



软件分析

Steensgaard算法和 过程间指针分析

熊英飞
北京大学

复习：Anderson指向分析算法填空



赋值语句	约束
$a = \&b$	$a \supseteq \{b\}$
$a = b$	
$a = *b$	
$*a = b$	

复习：Anderson指向分析算法填空



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

能不能换成=?



Steensgaard指向分析算法

- Anderson算法的复杂度为 $O(n^3)$
 - n 为程序中变量的数量
- 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。
 - α 为阿克曼函数的逆
 - $\alpha(2^{132}) < 4$



Steensgaard指向分析算法

- Steensgaard算法把Anderson算法中的超集关系换成等价关系
- 通过不断合并同类项来保证算法在接近线性的时间内完成



Steensgaard指向分析算法

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

- 产生约束

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

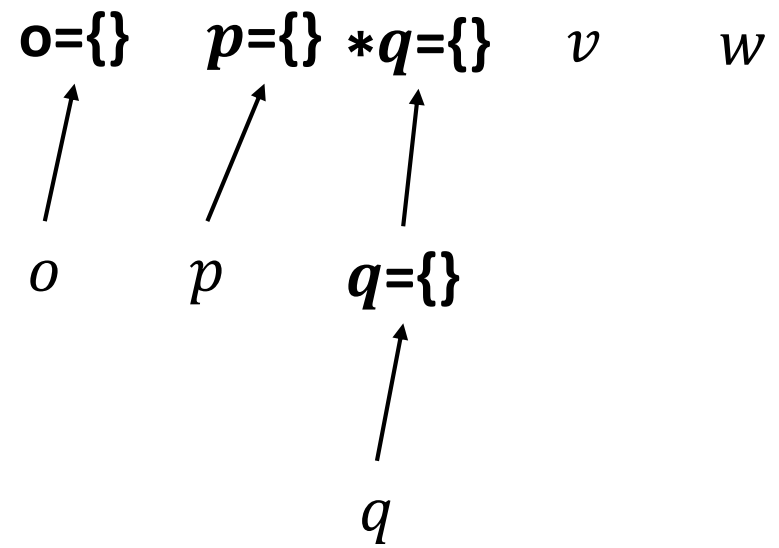
通用约束，
用于传递
相等关系

- 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

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

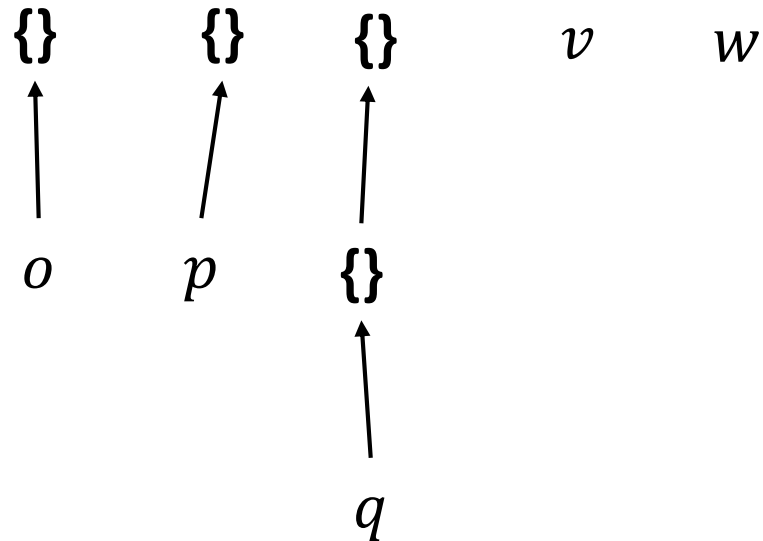


- 边表示指向关系
- 每个元素只能有一个后继
- 如果合并导致多于一个后继，就合并后继



合并操作执行方法

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

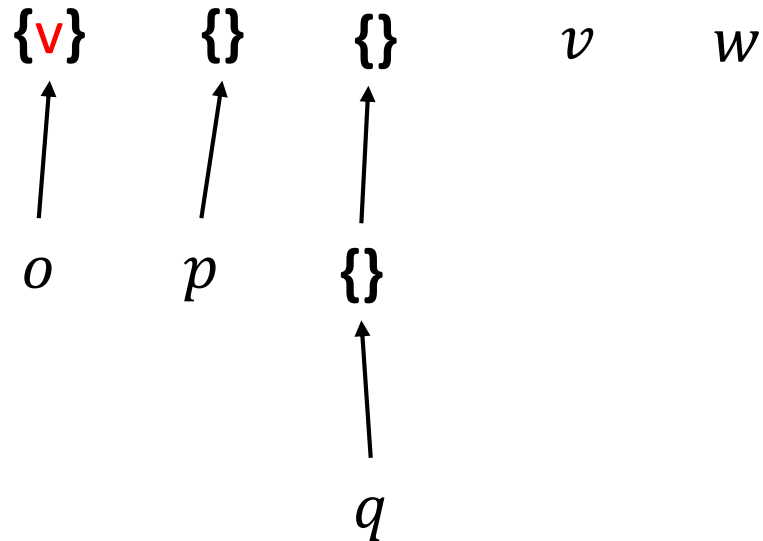


- 边表示指向关系
- 每个元素只能有一个后继
- 如果合并导致多于一个后继，就合并后继



合并操作执行方法

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

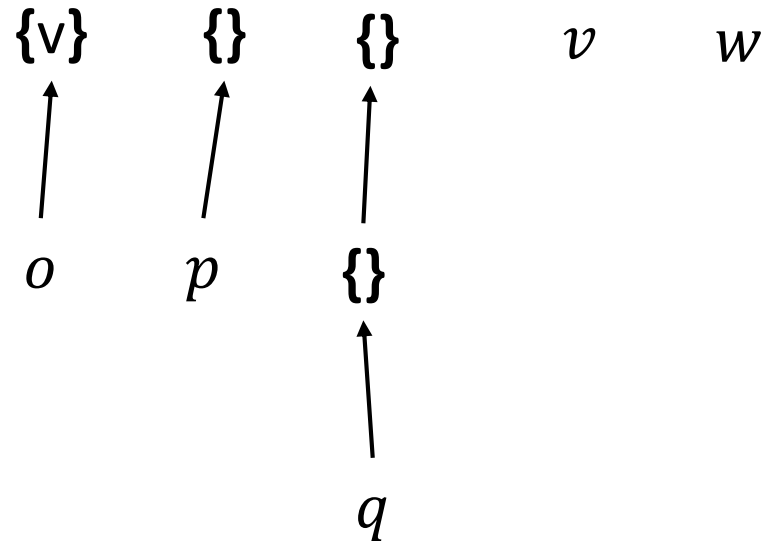


