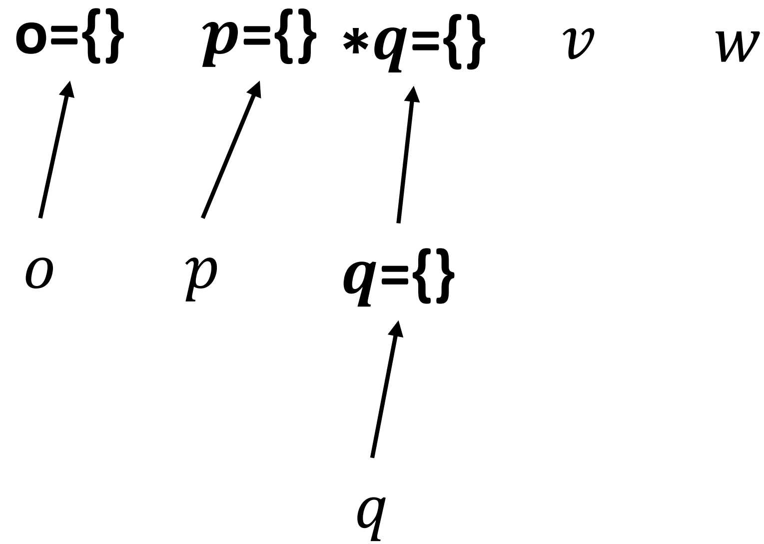
```
o=&v;
q=&p;
if (a > b) {
  p=*q;
  p=o; }
*q=&w;
```

- 产生合并操作
  - Union($\boldsymbol{o}, v$)
  - Union(**q**, p)
  - Union(**p**, ***q**)
  - Union(**p**, **o**)
  - Union(***q**, w)

Microsoft Research 微软亚洲研究院

合并操作执行方法

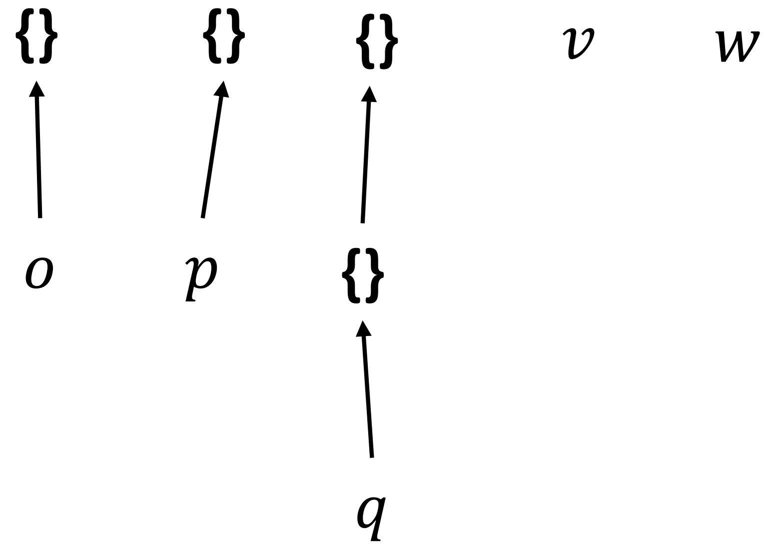
- Union($\boldsymbol{o}, v$)
- Union(**q**, p)
- Union(**p**, ***q**)
- Union(**p**, **o**)
- Union(***q**, w)



- 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

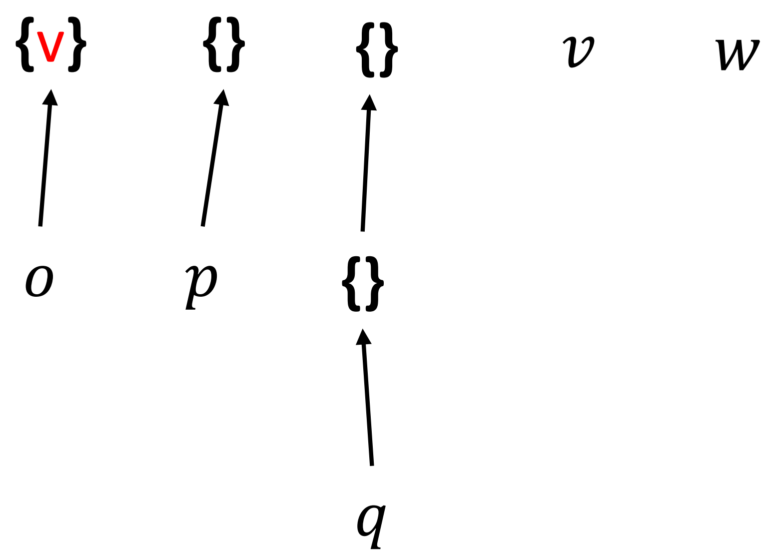
- Union($\boldsymbol{o}, v$)
- Union(**q**, p)
- Union(**p**, ***q**)
- Union(**p**, **o**)
- Union(***q**, w)



