



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

复习： Anderson指向分析算法



$o = \&v;$

- 产生约束

$q = \{\}$

$w = \&w;$

- $o \supseteq \{v\}$
- $w \supseteq \{w\}$
- $q \supseteq \{p\}$
- $q \supseteq \{r\}$

$p = \{\}$

$q = \&p;$

if ($a > b$) {

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

$o = \{\}$

$q = \&r;$

$*q = p;$

$w = *q;$

$p = o; \}$

$w = \{\}$

$r = \{\}$

复习： Anderson指向分析算法



$o = \&v;$

- 产生约束

$q = \{pr\}$

$w = \&w;$

- $o \supseteq \{v\}$
- $w \supseteq \{w\}$
- $q \supseteq \{p\}$
- $q \supseteq \{r\}$

$p = \{\}$

$q = \&p;$

if ($a > b$) {

$q = \&r;$

$*q = p;$

$w = *q;$

$p = o; }$

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

$o = \{v\}$

$w = \{w\}$

$r = \{\}$

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

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

$q = \{pr\}$

$p = \{\}$

$o = \{v\}$

$w = \{w\}$

$r = \{\}$

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

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

$q = \{pr\}$

$p = \{v\}$

$o = \{v\}$

$w = \{wv\}$

$r = \{v\}$



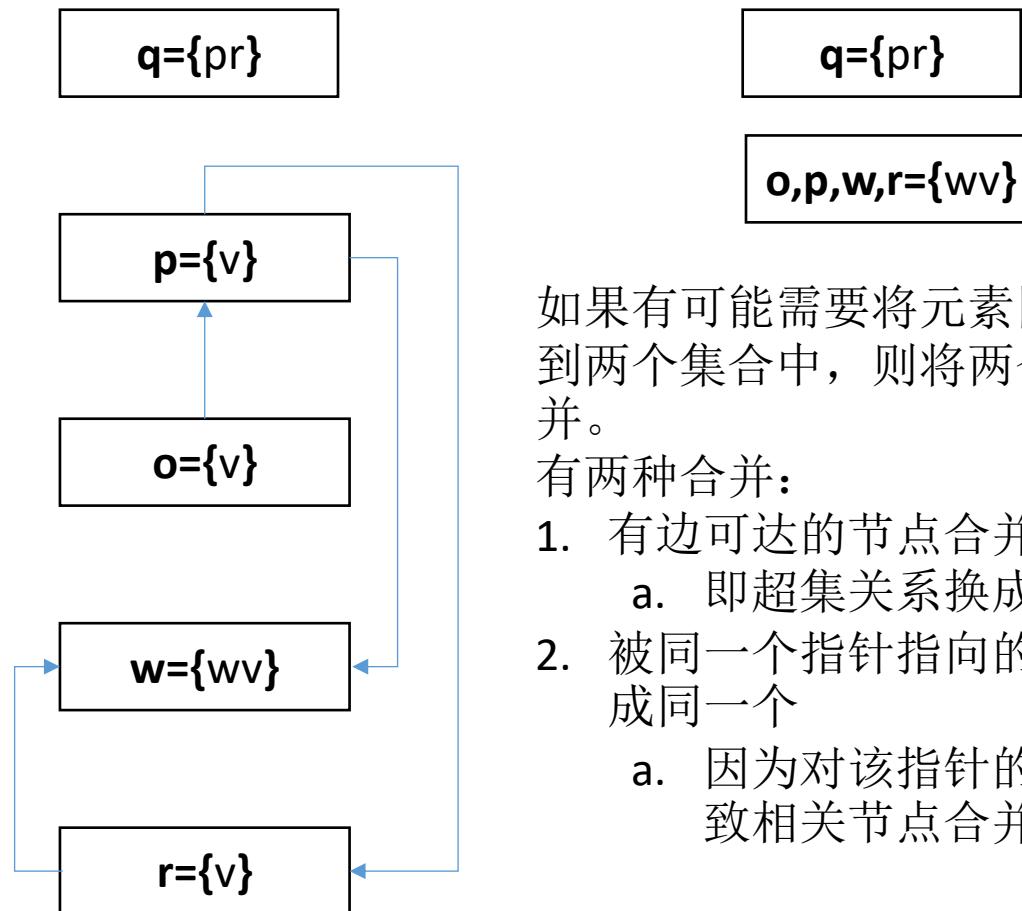
Steensgaard指向分析算法

- 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 $O(n\alpha(n))$, 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