对于集合和元素的合并，添加元素到集合中，并添加元素的后继为集合的后继



合并操作执行方法

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

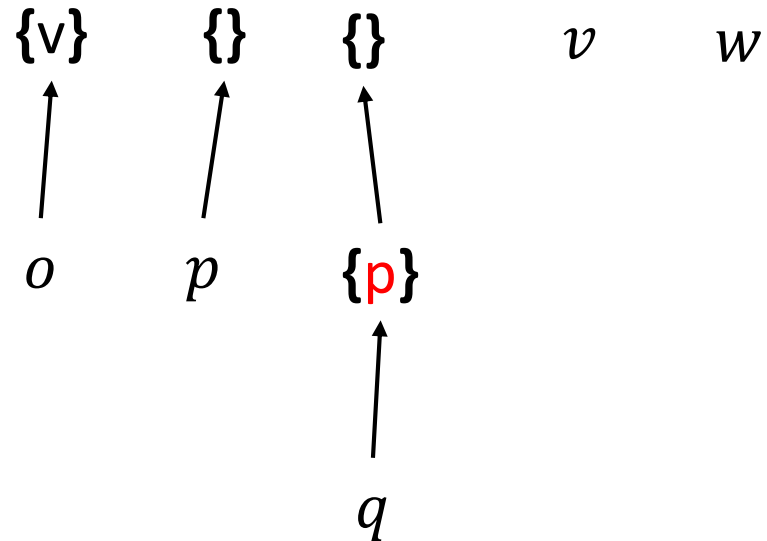


对于集合和元素的合并，添加元素到集合中，并添加元素的后继为集合的后继



合并操作执行方法

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

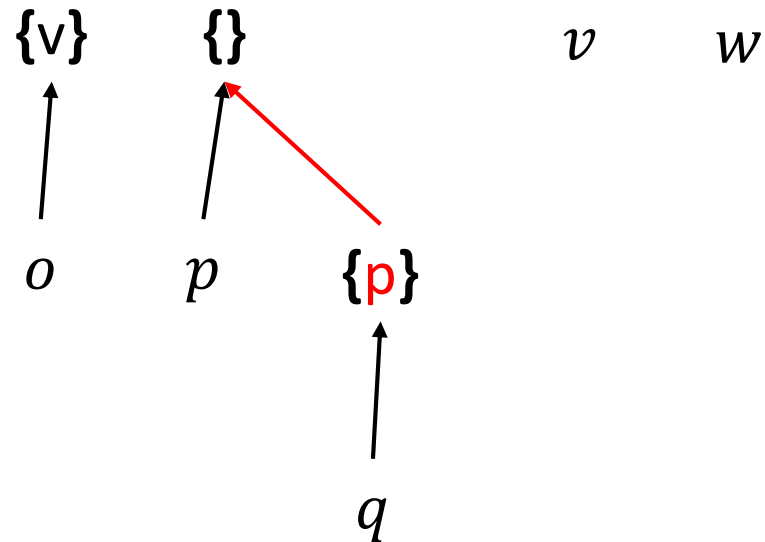


对于集合和元素的合并，添加元素到集合中，并添加元素的后继为集合的后继



合并操作执行方法

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

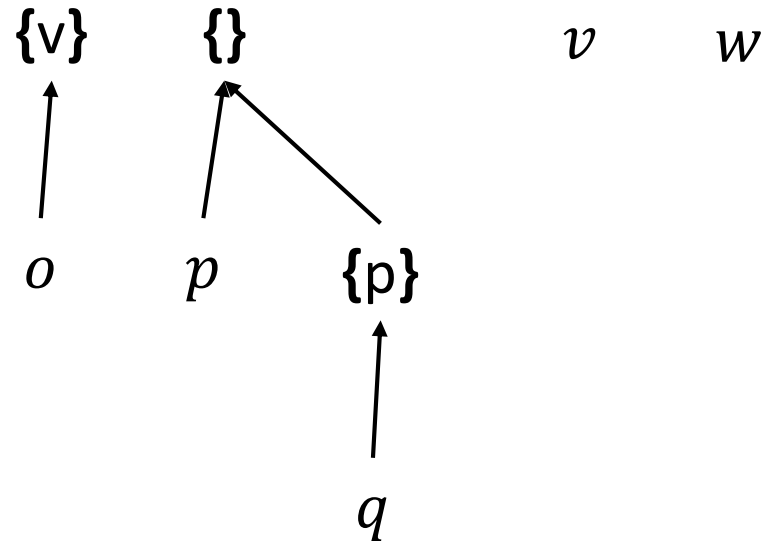


继续合并后继



合并操作执行方法

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

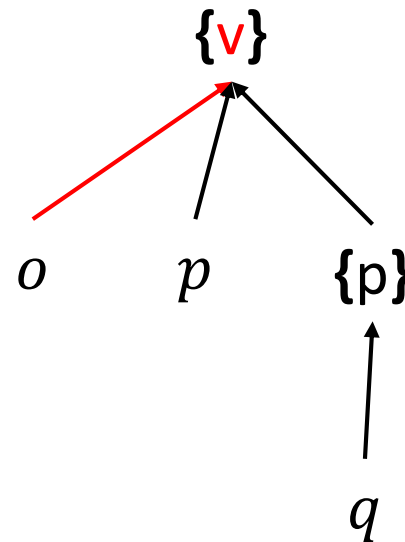


