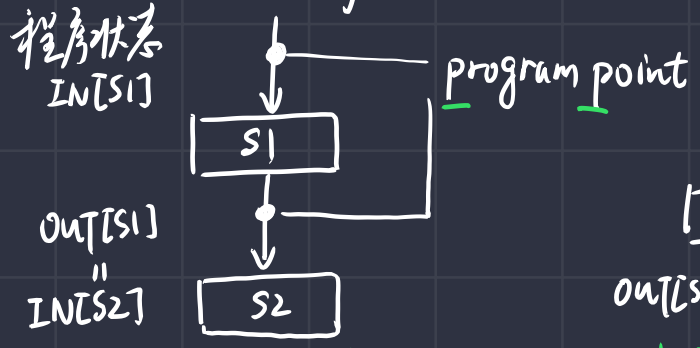
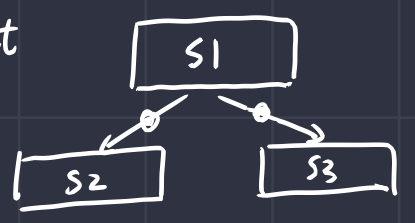
软件分析

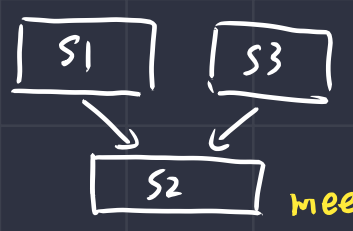
一、数据流分析应用

1. Preliminaries of Data Flow Analysis





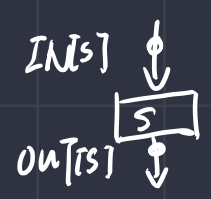
$OUT[S1] = IN[S2] = IN[S3]$



$IN[S2] = OUT[S1] \wedge OUT[S3]$

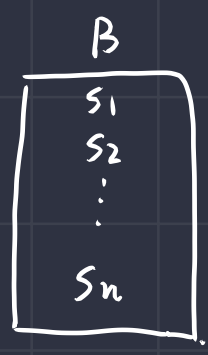
∀ PP 关联个数据流值，代表程序状态的抽象

- Forward Analysis



，记作：$OUT[S] = f_S(IN[S])$

- Backward Analysis，$IN[S] = f_S(OUT[S])$



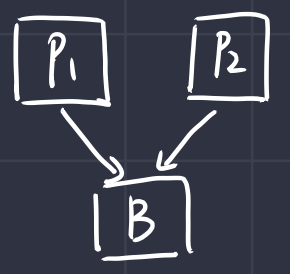
$IN[S_{i+1}] = OUT[S_i]$

$IN[B] = IN[S_1]$

$OUT[B] = OUT[S_n]$

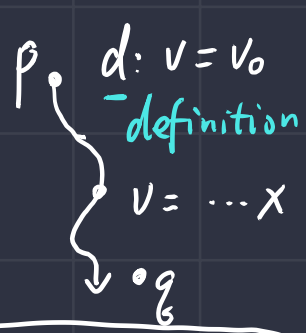
$= f_B(IN[B])$

$\Rightarrow f_B = f_{S_n} \circ \dots \circ f_{S_2} \circ f_{S_1}$



$IN[B] = \wedge_{P \text{ a predecessor of } B} f_B\, OUT[P]$

2. Reaching Definitions Analysis



Reach 条件：① $P \to q$ 存在路径

② d 不被 killed（v 不被重新定义）

bit vectors：$D_1\ D_2, \dots, D_n$ (definition)

$0\ 0, \dots, 1$

（0 代表不能 reach）

transfer function

B | D: v = x op y | → kill 所有定义 v 的地方 $\Rightarrow OUT[B] = gen_B \cup (IN[B] - kill_B)$

$IN[B] = \cup_{P \text{ a predecessor of } B} OUT[P]$

迭代算法：

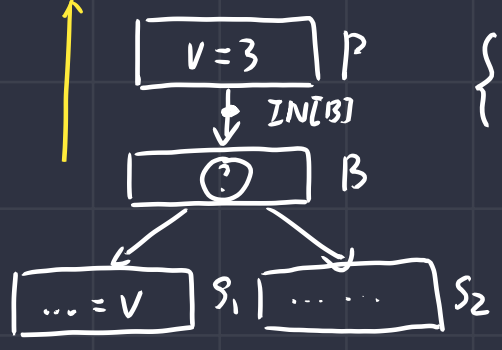
```
OUT[entry] = Φ;
for (each basic block B \ entry)
    OUT[B] = Φ;        初始化
while (changes to any OUT occur)
    for (each basic block \ entry) {
        IN[B] = ∪P a predecessor of B OUT[P];
        OUT[B] = genB ∪ (IN[B] - killB)
    }
```

应用：检测未被初始化

3. Live Variables Analysis 活跃变量分析

数据如何抽象 $V_1, V_2, V_3, \dots, V_{100}$ (0: dead, 1: live)

bit vectors 0 0 0 ..., 1

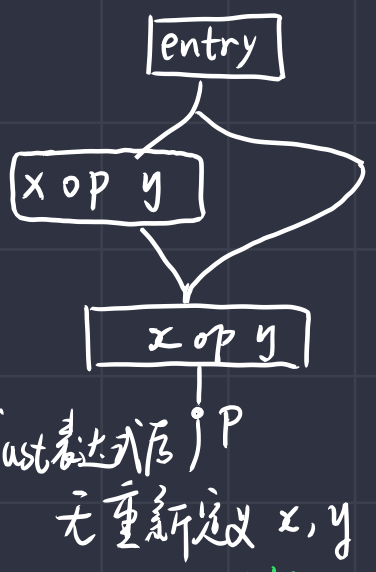


$$\begin{cases} OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S] \\ IN[B] = use_B \cup (OUT[B] - def_B) \end{cases}$$

INPUT: CFG (def_B and use_B computed for each basic block B)
OUTPUT: IN[B] and OUT[B] for each basic block B
METHOD:

```
IN[exit ]= ∅;
for (each basic block B\exit)
    IN[B] = ∅;
while (changes to any IN occur)
    for (each basic block B\exit) {
        OUT[B] = ∪ S a successor of B IN[S];
        IN[B] = use_B ∪ (OUT[B] - def_B);
    }
```

4. Available Expressions Analysis (Must analysis)



(ust表达式后) P 无重新定义 x, y

x op y 可被替换成值

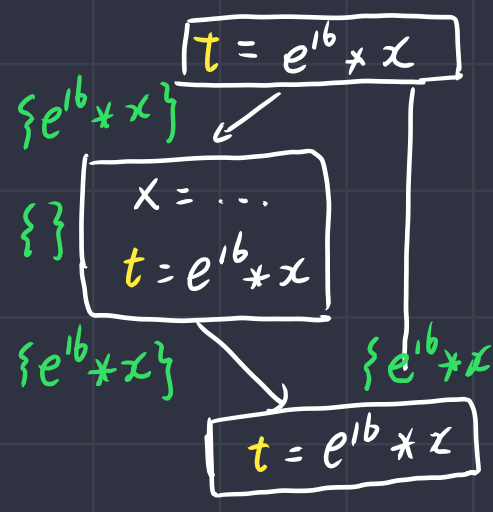
All expressions $E_1, E_2, \dots, E_{100}$ → bit vectors 1代表是Available

a = x op y

- Add to OUT the expression x op y (gen)
- Delete from IN any expression involving variable a (kill)

$$\Rightarrow \begin{cases} OUT[B] = gen_B \cup (IN[B] - kill_B) \\ IN[B] = \bigcap_{P \text{ a predecessor of } B} OUT[P] \end{cases}$$

初始化为 1111 (除 OUT[entry])



Lab1:

二. 数据流分析基础

1. CFG 之有 k个 statement (node), domain of the values in data flow analysis is V

定义k元组 $(OUT[n_1], OUT[n_2], \dots, OUT[n_k]) \rightarrow V_1 \times V_2 \times \dots \times V_k \Rightarrow V^k$

每次迭代 as a Function $F: V^k \rightarrow V^k$

init → $(\perp, \perp, \dots, \perp) = X_0$

iter1 → $(v_1', v_2', \dots, v_k') = X_1 = F(X_0)$

iter2 → $(v_1^2, v_2^2, \dots, v_k^2) = X_2 = F(X_1)$

⋮

iter i $(v_1^i, v_2^i, \dots, v_k^i) = X_i = F(X_{i-1})$

iter i+1 → $(v_1^i, v_2^i, \dots, v_k^i) = X_{i+1} = F(X_i)$

$\because X_i = X_{i+1}$

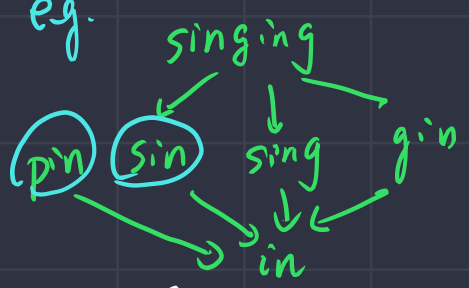
$\therefore X_i = X_{i+1} = F(X_i)$ 不动点

2. 偏序 Partial Order

poset (偏序集) as a pair $(P, \sqsubseteq)$ 集合 偏序关系

1) 自反性 $\forall x \in P, x \sqsubseteq x$ Reflexivity
2) 反对称性 $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$ Antisymmetry
3) 传递性 $\forall x, y, z \in P, x \sqsubseteq y, y \sqsubseteq z \Rightarrow x \sqsubseteq z$ Transitivity

eg.



"pin" 和 "sin" 可以无法比较

$h = 2$

3. 上界和下界. Given a poset $(P, \sqsubseteq)$, 子集 $S \subseteq P$,

$u \in P$ 是 upper bound of S, if $\forall x \in S, x \sqsubseteq u$. Similarly,

$l \in P$ 是 lower bound 的 S. if $\forall x \in S, l \sqsubseteq x$.

最小上界 least upper bound (lub 或 join) $\Rightarrow \sqcup S$ 对 $\forall$ 上界 u, $\sqcup S \sqsubseteq u$

最大下界 least greatest lower bound (glb 或 meet) $\Rightarrow \sqcap S$, $\cdots l \sqsubseteq \sqcap S$

当集合只有2个元素. $\sqcup S \Rightarrow a \sqcup b$, $\sqcap S \Rightarrow a \sqcap b$

性质: 偏序集不一定有 lub 和 glb, 若有则唯一

4. Lattice 格. 对 poset $(P, \sqsubseteq)$, $\forall a, b \in P$, if $a \sqcup b$ and $a \sqcap b$ 存在, 那么 $(P, \sqsubseteq)$ 为格

半格: 只存在 $a \sqcup b$ → join 半格 ; 只存在 $a \sqcap b$, meet 半格

△ 全格 $\forall S \subseteq P$. $\sqcup S$ and $\sqcap S$ exist → 表示集合元素性

(Complete Lattice) 最大元素 $\top = \sqcup P$ called top
最小元素 $\bot = \sqcap P$ called bottom

∀ 有界的格 → 全格

Product Lattice, $L_1 = (P_1, \sqsubseteq_1)$, $L_2 = (P_2, \sqsubseteq_2) \cdots$, 对 $\forall i$, $(P_i, \sqsubseteq_i)$ has $\sqcup_i$ and $\sqcap_i$

例 $L^n = (P, \sqsubseteq)$: $P = P_1 \times P_2 \times \cdots \times P_n$, $(x_1, x_2, \cdots, x_n) \sqsubseteq (y_1, y_2, \cdots, y_n) \Leftrightarrow (x_1 \sqsubseteq y_1) \wedge (x_2 \sqsubseteq y_2) \cdots$

5. Data Flow Analysis Framework via Lattice

(D, L, F)

方向 Lattice transfer functions from V to V

$\downarrow$

$V, \sqcap, \sqcup$

6. Lattice 之间的函数 $f: L \to L$

单调性 f is monotonic if $\forall x, y \in L$, $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$

Fixed-Point Theorem 不动点定理

Given a complete lattice $(L, \sqsubseteq)$ if

(1) $f: L \to L$ is monotonic (2) L is finite (有限)

则能找到 the least fixed point (最小不动点) 通过迭代

$f(\bot), f(f(\bot)), \ldots, f^k(\bot)$ until a fixed point is reached

greastest fixed point (最大不动点)

$f(\top), f(f(\top)), \ldots, f^k(\top)$ until a fixed point is reached

证明: By the definition of $\bot$ and $f: L \to L$

$$\bot \sqsubseteq f(\bot)$$

As f is monotonic, $f(\bot) \sqsubseteq f(f(\bot)) = f^2(\bot)$

Similarly, $\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots \sqsubseteq f^i(\bot)$

As L is finite, for some k $f^{Fix} = f^k(\bot) = f^{k+1}(\bot)$, 不动点存在

Assume we have another fixed point x, i.e., $x = f(x)$, we have $\bot \sqsubseteq x$

数学归纳法 $f(\bot) \sqsubseteq f(x)$

Assume $f^i(\bot) \sqsubseteq f^i(x)$, as f is monotomic $f^{i+1}(\bot) \sqsubseteq f^{i+1}(x)$

Thus by induction, $f^i(\bot) \sqsubseteq f^i(x)$

Thus $f^i(\bot) \sqsubseteq f^i(x) = x$, $f^{Fix} = f^k(\bot) \sqsubseteq x$. 得证. (唯一不动点)

7. 应用到 DFA. $\begin{cases} f: L \to L \text{ monotonic} \\ \sqcup/\sqcap \quad L \times L \to L \text{ monotonic} \end{cases}$ Proof: $\forall x, y, z \in L$. $x \sqsubseteq y$ 求证 $x \cup z \sqsubseteq y \cup z$

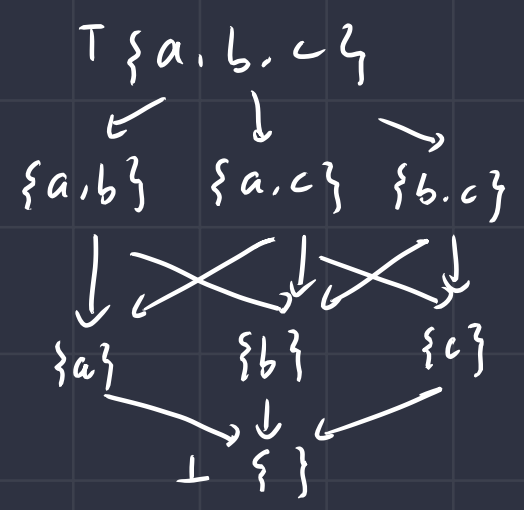
$y \sqsubseteq y \cup z$ 由 $x \sqsubseteq y$ 得 $x \sqsubseteq y \cup z$ 又 $z \sqsubseteq y \cup z$