- 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

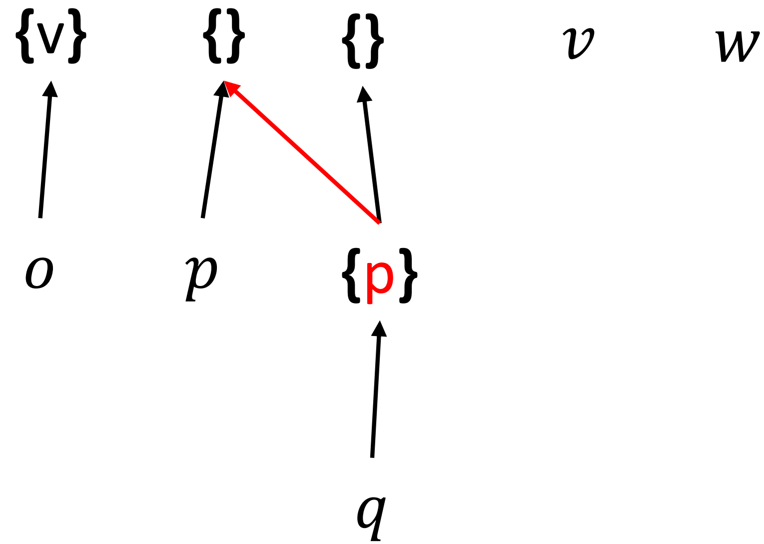
- Union($\boldsymbol{o}, v$)
- Union(**q**, p)
- Union(**p**, ***q**)
- Union(**p**, **o**)
- Union(***q**, w)



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

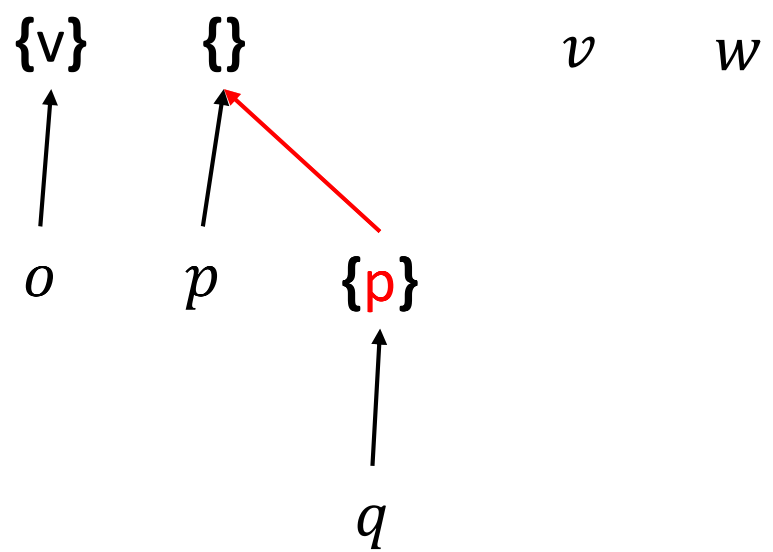
- Union($\boldsymbol{o}$, v)
- Union(**q**, p)
- Union(**p**, ***q**)
- Union(**p**, **o**)
- Union(***q**, w)



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

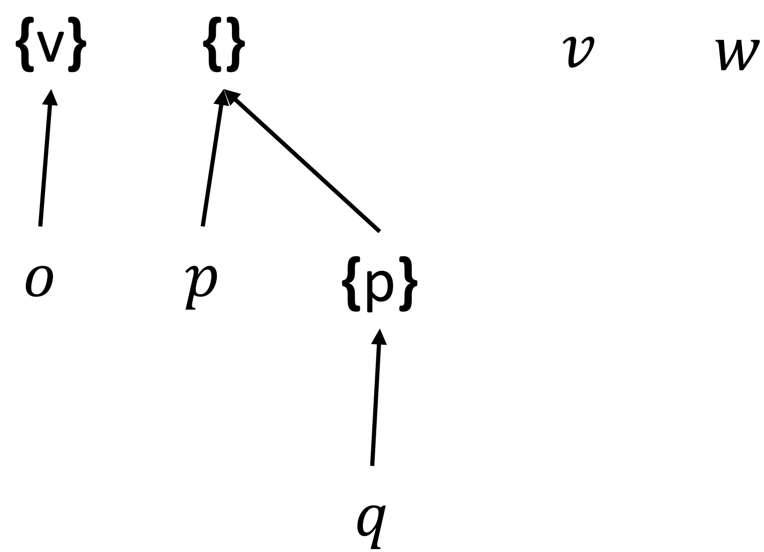
- Union($\boldsymbol{o}, v$)
- Union(**q**, p)
- Union(**p**, ***q**)
- Union(**p**, **o**)
- Union(***q**, w)



