一、PTA的规则

我们先针对上节中提到的除涉及方法调用的Pointer-affecting语句进行分析，涉及方法调用的之后再回来说.

(1) 符号约定

Variables:	$x, y \in V$	→ 一般属于某个实例
Fields:	$f, g \in F$	
Objects:	$o_i, o_j \in O$	→ 实例集
Instance fields:	$o_i.f, o_j.g \in O \times F$	
Pointers:	$\text{Pointer} = V \cup (O \times F)$	
Points-to relations:	$pt : \text{Pointer} \rightarrow \mathcal{P}(O)$	

- $\mathcal{P}(O)$ denotes the powerset of O
- $pt(p)$ denotes the points-to set of p

实际上我们要求解的就是pt映射，表征两类Pointer到具体object集的映射（也即O的某个子集）

(2) Rules

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

① i: x = new T()

没有条件，直接将新产生的对象 o_i 加入 $pt(x)$

② x = y

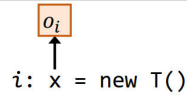
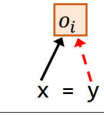
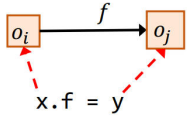
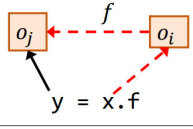
假如对象 o_i 属于集合 $pt(y)$，就把它加入到 $pt(x)$ 中

③ x.f = y

若 $o_i \in pt(x)$，$o_j \in pt(y)$，则 $o_j \in pt(o_i.f)$

④ y = x.f

若 $o_i \in pt(x)$，$o_j \in pt(o_i.f)$，则 $o_j \in pt(y)$

Kind	Rule	Illustration
New	$\frac{}{o_i \in pt(x)}$	
Assign	$\frac{o_i \in pt(y)}{o_i \in pt(x)}$	
Store	$\frac{o_i \in pt(x),\ o_j \in pt(y)}{o_j \in pt(o_i.f)}$	
Load	$\frac{o_i \in pt(x),\ o_j \in pt(o_i.f)}{o_j \in pt(y)}$	

二、如何进行PTA?

1、PTA的逻辑抽象

- Essentially, pointer analysis is to **propagate** points-to information among pointers (variables & fields)

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

一切信息的来源

本质就是在指针之间传播指向信息。

也即PTA实际上就是在节点间基于某种约束进行Points-to Information的传播

所谓的"闭包"，实际上就是由语句确定的一组规则约束，指集的变化一定可以传播到各集中。

→ 结构+闭包关系

对于约束结构我们可以使用图（PFG）来抽象

2、PFG的构成

Pointer flow graph of a program is a *directed graph* that expresses how objects flow among the pointers in the program.

(1) Nodes: $V \cup (O \times F)$

PFG的节点由程序中出现的局部变量、实例

成员构成，也即各种 Pointers（分析对象）

2）Edges：Pointers × Pointers

边 $x \to y$ 表征 x 的指向信息可能会流向 y，其本质上代表着由 Pointer-affecting 语句所确立的闭包约束。

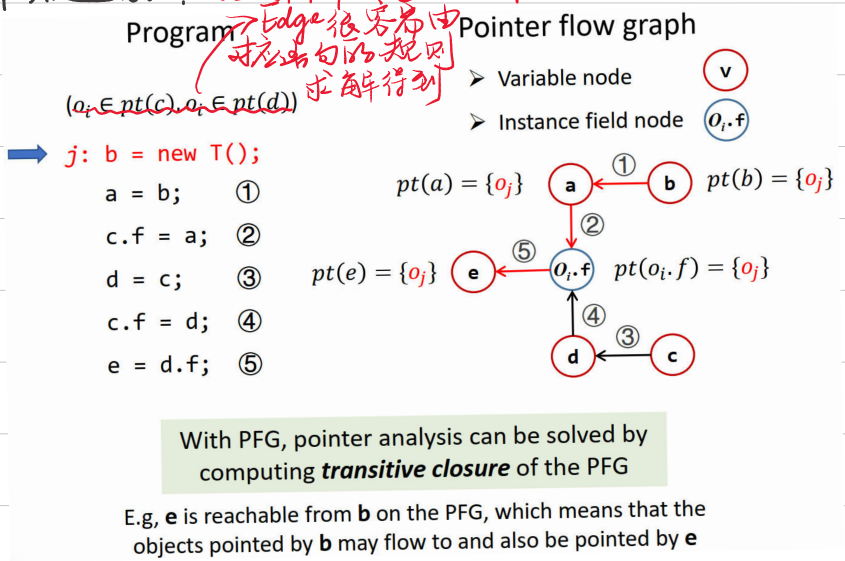
- PFG edges are added according to the statements of the program and the corresponding rules