$x \cup z$ 是最小上界 $\Rightarrow x \cup z \sqsubseteq y \cup z$

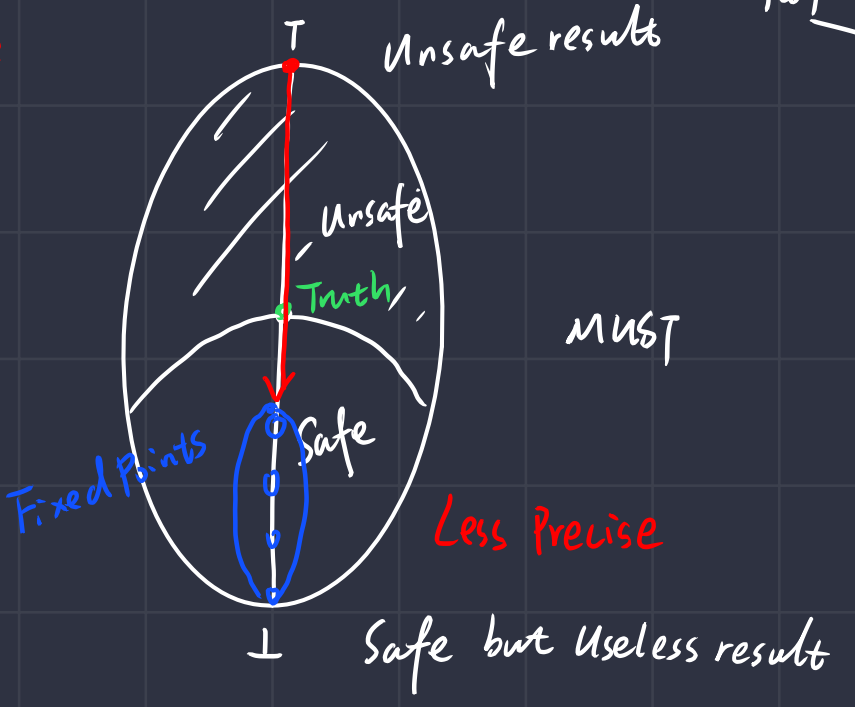
When will the Algorithm Reach the Fixed Point?

k-tuple, 最坏迭代 $h \cdot k$ 次

$h=3$



8. May and Must Analysis, a Lattice View



9 Meet-Over-All-Paths Solution (MOP)

$MOP[S_i] = \sqcup/\sqcap \quad F_p(OUT[entry])$

A Path P from Entry to S_i

Ours = $F(x \cup y)$

MOP = $F(x) \cup F(y)$

$\Rightarrow MOP \subseteq Ours$

Bit-vector or Gen/kill distributive

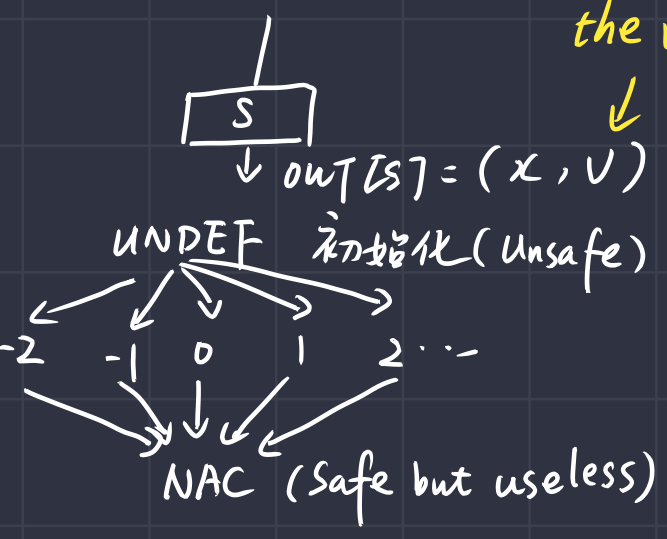
若 F is distributive 即 $F(x \cup y) = F(x) \cup F(y)$, MOP = Ours

10. Constant Propagation 常量传播

P → { x ，x是否保证持有常数

D, L, F

Forward



· 值域: ... -2 -1 0 1 2 ... NAC (Safe but useless)

· Meet Operator ⊓

- NAC ⊓ v = NAC
- UNDEF ⊓ v = v — Uninitialized variables are not focus in our constant program analysis.
- c ⊓ c = c
- $c_1 \sqcap c_2$ = NAC

· Transfer Function S: x = ..., F: OUT[S] = gen ∪ (IN[S] − {(x, _)})

通配符

- S: x = c; gen = {(x, c)}
- S: x = y; gen = {(x, val(y))}
- S: x = y op z; gen = {(x, f(y, z))}

$f(y,z) = \begin{cases} val(y)\ op\ val(z) & // val(y), val(z) 都是常量 \\ NAC & // val(y) 或 val(z) 是 NAC \\ UNDEF & // Otherwise \end{cases}$

不是distributed

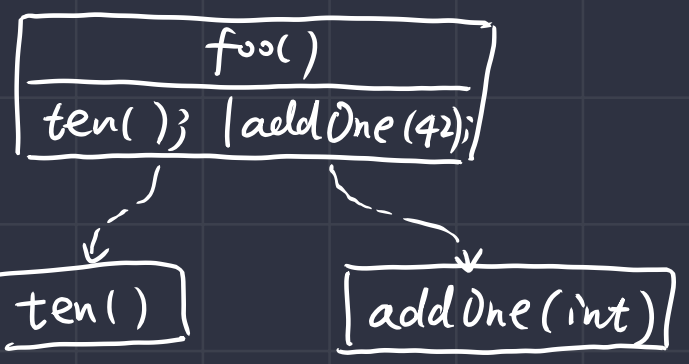
11 Worklist Algorithm (an optimization of Iterative Algorithm)

```
初始化
Worklist ← all basic block
while (Worklist is not empty)
    pick a basic block B from Worklist
    old_OUT = OUT[B]
    IN[B] = U_{P a predecessor of B} OUT[P];
    OUT[B] = gen_B ∪ (IN[B] − kill_B);
    if (old_OUT ≠ OUT[B])
        Add all successors of B to Worklist
```

SPFA类似思想

三 过程间分析

Call Graph Construction (CHA)



1. Java 函数调用

	Staticcall	Special Call	Virtual Call
Instruction	invokestatic	invokespecial	invokeinterface invokevirtual
Target methods	静态方法	构造方法 私有实例方法 父类实例方法	其他实例方法
#Target methods	1	1	≥1
Determinacy	Compile-time	Compile-time	Runtime

2. Method Dispatch

o: foo(...);

方法签名: <C: T foo(P, Q, R)> = class type + method name + descriptor

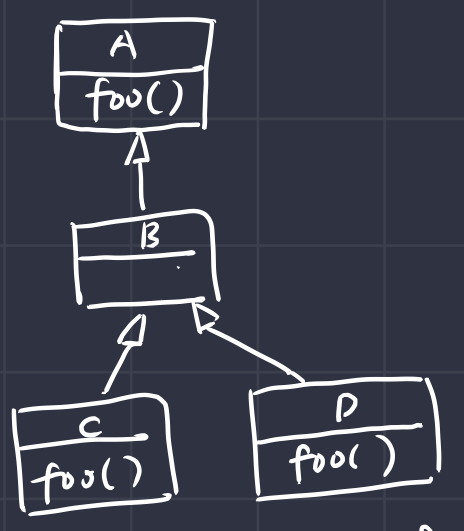
descriptor: return type + parameter types

- type of receiver object (o): C
- method signature at the call site: m

定义 Dispatch(c, m) = { m'. m'与m有相同name和descripter
Dispatch(c', m), c'是c的父类

3. Class Hierarchy Analysis (CHA)

```
Resolve(cs)                                   → 找目标方法
  T = {}
  m = method signature at cs
  if cs is a static call then
    T = { m }
  if cs is special call then
    c^m = class type of m
    T = { Dispatch(c^m, m) }
  if cs is a virtual call then
    c = declared type of receiver variable at cs
    foreach c' that is a subclass of c or c itself do
      add Dispatch(c', m) to T
  return T
```