继续合并后继

合并操作执行方法

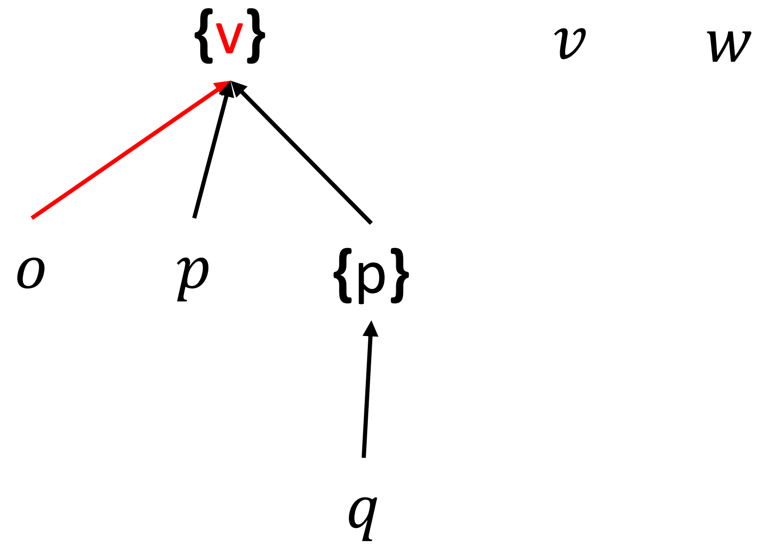
- Union($\boldsymbol{o}, v$)
- Union(**q**, p)
- Union(**p**, *q)
- Union(**p**, **o**)
- Union(*q, w)



对于集合的合并，直接合并两个集合

合并操作执行方法

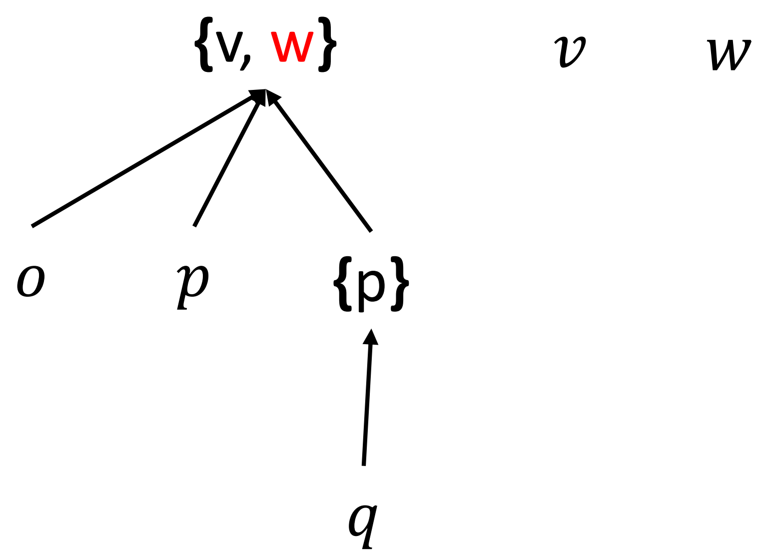
- Union($\boldsymbol{o}, v$)
- Union(**q**, p)
- Union(**p**, ***q**)
- Union(**p**, **o**)
- Union(***q**, w)



对于集合的合并，直接合并两个集合

合并操作执行方法

- Union($\boldsymbol{o}$, v)
- Union(**q**, p)
- Union(**p**, ***q**)
- Union(**p**, **o**)
- Union(***q**, w)



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

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

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pts(a)=pts(b)且|pts(a)|=1，则a和b为must aliases
  - 如果pts(a)∩pts(b)=∅，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

作业

- LLVM本身带有指针分析DSA，查找资料并回答以下问题
  - 该算法是Anderson风格， Steensgaard风格，还是两者都不是？
  - 该算法是否是flow-sensitive的？
  - 该算法是否是field-sensitive的？