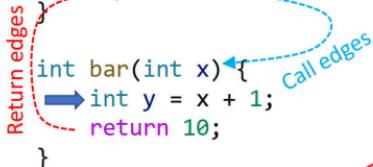


在之前的分析中，我们假设没有出现方法调用。如果要在 Intraprocedural Analysis 中引入方法调用，为了 Safe，我们只能进行最保守的估计：

### Constant Propagation

```
void foo() {  
    int n = bar(42);  
}  
  
int bar(int x) {  
    int y = x + 1;  
    return 10;  
}
```



使得信息能在方法间进行传递

So far, all analyses we learnt are **intraprocedural**. How to deal with method calls?

- Make the **most conservative assumption** for method calls, for safe-approximation
- Source of **imprecision** 没有信息流, 且保留了抛值的效果
- $x = \text{NAC}$ ,  $y = \text{NAC}$
- $n = \text{NAC}$

For better precision, we need **Interprocedural analysis**: propagate data-flow information along **interprocedural control-flow edges** (i.e., call and return edges)

- $x = 42$ ,  $y = 43$
- $n = 10$



这样做会造成长大的精度损失。于是我们可以构建方法间的有向边来传递信息，这就是为什么我们需要 Call Graph.

## 1. Call Graph Construction

### 1.1 Call Graph 定义

表征程序内调用关系的有向图。

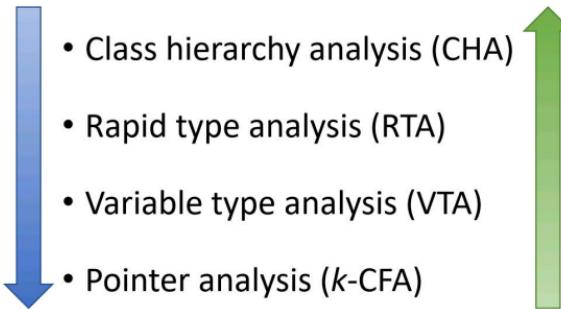
Call-sites  $\Rightarrow$  target method 怎样确定？

## ② 构建方法

# Call Graph Construction for OOPs (focus on Java)

几种针对OOP的构建方法

More efficient

- 
- Class hierarchy analysis (CHA)
  - Rapid type analysis (RTA)
  - Variable type analysis (VTA)
  - Pointer analysis ( $k$ -CFA)

More precise

## ③ Java中的方法调用

	Static call	Special call	Virtual call
Instruction	invokestatic	invokespecial	invokeinterface invokevirtual
Receiver objects	×	✓	✓
Target methods	• Static methods	• Constructors • Private instance methods • Superclass instance methods	• Other instance methods
#Target methods	1	1	$\geq 1$ (polymorphism)
Determinacy	Compile-time	Compile-time	Run-time

编译时可能的目标方法会有多个  
具体要到运行时才能够确定。

因此关键点就在于怎样处理 virtual call,

Static/Special Call 的目标方法确定且唯一，很好处理

## ★ 像解 Virtual Call To Method Dispatch

D.foo(...)

运行时，基于两个因素确定被调用的具体方法：

① Receive Object O 的具体指向的类型

② foo 方法的方法签名

+ 层次结构

```
class C {  
    T foo(P p, Q q, R r) { ... }
```

Java方法必须  
依赖于类

C.foo(P, Q, R) for short

- Signature = class type + method name + descriptor
- Descriptor = return type + parameter types

我们可以定义 Dispatch(c, m) 来表示这个过程：

$Dispatch(c, m) = \begin{cases} m' & c \text{ 中有 name+descriptor} \text{ 且} \\ & \text{和 } m \text{ 一样的非抽象方法} \\ Dispatch(c', m) & c' \text{ 是 } c \text{ 的父类} \end{cases}$

签名中的类只是为唯一全局标注方法所加上所属，我们在对比时，使用 method name + descriptor 就足够了，因为方法具体所属类实际上是由起始点 + 类层次结构确定的。为类层次结构中起始点向上找

到的最近的那个。

#### (4) Class Hierarchy Analysis (CHA)

CHA 根据程序中的类继承结构的信息来求解各个 Virtual Call 的 target method, 从而得到 Call Graph.