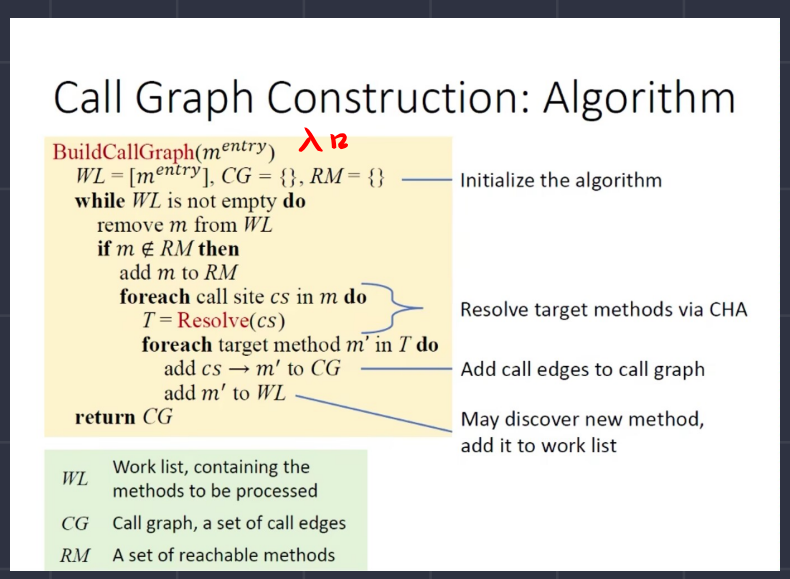


Resolve(b.foo()) = { A.foo(), C.foo(), D.foo() }

只关注声明类型

Common usage: IDE

4. 用CHA构造整个程序的调用图



类BFS

5. Interprocedural Control-Flow Graph (ICFG)

ICFG = 各方法的CFGs + Call edges + Return edges

```
void foo(){
    bar(...); // call site
    int n=3; // return site
}
```

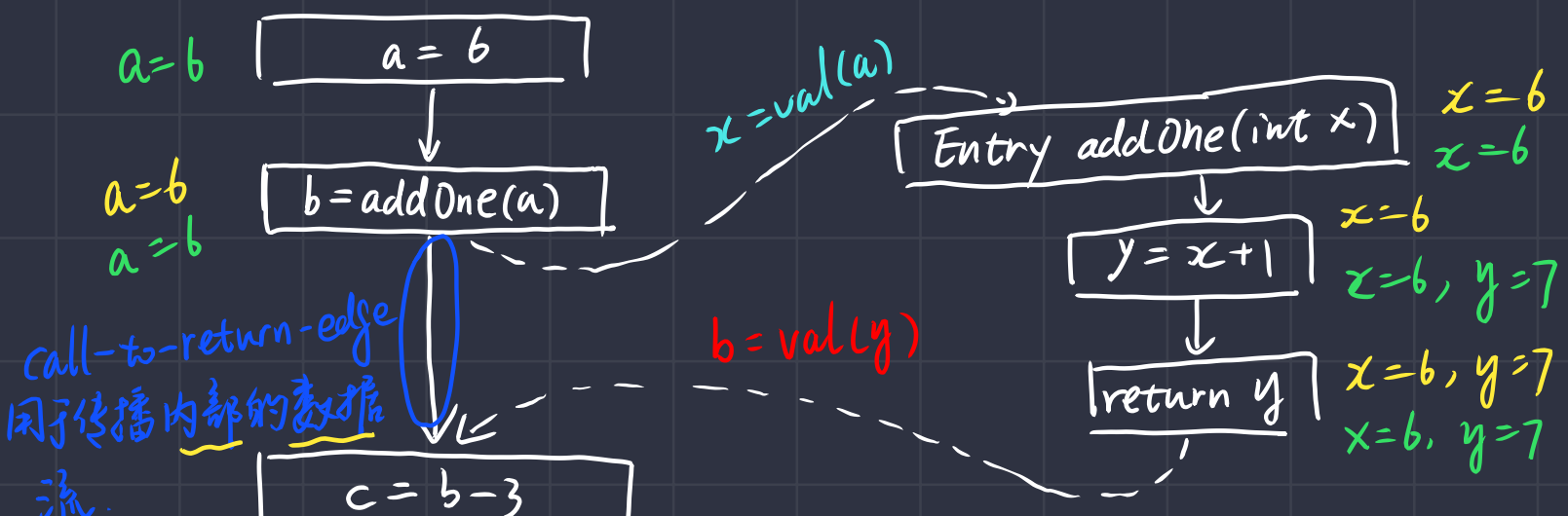
返回点，紧跟call site

6. Interprocedural Data-Flow Analysis

与过程内相比，还需要 edge transfer { call edge transfer / Return edge transfer

过程间常量传播 { call edge transfer 传参数值 / Return edge transfer 传返回值

① b = addOne(a); kill掉 b (LHS)



四. 指针分析，回答指针能指向哪些内存地址/对象

别名 aliases：

```
p = new C();
q = p;
```

p和q指向同一个对象

四个要素

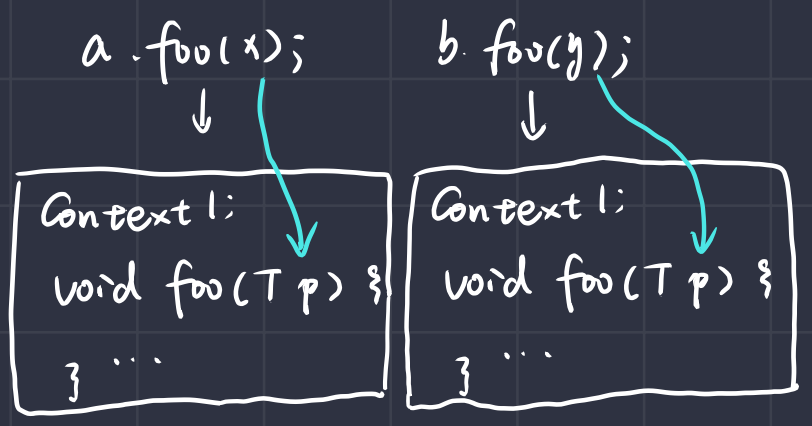
1. Heap Abstraction 堆抽象 限制静态分析对象数目，保证可终止

分类：Allocation-Site Abstraction ✓ 一个抽象对象代表创建点创建的所有对象

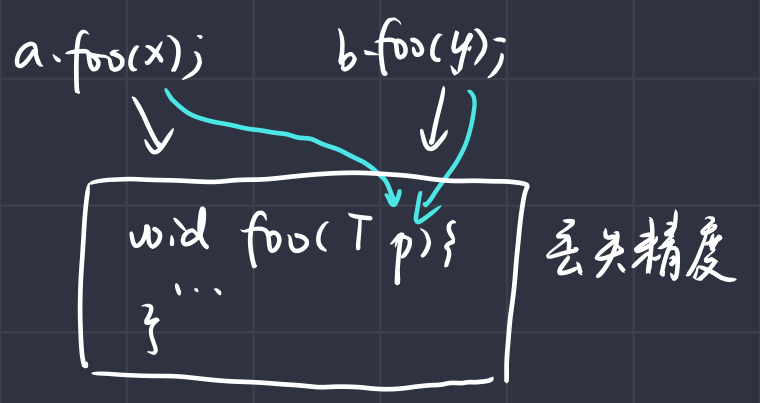
```
1 for (i=0; i<3; ++i) {
2     a = new A(); O2(创建点)
3     ...
4 }
```