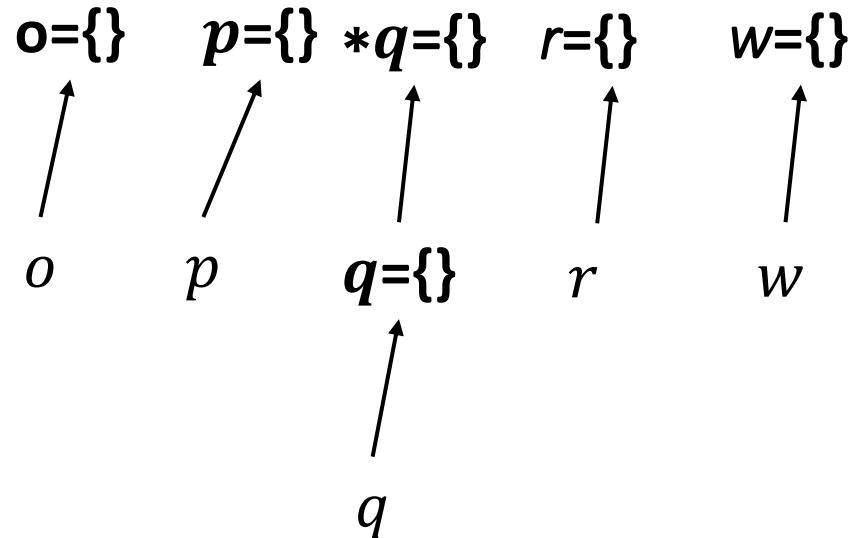
Steensgaard指向分析算法

- $o = \&v;$ • Anderson约束 • Steensgaard约束
 - $w = \&w;$ • $o \supseteq \{v\}$ • $v \in o$
 - $q = \&p;$ • $w \supseteq \{w\}$ • $w \in w$
 - $\text{if } (a > b) \{$ • $q \supseteq \{p\}$ • $p \in q$
 - $q = \&r;$ • $q \supseteq \{r\}$ • $r \in q$
 - $*q = p;$ • $\forall v \in q. v \supseteq p$ • $*q = p$
 - $w = *q;$ • $\forall v \in q. w \supseteq v$ • $w = *q$
 - $p = o; \}$ • $p \supseteq o$ • $p = o$
- 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
- 通用约束，
用于传递
相等关系