Kind	Statement	Rule	PFG Edge
New	`i: x = new T()`	$\overline{o_i \in pt(x)}$	N/A
Assign	`x = y`	$\dfrac{o_i \in pt(\boldsymbol{y})}{o_i \in pt(\boldsymbol{x})}$	$x \leftarrow y$
Store	`x.f = y`	$\dfrac{o_i \in pt(x),\ o_j \in pt(\boldsymbol{y})}{o_j \in pt(\boldsymbol{o_i.f})}$	$o_i.f \leftarrow y$
Load	`y = x.f`	$\dfrac{o_i \in pt(x),\ o_j \in pt(\boldsymbol{o_i.f})}{o_j \in pt(\boldsymbol{y})}$	$y \leftarrow o_i.f$

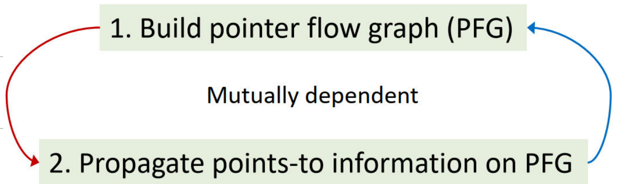
由程序很容易获得 PFG 的节点集，关键建在于如何根据 Pointer-affecting 语句导出边集。你可能会说这还不容易，根据节点集直接梭哈不就行了吗？但很不幸，PFG中的节点由变量（v）与实例成员（$o_i.f$）构成，前者可以直接从语句中提取，后者的确立则需要 $x.f$ 中 x 的指向信息，而这个信息在 PTA 之前我们是

不知道的！

给定所需实例的指向信息，→Edge很容易由对应语句的规则求解得到



因此PFG构建的难点，就在于涉及x.f的语句该如何处理？（pt(x)随着分析的迭代而变化，则PFG也会随之动态变化，从而又会影响信息传递的迭代）。



构建PFG与在PFG上传播信息是一个相互影响的整体的过程。

3. 算法

```
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)
```

```
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

(1) S是包含程序中所有语句的无序集

Flow-Insensitive

(2) Worklist存储待接收的指向信息pts和指向对象n.

- Worklist contains the points-to information to be processed
  - $WL \subseteq \langle \text{Pointer}, \mathcal{P}(O) \rangle^*$
- Each worklist entry $\langle n, pts \rangle$ is a pair of pointer n and points-to set pts, which means that pts should be propagated to $pt(n)$
  - E.g., $[\langle x, \{o_i\} \rangle, \langle y, \{o_j, o_k\} \rangle, \langle o_j.f, \{o_l\} \rangle \dots]$

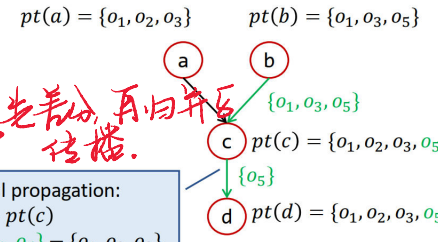
(3) Differential Propagation

注意到在中WL中的待传播项进行处理时，

我们实际传播的信息是 $\Delta = pts - pt(n)$，也即原本的 $pt(n)$ 中没有的指向信息。虽然后面进行 Propagate 操作时进不进行这步去重不会对算法的正确性产生影响，但是对于 Propagate 中的①归并 pts 至 $pt(n)$ 中②传播 pts 至 n 的所有后继节点，减少 pts 的规模无疑可以大大提高算法效率。

- ***Differential propagation*** is employed to avoid propagation and processing of redundant points-to information
- **Insight**: existing points-to information in $pt(n)$ have already been propagated to n's successors, and ***no need*** to be propagated again

Solve(S)
…
while WL is not empty **do**
  remove $\langle n, pts\rangle$ from WL
  $\Delta = pts - pt(n)$
  Propagate(n, Δ)
…



In practice, Δ is usually small compared with the original set, so propagating only the new points-to information (Δ) improves efficiency

Differential propagation:
$\Delta = pts - pt(c)$
$= \{o_1, o_3, o_5\} - \{o_1, o_2, o_3\}$
$= \{o_5\}$

64

(4) Load/Store 语句

对于 New 语句，我们简单地将初始化信息加入 WL 中即可；对于 Assign 语句，加条边即可。难点在于涉及 x.f 的语句该如何处理？我们的想法很简单，当指向信息传播到代表变量 x 的节点，

n时，n指向的Object可能会发生变化，因此对所有涉及x.f的语句，我们都需要在Δ中的所有O_i上增加对应的边。

☆算法总结

①对于无序语句集S，先处理其中的New语句与Assign语句：

New——添加初始指向信息进入WL

Assign——增加新边(同时添加初始推广信息入WL)

②之后对WL中各项进行处理<n, pts>

A.首先算出pts相较于n原有指向信息的增益Δ

B.然后归并Δ入pt(n)中(同时添加次级推广信息入WL)

C.最终如果n表征的变量x在S中涉及x.f，在pt(n)新增加的可能性指向Δ上，根据相关语句增加新边(同时添加初始推广信息入WL)

三、含有 Method Calls 的 PTA

回顾之前学到的 Inter-procedural Analysis，其关键点在于 Call Graph 的构建（可以基于此得到 ICFG，进而进行 IDFA），而构建 CG 的关键点又在于如何求解 call site → target method（核心在于 virtual call 的求解），其中如何估计动态分发的起始点又是重中之重。

CHA：用 declared Type 估计起始点。

PTA：用 pta 近似起始点，更准确。

1) Rules

Kind	Statement	Rule	PFG Edge
Call	l: r = x.k(a1,…,an)	① $o_i \in pt(x),\ m = \text{Dispatch}(o_i, k)$ $o_u \in pt(aj), 1 \le j \le n$ $o_v \in pt(m_{ret})$ ② $o_i \in pt(m_{this})$ ③ $o_u \in pt(m_{pj}), 1 \le j \le n$ ④ $o_v \in pt(r)$	$a1 \rightarrow m_{p1}$ … $an \rightarrow m_{pn}$ $r \leftarrow m_{ret}$

- Dispatch(o_i, k): resolves the virtual dispatch of k on o_i to a target method (based on type of o_i)
- m_{this}: this variable of m
- m_{pj}: the j-th parameter of m
- m_{ret}: the variable that holds the return value of m