$O_2, i=0$
$O_2, i=1$ ⟶ O_2
$O_2, i=2$

2. Context Sensitivity 上下文敏感 ✓



上下文不敏感 ✓



丢失精度

3. Flow Sensitivity 流敏感

```
1 | c = new C();
2 | c.f = "x";
3 | s = c.f;
4 | c.f = "y";
```

c → {O_1}
c → {O_1}
O_1.f → {"x"}
...
s → {"x"}

c → {O_1}
O_1.f → {"y"}
s → {"x"}

非流敏感：忽视语句顺序

c → {O_1}
O_1.f → {"x","y"}
s → {"x","y"}

✓ 主流做法（OO程序分析）

false positive

4. Analysis Scope 回答应该分析程序哪一部分

Whole-program ✓　　Demand-driven（需求驱动）

```
1  x = new A();
2  y = x;
3  y.foo();
4  z = new T();
5  z.bar();
```

x → {O_1}
y → {O_1}
z → {O_4}

若只对第5行感兴趣

z → {O_4}

指针分析关注的语句

Pointers in Java

- Local Variable：x
- Static field：C.f　global variable
- Instance field：x.f　建模成object with a field f
- Array element：array[i]　忽略下标，array[i] => array.arr　建模成单个field

Pointer-Affecting Statements：

New	x = new T()
Assign	x = y
Store	x.f = y
Load	y = x.f
Call	r = x.k(a,...)

x.f.g.h = y;

⇓

t1 = x.f;
t2 = t1.g;
t2.h = y;

三地址码

五、指针分析基础

1. Domains and Notations

Variable：x, y ∈ V
Fields：f, g ∈ F
Objects：O_i, O_j ∈ O
Instance fields：O_i.f, O_j.g ∈ O × F
Pointers：Pointer = V ∪ (O × F)

Points-to relations：

pt = Pointer → $\mathcal{P}$(O)　　powerset of O　O的幂集

pt(p)：points-to set of p

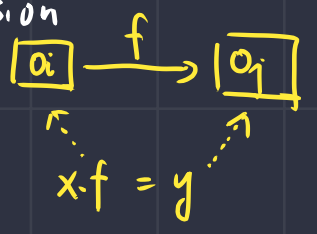
2. Rules 规则

Kind	Statement	Rule
New	i: $x = \text{new } T()$ 开始	$\frac{}{o_i \in pt(x)}$
Assign	$x = y$	$\frac{o_i \in pt(y)}{o_i \in pt(x)}$
Store	$x.f = y$	$\frac{o_i \in pt(x),\ o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	$y = x.f$	$\frac{o_i \in pt(x),\ o_j \in pt(o_i.f)}{o_j \in pt(y)}$

把o_i加入x的指针集

前提条件 Premises

Conclusion



形成一系列约束关系　关键：when pt(x) changed，要传播改变部分给x的相关指针 ⇒图

3. Pointer Flow Graph (PFG) 指针流图 (有向图)

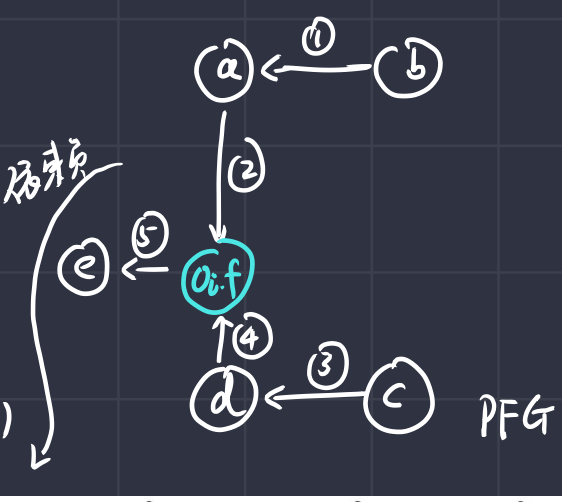
Node: Pointer = $V \cup (O \times F)$　a variable or a field of an abstract object

Edges: Pointer × Pointer　$x \to y$　x指向object may flows to y指向object

建边规则按红色箭头指示

e.g.

```
a = b;    ①
c.f = a;  ②
d = c;    ③
c.f = d;  ④
e = d.f;  ⑤
```



⇒传递闭包 transitive closure

PFG和关系传播相互依赖、动态增加

4. 算法

Pointer Analysis: Algorithms

```
Solve(S)
  WL = [ ], PFG = { }
  foreach i: x = new T() ∈ S do
    add ⟨x, {o_i}⟩ to WL
  foreach x = y ∈ S do
    AddEdge(y, x)
  while WL is not empty do
    remove ⟨n, pts⟩ from WL
    Δ = pts − pt(n)
    Propagate(n, Δ)
    if n represents a variable x then
      foreach o_i ∈ Δ do
        foreach x.f = y ∈ S do
          AddEdge(y, o_i.f)
        foreach y = x.f ∈ S do
          AddEdge(o_i.f, y)
```

main algorithm

初始化

避免冗余处理

```
AddEdge(s, t)
  if s → t ∉ PFG then
    add s → t to PFG
    if pt(s) is not empty then
      add ⟨t, pt(s)⟩ to WL

Propagate(n, pts)
  if pts is not empty then
    pt(n) ∪= pts
    foreach n → s ∈ PFG do
      add ⟨s, pts⟩ to WL
```

S	Set of statements of the input program
WL	Work list
PFG	Pointer flow graph

Tian Tan @ Nanjing University　44

要分析语句的集合

WL ⊆ ⟨Pointer, pt(O)⟩*

⟨n, pts⟩：pts需要加入pt(n)

5. 指针分析处理方法调用

规则

Kind	Statement
Call	$l: r = x.k(a_1, \ldots, a_n)$

Rule

$o_i \in pt(x),\ m = dispatch(o_i, k)$
$o_u \in pt(a_j),\ 1 \le j \le n$
$o_v \in pt(m_{ret})$

$o_i \in pt(m_{this})$
$o_u \in pt(m_{pj}),\ 1 \le j \le n$
$o_v \in pt(r)$

传递回值
PFG: $r \leftarrow m_{ret}$

实参传形参, PFG建边 $a_j \rightarrow m_{pj}$

算法:

Algorithms

New part

Solve(m^{entry})
  $WL=[\,], PFG=\{\}, S=\{\}, RM=\{\}, CG=\{\}$
  AddReachable(m^{entry})
  while WL is not empty **do**
    remove $\langle n, pts\rangle$ from WL
    $\Delta = pts - pt(n)$
    Propagate(n, Δ)
    if n represents a variable x **then**
      foreach $o_i \in \Delta$ **do**
        foreach `x.f = y` $\in S$ **do**
          AddEdge($y, o_i.f$)
        foreach `y = x.f` $\in S$ **do**
          AddEdge($o_i.f, y$)
        ProcessCall(x, o_i)

S	Set of reachable statements
S_m	Set of statements in method m
RM	Set of reachable methods
CG	Call graph edges

AddReachable(m)
  if $m \notin RM$ **then**
    add m to RM
    $S \cup= S_m$
    foreach `i: x = new T()` $\in S_m$ **do**
      add $\langle x, \{o_i\}\rangle$ to WL
    foreach `x = y` $\in S_m$ **do**
      AddEdge(y, x)

ProcessCall(x, o_i)
  foreach `l: r = x.k(a1,…,an)` $\in S$ **do**
    m = Dispatch(o_i, k)
    add $\langle m_{this}, \{o_i\}\rangle$ to WL
    if $l \rightarrow m \notin CG$ **then**
      add $l \rightarrow m$ to CG
      AddReachable(m)
      foreach parameter p_i of m **do**
        AddEdge(a_i, p_i)
      AddEdge(m_{ret}, r)