合并操作执行方法

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

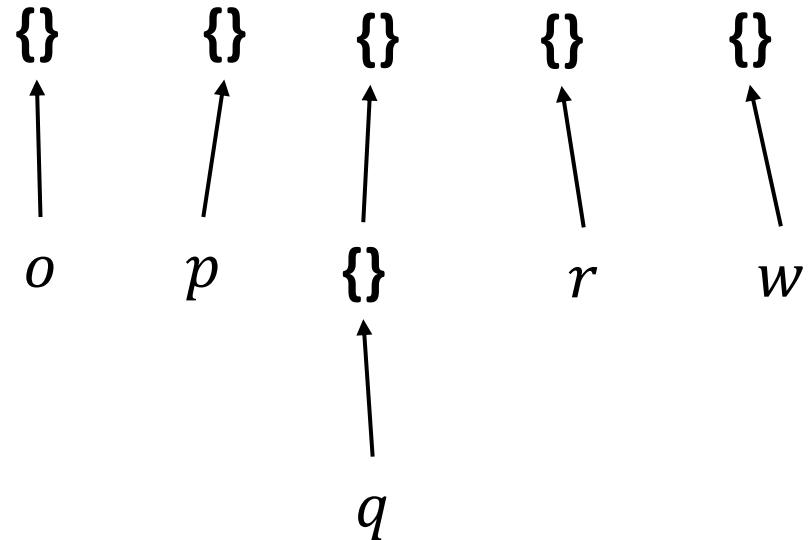


- 边表示指向关系
- 每个元素只能有一个后继
- 如果合并导致多于一个后继，就合并后继



合并操作执行方法

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

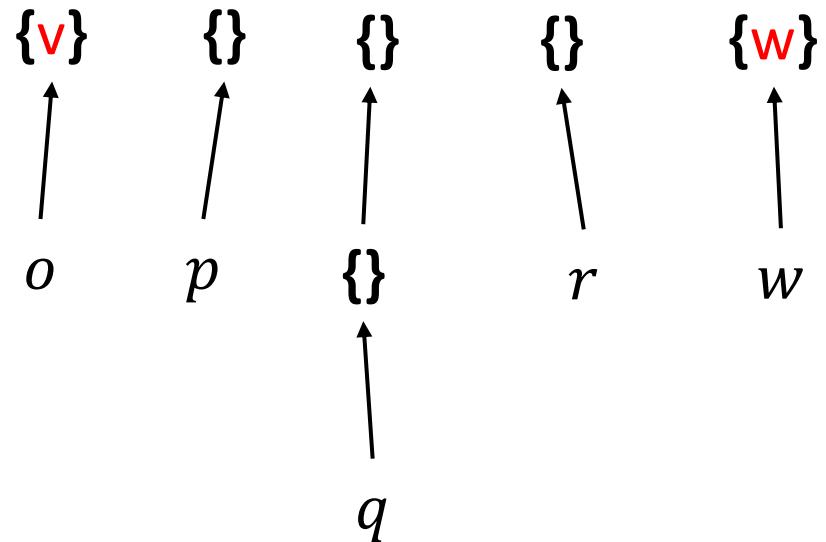


- 边表示指向关系
- 每个元素只能有一个后继
- 如果合并导致多于一个后继，就合并后继



合并操作执行方法

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

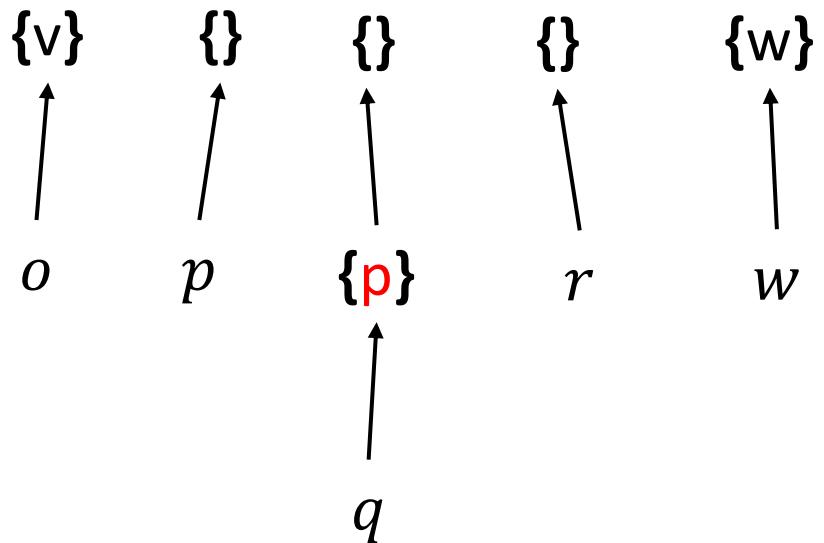


- 边表示指向关系
- 每个元素只能有一个后继
- 如果合并导致多于一个后继，就合并后继



合并操作执行方法

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

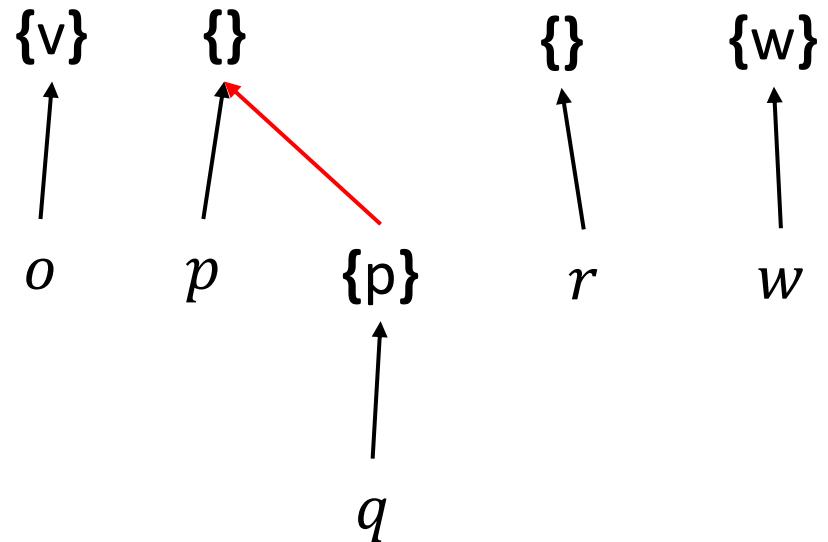