```
C x = new T();
…
r = x.foo(a1, a2);

class T … {  …
  B foo(A p1, A p2) {
    this…
    return ret;
  }}
```

上面处理 call site 处指向信息流动的规则实际上与方法调用的机制是十分相似的。

① 基于 Receive Object 的指向信息 + 方法签名确定其 target method.

② 确定目标方法 m 的 this 指针的指向。

③ 在 cs 处的参数与 m 处的参数间构建 PFG 边，使得前者的指向信息可以流向后者。

④ 在 m 的返回值与 cs 处的左值间构建新边，使得前者的指向信息可以流向后者。

Ps: 之所以②没有像③④一样构建新的 PFG 边使得信息传导过去，是由于 this 指针所指向的对象实际上就是 target method 所属类的实例，而 target method 所属的类在 Dispatch 时已经完全确定了（要不然怎么能说 target method 是确定的呢？）因此为了精确，我们直接传过去？

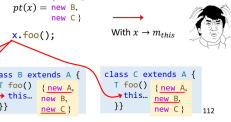
Rule: Call

Why not add PFG edge $x \rightarrow m_{this}$?

Kind	Statement	Rule	PFG Edge
Call	l: r = x.k(a1,…,an)	$o_i \in pt(x),\ m = \text{Dispatch}(o_i, k)$ $o_w \in pt(a_j), 1 \le j \le n$ $o_x \in pt(m_{ret})$ $o_i \in pt(m_{this})$ $o_w \in pt(m_{pj}), 1 \le j \le n$ $o_x \in pt(r)$	$a1 \rightarrow m_{p1}$ … $an \rightarrow m_{pn}$ $r \leftarrow m_{ret}$

Receiver object should only flow to this variable of the corresponding target method

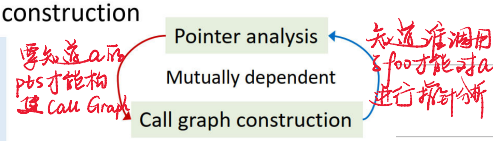
PFG edge $x \rightarrow m_{this}$ would introduce spurious points-to relations for this variables



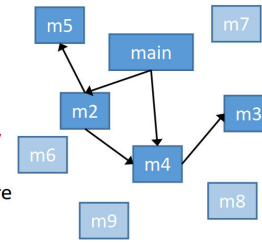
(2) Interprocedural PTA (IPTA)

- Run together with call graph construction

```
void foo(A a) {
    …
    b = a.bar();
    …
}
```



- Call graph forms a "reachable world"
  - Entry methods (e.g., the main method) are reachable from the beginning
  - The other reachable methods are gradually discovered during analysis
  - Only reachable methods and statements are analyzed



实际上就是从入口方法(main)开始,不断扩张,分析可达世界。①只分析可达世界,效率自然更高②避免了不可达方法的干扰,因此精度更高。

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)

AddReachable(m)
  if $m \notin RM$ **then**
    add m to RM
    S ∪= 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)

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