对于集合的合并，直接合并两个集合



合并操作执行方法

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



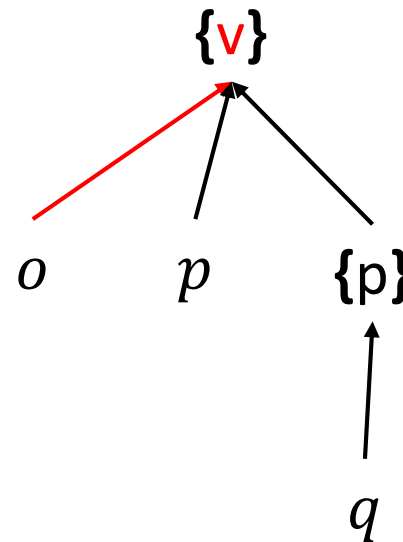
v w

对于集合的合并，直接合并两个集合



合并操作执行方法

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



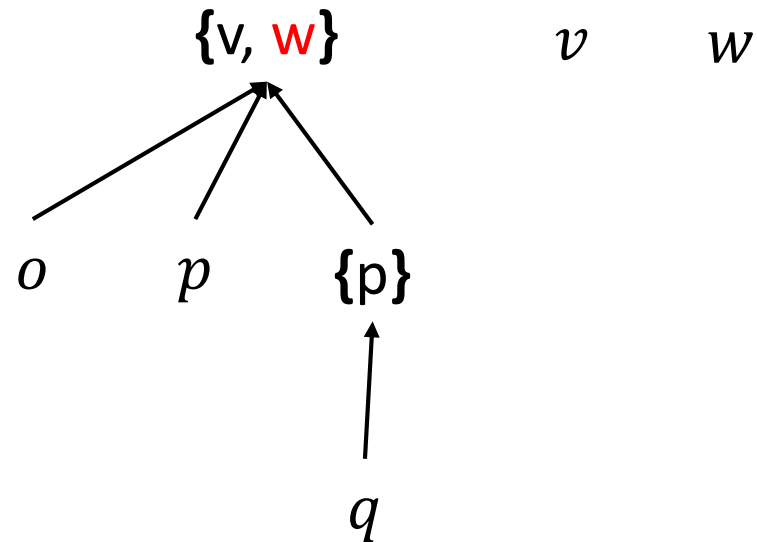
v w

对于集合的合并，直接合并两个集合



合并操作执行方法

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



- 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，但实践中常直接从指向分析导出



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

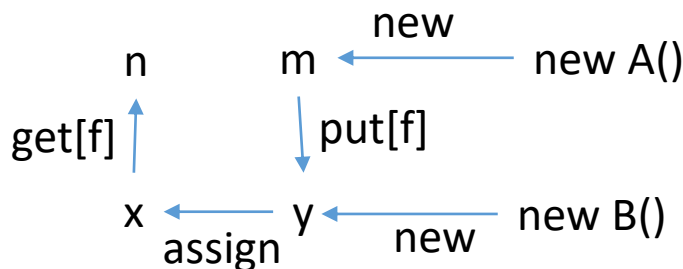
- 降低上下文敏感性：把被调方法根据上下文克隆n次
- 降低域敏感性：把域展开n次