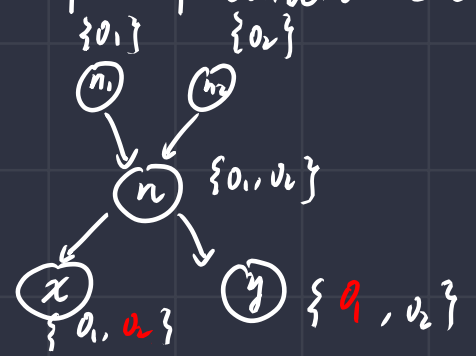
o_i是流到x的新对象

解目标方法

115

六. 上下文敏感指针分析 Context-Sensitive Pointer Analysis

C.I. (上下文不敏感)

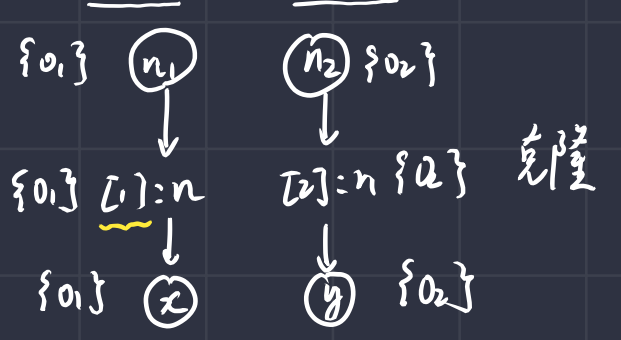


⇒ 不准, 速度快

```
x = id(n1);
y = id(n2);
int i = x.get();
Number id(Number n){
    return n;
}
```

C.S. call-site sensitivity id → [1] 和 [2]

1. Cloning-Based Context Sensitivity 方法、变量加上下文



克隆

2. Context-Sensitive Heap 上下文加在堆区抽象对象

3. Domains and Notations

Context $c, c', c'' \in C$ 上下文
Context-sensitive method: $c:m \in C \times M$
Context-sensitive variables: $c:x, c':y \in C \times V$
Context-sensitive objects: $c:o_i, c':o_j \in C \times O$
Fields: $f, g \in F$
Instance fields: $c:o_i.f, c':o_j.g \in C \times O \times F$
Context-Sensitive pointers: $CSPointer = (C \times V) \cup (C \times O \times F)$
pt: $CSpointer \rightarrow \mathcal{P}(C \times O)$

4. 规则 Rules

Kind	Statement	Rule (under context c)
New	$i: x = \text{new } T()$ In $c:m$	$\dfrac{}{c:o_i \in pt(c:x)}$
Assign	$x = y$ In $c:m$	$\dfrac{c':o_i \in pt(c:y)}{c':o_i \in pt(c:x)}$
Store	$x.f = y$ In $c:m$	$\dfrac{c':o_i \in pt(c:x),\ c'':o_j \in pt(c:y)}{c'':o_j \in pt(c':o_i.f)}$
Load	$y = x.f$ In $c:m$	$\dfrac{c':o_i \in pt(c:x),\ o_j \in pt(c':o_i.f)}{c'':o_j \in pt(c:y)}$
Call	$l: r = x.k(a_1,\dots,a_n)$	$\dfrac{\begin{array}{c}c':o_i \in pt(c:x) \\ m = Dispatch(o_i,k)\quad c^t = Select(c,l,c':o_i,m) \\ c'':o_n \in pt(c:a_j),\ 1 \le j \le n \\ c''':o_v \in pt(c^t:m_{ret})\end{array}}{\begin{array}{c}c':o_i \in pt(c^t:m_{this}) \\ c'':o_n \in pt(c^t:m_{p_j}),\ 1 \le j \le n \\ c''':o_v \in pt(c:r)\end{array}}$

PFG edges: $c:x \leftarrow c:y$

目标方法上下文

5. PFG with C.S.

Nodes: CSPointer = $(C \times V) \cup (C \times O \times F)$

6. 算法: Edges: CSPointer × CSPointer

C.S. Pointer Analysis: Algorithm

```
Solve(m^entry)
  WL=[ ], PFG={ }, S={ }, RM={ }, CG={ }
  AddReachable([ ]: m^entry)
  while WL is not empty do
    remove ⟨n, pts⟩ from WL
    Δ = pts − pt(n)
    Propagate(n, Δ)                    与C.2.分析一致
    if n represents a variable c: x then
      foreach c': o_i ∈ Δ do
        foreach x.f = y ∈ S do
          AddEdge(c: y, c': o_i.f)
        foreach y = x.f ∈ S do
          AddEdge(c': o_i.f, c: y)
        ProcessCall(c: x, c': o_i)
```

```
AddReachable(c: m)
  if c: m ∉ RM then
    add c: m to RM
    S ∪= S_m
    foreach i: x = new T() ∈ S_m do
      add ⟨c: x, {c: o_i}⟩ to WL
    foreach x = y ∈ S_m do
      AddEdge(c: y, c: x)              与C.2.分析一致

ProcessCall(c: x, c': o_i)
  foreach l: r = x.k(a1,…,an) ∈ S do
    m = Dispatch(o_i, k)
    c^t = Select(c, l, c': o_i)
    add ⟨c^t: m_this, {c': o_i}⟩ to WL
    if c: l → c^t: m ∉ CG then
      add c: l → c^t: m to CG
      AddReachable(c^t: m)
      foreach parameter p_i of m do
        AddEdge(c: a_i, c^t: p_i)
      AddEdge(c^t: m_ret, c: r)
```

S	Set of reachable statements
S_m	Set of statements in method m
RM	Set of **C.S.** reachable methods
CG	**C.S.** call graph edges

$c:2 \to c^t: T.foo(A,A) \in CG$

7. Context Sensitivity Variants

Call-site sensitivity
Object sensitivity
Type sensitivity

Select(c, l, c': o_i, m)

- c
- l
- c: o_i
- T: func(A, A)

1) Call-site sensitivity Select(c, l, c': o_i, m) = [l', ..., l'', l]

where c = [l', ..., l'']

Also called call-string sensitivity or k-CFA

递归？限制上下文长度 以调用链末尾k个调用点 (usually $k \le 3$) 常用 方法 k=2 堆 k=1

2) Object Sensitivity Select(c, l, c': o_i, m) = [o_j, ..., o_k, o_i]

where c' = [o_j, ..., o_k]

实际表现(OO语言) Object Sensitivity 表现更好

3) Type Sensitivity Select(c, l, c': o_i, m) = [t', ... t'', InType(o_i)]

where c' = [t', ..., t'']

```
class X {
    void main() {
        Y y1 = new Y();
    }
}
```

→ 上下文: [X]

精确度，换取速度

七. 静态分析 - 安全 Security

1. 信息流安全 Information Flow Security

互补 { Access Control 访问控制 点、
Information Flow Security 线 关心信息如何传播

```
a = x;
b.f = a;
y = b.f;
```

1) 信息流 variable x 信息传递到 y ⇒ 信息流 x → y

2) Security Levels (Classes) 分级 { H high security; L low security

modeled as lattice $L \le H$ 可以是复杂的格

3) Information Flow Policy

Noninterference Policy : Secret(H) ⇢ Public(L)

$X_L = y_H$ ✗ $x_L = y_L + z_H$ ✗

2. Confidentiality 和 Integrity

secret (H) $\xrightarrow{\times}$ Public (L)　　保密保护

trusted (H) $\xleftarrow{\times}$ untrusted (L)　　Shell 注入、SQL 注入　　完整保护

更广泛的意义 Integrity：正确性、完整性、一致性

3. Explict Flows and Covert Channels

$x_H = y_H$

$x_L = y_H$

$x_L = y_H + z_H$

隐式流 implict flow：

```
secret_H = getSecret();
if (secret_H < 0)        // 有信息流，无数据流
    public_L = 1;
else public_L = 0;
```

Does Secret information Leak?

```
while (secret_H < 0) { … };
```
Leak, we can conclude that secret is negative if the program does not terminate

Termination channels

```
if (secret_H < 0)
  for (int i = 0; i < 1000000; ++i) { … };
```
Leak, we can conclude that secret is negative if the program execution spends more time

Timing channels

```
if (secret_H < 0)
  throw new Exception("…");
```
Leak, we can conclude that secret is negative if we observe the exception

Exceptions

```
int sa_H[] = getSecretArray();
sa_H[secret_H] = 0;
```
Leak, exception may reveal that secret is negative

Covert/Hidden Channels

4. Taint Analysis 污点分析　　同位素标记法

把程序数据分为两类 { Data of interest — tainted data；Other data — untainted data }

两类问题 ⟹ { Confidentiality；Integrity }

Source $\xrightarrow{?}$ sink　　Can taint flow to a sink?