- 边表示指向关系
- 每个元素只能有一个后继
- 如果合并导致多于一个后继，就合并后继



合并操作执行方法

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

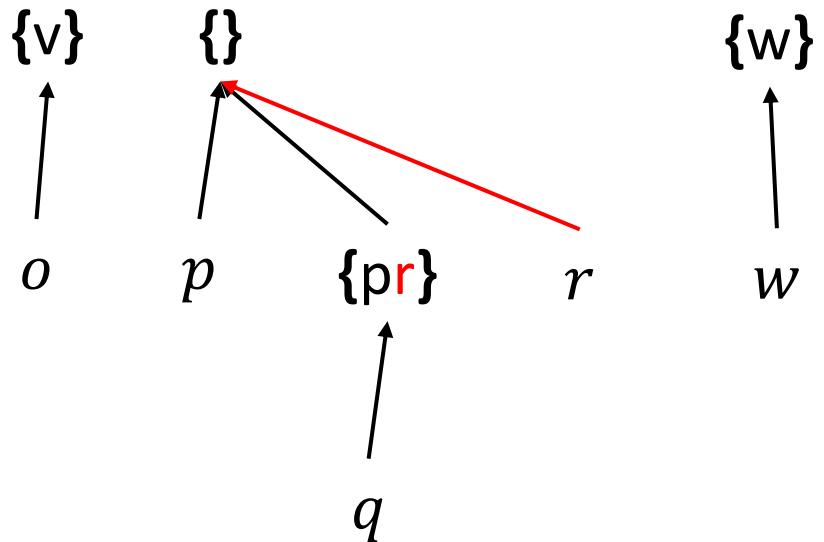


- 边表示指向关系
- 每个元素只能有一个后继
- 如果合并导致多于一个后继，就合并后继



合并操作执行方法

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

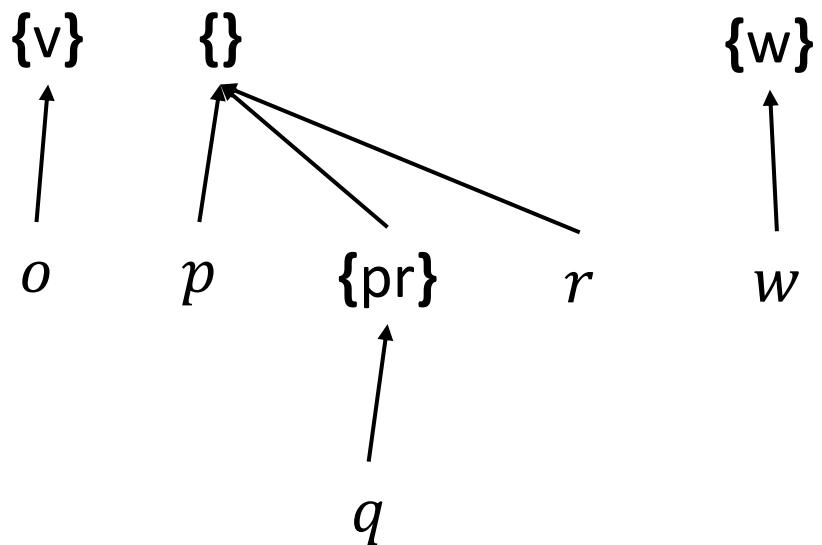


- 边表示指向关系
- 每个元素只能有一个后继
- 如果合并导致多于一个后继，就合并后继



合并操作执行方法

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

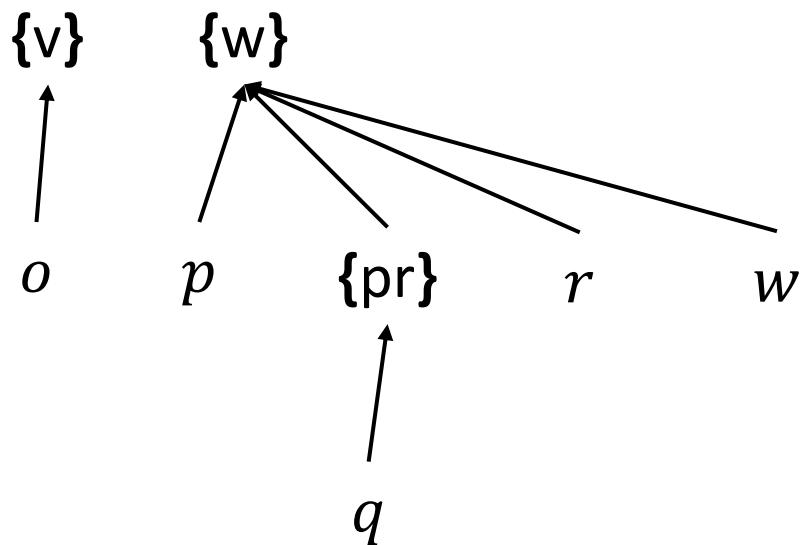


- 边表示指向关系
- 每个元素只能有一个后继
- 如果合并导致多于一个后继，就合并后继



合并操作执行方法