复习：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 \bar{f}

```

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

以下文法安全吗？精确吗？

- $FlowTo = new FlowTo'$
 $FlowTo' = put[f] FlowTo' get[f]$
 $| FlowTo' FlowTo' | assign | \epsilon$
- $FlowTo = new (Alias | put[f] Alias get[f])*$
 $PointsTo = (Alias | get[f] Alias put[f])* new$
 $Alias = PointsTo FlowTo | assign | assign$

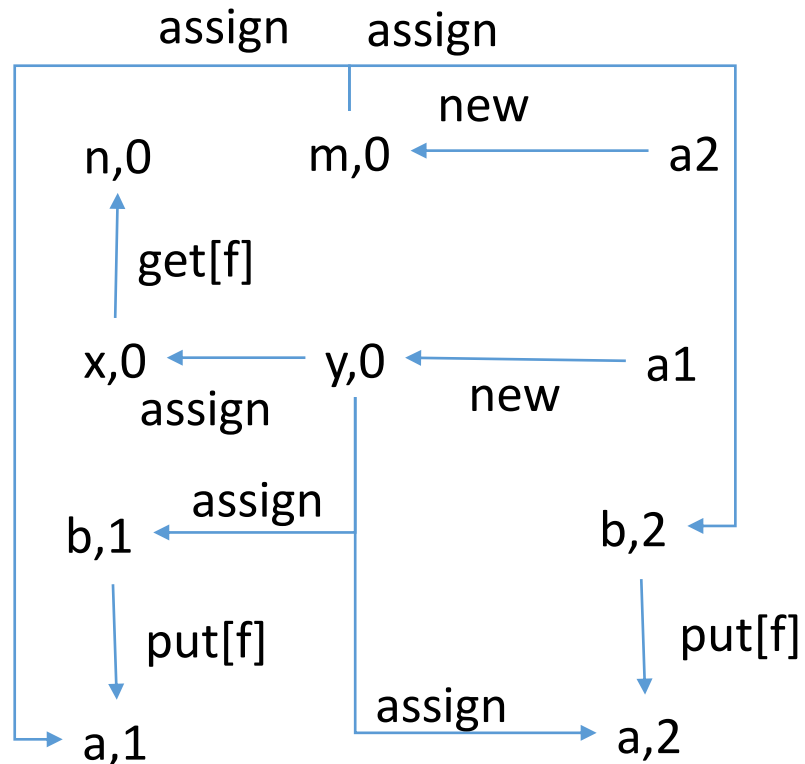


降低上下文敏感性

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;
```





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
 - $a \supseteq b$
 - $a \rightarrow next = b \rightarrow next$
- a->next=b产生
 - $a \rightarrow next \supseteq b$
 - $a \rightarrow next = b \rightarrow next$
- a=b->next产生
 - $a \supseteq b \rightarrow next$
 - $a \rightarrow next = b \rightarrow next$

能不能
换成 \supseteq

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 = b->next
 - a->next->next = b->next->next
- a=b->next产生
 - a \supseteq b->next
 - a->next = b->next->next
 - a->next->next = 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 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
 - $\forall y \in f\#y. \forall m \in \text{targets}(y, g),$
 $f\#x \supseteq m\#\text{ret}$
 $m\#\text{this} \supseteq \text{filter}(f\#y, \text{declared}(m))$
 $m\#a \supseteq f\#a$
 $m\#b \supseteq f\#b$
- 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** $\supseteq \{3\}$
- $\forall i \in \mathbf{main\#i}, \forall m \in \text{targets}(i, f),$
 - **main#i** \supseteq **m#ret**
 - **m#this** \supseteq $\text{filter}(\mathbf{main\#i}, \text{declared}(m))$
- $\forall i \in \mathbf{main\#i}, \forall m \in \text{targets}(i, f),$
 - **main#x** \supseteq **m#ret**
 - **mthis** \supseteq $\text{filter}(\mathbf{main\#i}, \text{declared}(m))$
- **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



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