⇒ Which tainted data a pointer (at a sink) can point to?

拓展指针分析域

Variable：$x, y \in V$

Fields：$f, g \in F$

Objects：$o_i, o_j \in O$

Instance fields：$o_i.f, o_j.g \in O \times F$

Pointers：$Pointer = V \cup (O \times F)$

Tainted data：$t_i, t_j \in T \subset O$　　i 表示 callsite i

Inputs：Sources：a set of source methods

Sinks：a set of sink methods

Outputs：TaintFlows

元素 $\langle t_i, m \rangle$ 表示 tainted data from call site i may flow to sink method m

Rules：Call

① 处理 Source　　Statement　　Rules

Call　$l: r = x.k(a_1, \dots, a_n)$

$$\frac{l \to m \in CG \quad m \in Sources}{t_i \in pt(r)}$$

(传播规则和指针分析完全一致)

② 处理 Sink

Call　$l: r = x.k(a_1, \dots, a_n)$

$$\frac{l \to m \in CG \quad m \in Sinks \quad \exists i, 1 \le i \le n : t_j \in pt(a_i)}{\langle t_j, m \rangle \in TaintFlows}$$

八. 基于 Datalog 的程序分析

命令式语言 (C++ Java)　VS　声明式语言 (SQL)

1. Datalog : Prolog 子集、声明式逻辑编程语言

= Data + Log　　无副作用、无控制流、无函数、非图灵完备

1) • Predicate → table 关系
   • fact : Age : ("Xiaoming", 18) 存在于表中
   • Atoms E.g. Age("Xiaoming", 18), age >= 18, Age(person, age)
   • Rules : H <- $B_1, B_2, \dots, B_n$　(atom atom 逻辑与)

例程:
```
Adult(person) <-
    Age(person, age),
    age >= 18.
```

person	age
Xiaoming	18
Alan	16

2) Predicate 谓词 { EDB 原始的 / IDB 推导产生的

H ← B_1, B_2, B_3
IDB　EDB 或 IDB

3) 表达 逻辑或　1. 写两条 Rules　2.
```
SportFan(person) <-
    Hobby(person, "jogging");
    Hobby(person, "swimming").
```

表达 逻辑非　!B2(X2, X4)

e.g. 找补考学生:
```
MakeUpExamStd(student) <-
    Student(student),
    !PassedStd(student).
```

4) 支持递归: e.g. 可达性
```
Reach(from, to) <-
    Edge(from, to).
Reach(from, to) <-
    Reach(from, node),
    Edge(node, to).
```

rule is safe, $\forall x$. x至少出现在一个 non-negated relational atom 防止无穷产生

不能出现 $A(x) \leftarrow B(y), x > y$　　$A(x) \leftarrow B(y), !C(x,y)$ ×

Recursion and Negation　$A(x) \leftarrow B(x), !A(x)$ ×

2. 通用Datalog的指针分析

Datalog Model for Pointer Analysis

Kind	Statement
New	i: x = new T()
Assign	x = y
Store	x.f = y
Load	y = x.f

EDB

```
New(x : V, o : O)
Assign(x : V, y : V)
Store(x : V, f : F, y : V)
Load(y : V, x : V, f : F)
```

IDB

```
VarPointsTo(v: V, o : O)
```

e.g., fact VarPointsTo(x,o_i) represents $o_i \in pt(x)$

```
FieldPointsTo(oi : O, f: V, oj : O)
```

e.g., fact FieldsPointsTo(o_i,f,o_j) represents $o_j \in pt(o_i.f)$

Variables: V
Fields: F
Objects: O

规则: $\frac{}{o_i \in pt(x)}$　$x = new\ T()$

$\frac{o_i \in pt(y)}{o_i \in pt(x)}$　$x = y$

VarPointsTo(x, o) ←
New(x, o).

VarPointsTo(x, o) ←
Assign(x, y)
VarPointsTo(y, o).

Datalog处理方法调用　Label

EDB: VCall(l: S, x: V, k: M)　　ThisVar(m: M, this: V)

Dispatch(o: O, k: M, m: M)

IDB: Reachable(m: M)　　CallGraph(l: S, m: M)　Method

Rule

$o_i \in pt(x), m = dispatch(o_i, k)$
$o_u \in pt(a_j), 1 \le j \le n$
$o_v \in pt(m_{ret})$

▲ $o_i \in pt(m_{this})$
$o_u \in pt(m_{pj}), 1 \le j \le n$
$o_v \in pt(r)$

传递回值　PFG: $r \leftarrow m_{ret}$

实参传形参, PFG建边　$a_j \rightarrow m_{pj}$

多个参数

EDB: Argument(l: S, i: N, a: V)

Parameter(m: M, i: N, p: V)

① VarPointsTo(this, o),
Reachable(m),
CallGraph(l, m) ←
VCall(l, x, k),
VarPointsTo(x, o),
Dispatch(o, k, m),
ThisVar(m, this).

② VarPointsTo(p_i, o) ←
CallGraph(l, m),
Argument(l, i, a_i),
Parameter(m, i, p_i),
VarPointsTo(a_i, o)

EDB: MethodReturn(m:M, ret:V)

CallReturn(l:S, r:V) 接收返回值变量

③ VarPointsTo(r, o)<-
CallGraph(l, m),
Method(m, ret),
VarPointTo(ret, o),
CallReturn(l, r).

3. Datalog 污点分析 → PPT

九. CFL-Reachability and IFDS ↗ 程序分析框架

Infeasible Paths: Paths in CFG that do not correspond to actual executions. 动态运行时不会执行到

Realizable Paths: The paths in which "returns" are matched with corresponding "calls"