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

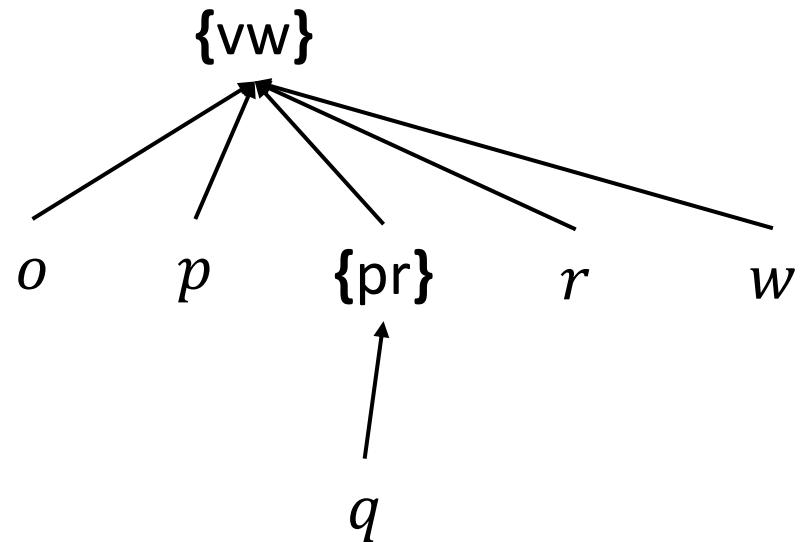


- 边表示指向关系
- 每个元素只能有一个后继
- 如果合并导致多于一个后继，就合并后继



合并操作执行方法

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



- 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
- Tom Reps等人2000年证明这是一个不可判定问题
 - 两个上下文无关语言的交集是否为空是不可判定的
 - 该问题的一个子问题P-PCP也是不可判定的
 - 上下文敏感的指针分析可以用CFL可达性求解
 - 域敏感的指针分析也可以用CFL可达性求解
 - 可以把P-PCP规约到上下文敏感和域敏感的指针分析



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

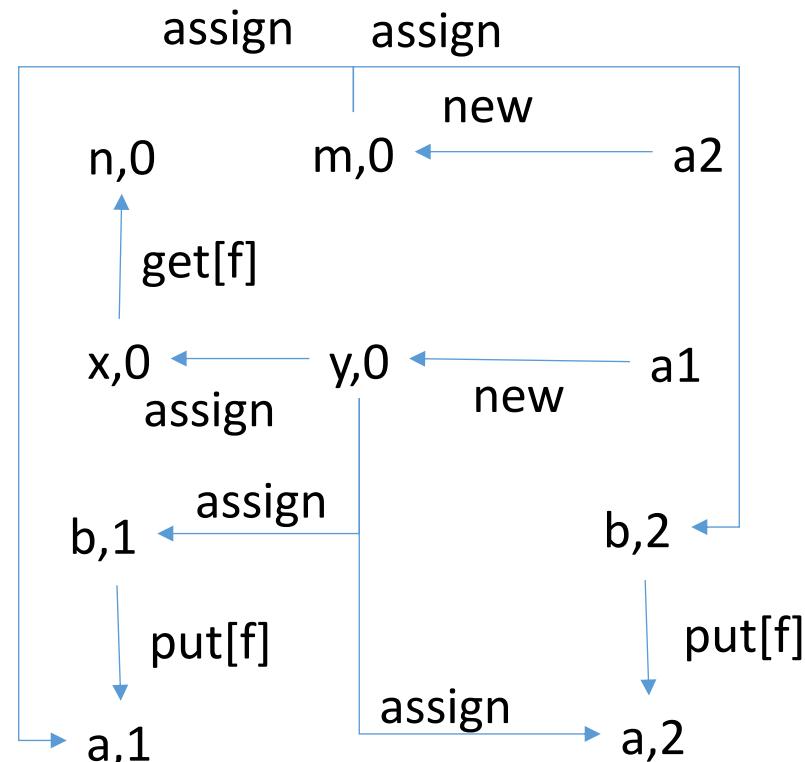
FlowTo= new (assign | put[f] Alias get[f])*

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;
```





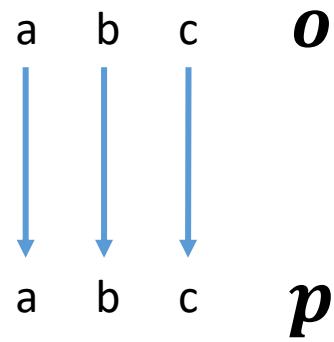
降低域敏感性

- 思路：转换成CFL可达性问题来进行精确的上下文敏感分析
- 遵循标准的多变量、流非敏感分析转换方法



约束的转换

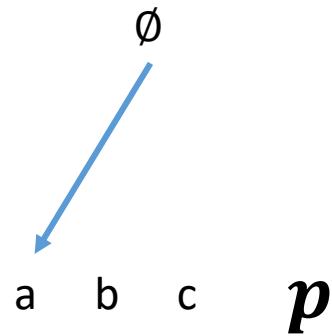
- 对于约束 $p \supseteq o$
- 形成如下可达性图





常量约束的转换

- 对于约束 $p \supseteq \{a\}$
- 按标准方法应该所有节点都连一条到 $p.a$ 的边
- 该方法等价于从空集连一条到 $p.a$ 的边





降低域敏感性

- 但是，原分析中还有全称量词
 - $\forall x \in a, x.\mathbf{next} \supseteq b$
- 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
 - $a' = \text{malloc}();$
 - $a' = b;$
 - $a' = c;$
 - 其中 a' 代表 a , $a->\text{next}$, $a->\text{prev}$
- 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
 - $a \rightarrow \text{next}$
- $a=b$ 产生
 - $a \supseteq b$
 - $a \rightarrow \text{next} \supseteq b \rightarrow \text{next}$
 - $b \rightarrow \text{next} \supseteq a \rightarrow \text{next}$
- $a \rightarrow \text{next}=b$ 产生
 - $a \rightarrow \text{next} \supseteq b$
 - $a \rightarrow \text{next} \supseteq b \rightarrow \text{next}$
 - $b \rightarrow \text{next} \supseteq a \rightarrow \text{next}$
- $a=b \rightarrow \text{next}$ 产生
 - $a \supseteq b \rightarrow \text{next}$
 - $a \rightarrow \text{next} \supseteq b \rightarrow \text{next}$
 - $b \rightarrow \text{next} \supseteq a \rightarrow \text{next}$

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;  
};  
Node* 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 \subseteq b->next
 - a->next->next \supseteq b->next->next
 - a->next->next \subseteq b->next->next
- a=b->next产生
 - a \supseteq b->next
 - a->next \supseteq b->next->next
 - a->next \subseteq b->next->next
 - a->next->next \supseteq b->next->next
 - a->next->next \subseteq 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();  
} }
```

- 只考慮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 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
 - $f() \{ \dots \}$
 - $x = y.g(a, b)$
 - $\dots \}$
- 生成约束
 - $\forall y \in f\#y. \forall m \in \text{targets}(y, g), f\#x \supseteq m\#\text{ret}$
 - $m\#\text{this} \supseteq \text{filter}(f\#y, \text{declared}(m))$
 - $m\#\text{a} \supseteq f\#\text{a}$
 - $m\#\text{b} \supseteq f\#\text{b}$
- 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** $\supseteq \{3\}$
- $\forall i \in \text{main#i}, \forall m \in \text{targets}(i, f),$
 - **main#i** $\supseteq m\#\text{ret}$
 - **m#this** \supseteq
filter(**main#i**, declared(m))
- $\forall i \in \text{main#i}, \forall m \in \text{targets}(i, f),$
 - **main#x** $\supseteq m\#\text{ret}$
 - **mthis** \supseteq filter (**main#i**, declared(m))
- **A.f#ret** $\supseteq \{1\}$
- **B.f#ret** $\supseteq \{2\}$
-
- 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



作业

给定下面的程序，进行流非敏感的指针分析，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CFL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