补充(回忆) 上下文无关文法 · $S \to aSb$ 左边可被右边替换无论S在哪里 ("天吃肉")
$S \to \varepsilon$

上下文有关文法: · $zS \to zaSb$

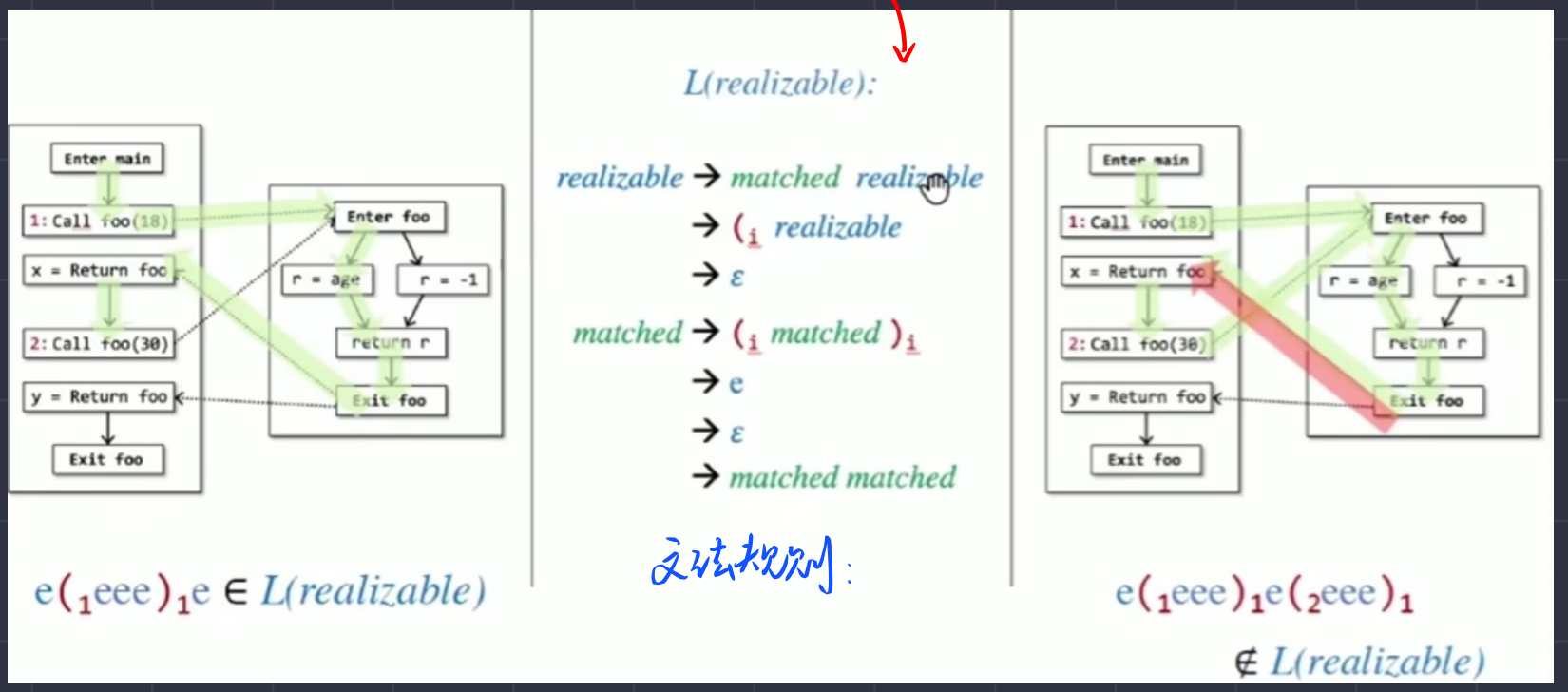
1. CFL-Reachability

括号匹配

CFL-Reachability

A path is considered to connect two nodes A and B, or B is reachable from A, only if the concatenation of the labels on the edges of the path is a word in a specified context-free language.

- A valid sentence in language L must follow L's grammar.
- A context-free language is a language generated by a context-free grammar (CFG).



2. IDFS: A Program Analysis Framework via Graph Reachability (无传播 → 图可达性)

用于过程间数据流分析 → (Interprocedual, Finite, Distribute, Subset Problem)

有限 $F(x \sqcap y) = F(x) \sqcap F(y)$

MOP ⇓
⇒ MRP solution

复习MOP: $pf_p = f_n \circ \dots \circ f_2 \circ f_1$

$$MOP_n = \bigsqcup_{p \in Paths(start, n)} pf_p(\bot)$$

$MRP_n = \bigcup_{p \in RPaths(start,n)} Pf_p(\bot)$ $MRP_n \sqsubseteq MOP_n$

IDFS

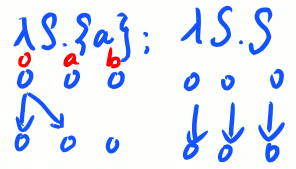
Overview of IFDS

Given a program P, and a dataflow-analysis problem Q

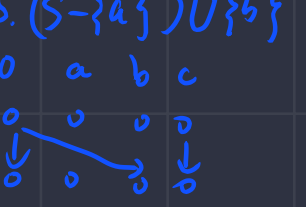
- Build a supergraph G* for P and define flow functions for edges in G* based on Q
- Build exploded supergraph $G^{\#}$ for P by transforming flow functions to representation relations (graphs)
- Q can be solved as graph reachability problems (find out MRP solutions) via applying Tabulation algorithm on $G^{\#}$

$O(ED^3)$

入表达式



以变量初始化为例



(有点记不过来了，可参考PPT或笔记网站)

representation relation R_f of flow function f: → 2(D+1)结点的图

$R_f \subseteq (D \cup 0) \times (D \cup 0)$

$R_f = \{(0,0)\}$

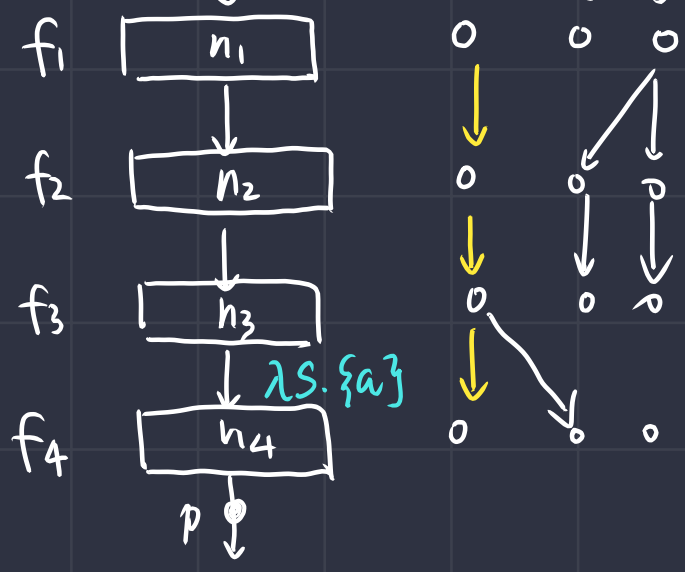
$\cup \{(0,y) \mid y \in f(\phi)\}$

$\cup \{(x,y) \mid y \notin f(\phi) \text{ and } y \in f(\{x\})\}$

Edge: $0 \to 0$

Edge: $0 \to d_1$

Edge: $d_1 \to d_2$



3. IFDS 中的 Distributivity (可分配性)

IFDS 不能直接用于常量传播、指针分析

Distributivity: $F(x \wedge y) = F(x) \wedge F(y)$, transfer function 只能处理一个值

结论：分析需要依赖多个输入 ⇒ 不能用 IFDS

十. Soundiness (2015年提出)

1. Soundness → 保守估计 所有程序行为(无论什么输入、运行行为)

Hard Language Features 静态分析难以分析的语言特征

例子：
- Java 反射、native code
- JavaScript eval、DOM
- C/C++ Pointer arithmetic、函数指针 → 可能指向的内存不确定

2. Soundiness

soundy analysis typically means analysis mostly sound, with well-identified unsound treatments to hard/specific language features.

3. Hard language features 以Java为例

1) Java Reflection 分析的噩梦，难缠的问题

反射：

```
Class Person {
    String name;
    void setName(String nm){...};
}
Person p = new Person();
p.setName("John");
```

绑定：Compile-time

Class
Method
Field

Meta-object

class.method.field

例：

```
Class c = Class.forName("Person");
Method m = c.getMethod("setName", ...);
Object p = c.newInstance();
```

m.invoke(a, ...);

等价 Run-time

反射有利于程序的松耦合

分析反射：String Constant analysis + Pointer Analysis 需要各种

⇓

使用中推导：利用反射语义和Java类型系统

2) Native Code

```
private native void writeBytes(byte b[], int off, int len)
        throws IOException.
```

声明，实现由C/C++语言完成

JNI Java Native Interface (A function module of JVM)

方法：手动建模

```
System.arraycopy(src, sp, dest, dp, ln);
```

⇓

```
for (int i=0; i<ln; i++)
    dest[dp+i] = src[sp+i];
```