Resolve( $cs$ )

$T = \{\}$  所有可能的 target Method

$m = \text{method signature at } cs$

**if**  $cs$  is a static call **then** 直接返回

$T = \{m\}$

**if**  $cs$  is special call **then** 因为仅类调用可能会产生多态, 所以从当前类型开始进行动态内存分发  
 $c^m = \text{class type of } m$

$T = \{\text{Dispatch}(c^m, m)\}$

**if**  $cs$  is a virtual call **then**

$c = \text{declared type of receiver variable at } cs$

**foreach**  $c'$  that is a subclass of  $c$  or  $c$  itself **do**

add  $\text{Dispatch}(c', m)$  to  $T$

**return**  $T$

对所有子类都掌动态分发

① 写出 Call Site 的函数签名

② 确定调用的类型

③ 找到进行 Dispatch 的初台点,

Special Call: 针对父类方法调用, 从父类开始

Virtual Call: 从 declared Type 的本身 + 所有子类开始

④ 对所有初始点，向上搜索 name + descriptor 符合签名的非抽象函数。

Ps: ① 优点：速度快

只考虑签名信息和程序的继承结构，忽略数据流与控制流信息。  
→毕竟没有真正动起来

② 缺点：精度低

会产生较多的误报。

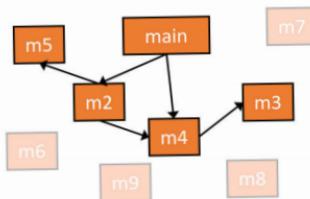
总体来看比较适合用于 IDE 中。

### 5) Call Graph 的构建

上面讲到的 CHA 只是针对单个 call-site 的处理，要构建 Call Graph，我们需要确定程序内所有方法的所有 call-sites to target method.

Build call graph for whole program via CHA

- Start from entry methods (focus on main method)
- For each reachable method  $m$ , resolve target methods for each call site  $cs$  in  $m$  via CHA ( $\text{Resolve}(cs)$ )
- Repeat until no new method is discovered



对这些方法的所有 cs 进行求解

具体的算法：

BuildCallGraph( $m^{entry}$ )

$WL = [m^{entry}]$ ,  $CG = \{\}$ ,  $RM = \{\}$

while  $WL$  is not empty do

remove  $m$  from  $WL$

if  $m \notin RM$  then

add  $m$  to  $RM$

foreach call site  $cs$  in  $m$  do

$T = Resolve(cs)$

foreach target method  $m'$  in  $T$  do

add  $cs \rightarrow m'$  to  $CG$

add  $m'$  to  $WL$

return  $CG$

→ 表征可以扩张到  $m$  的方法处理它

$WL$  Work list, containing the  
methods to be processed 需要被处理的方法

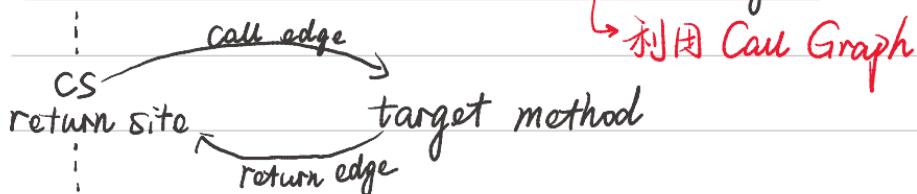
$CG$  Call graph, a set of call edges  $cs \rightarrow$  target method

$RM$  A set of reachable methods 表征方法有没有被扩张到

## 2. Interprocedural CFG

CFG 表征的是单个方法的结构，ICFG 则表征整个程序的结构，其包含连接各个方法的信息。

$ICFG = CFGs + Call \& Return edges$



```

static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

```

```

static
int addOne(int x) {
    int y = x + 1;
    return y;
}

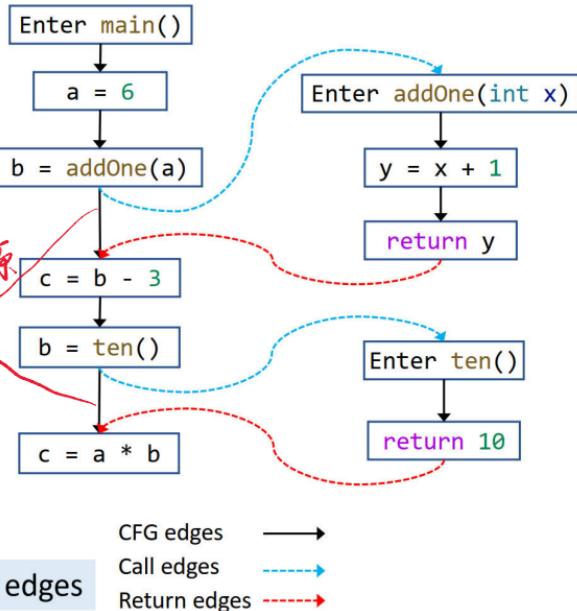
```

```

static int ten() {
    return 10;
}

```

$$\text{ICFG} = \text{CFGs} + \text{call \& return edges}$$



### 3. Interprocedural DFA

	Intraprocedural	Interprocedural
Program representation	CFG	$\text{ICFG} = \text{CFGs} + \text{call \& return edges}$
Transfer functions	Node transfer	Node transfer + edge transfer

唯一不同的是不仅 Node 而且具有 transfer function,  
Edge 上面也具有:

- ① Node: 普通 Node 不变; Call Node 当它不存在
- ② Edge: 普通 Edge 当它不存在; CTS Edge kill 指受影响的变量;

Ps: ① 的本质上是分离 local-dataflow 与受调用影响的变量。如果没有 kill 操作, 若原边上

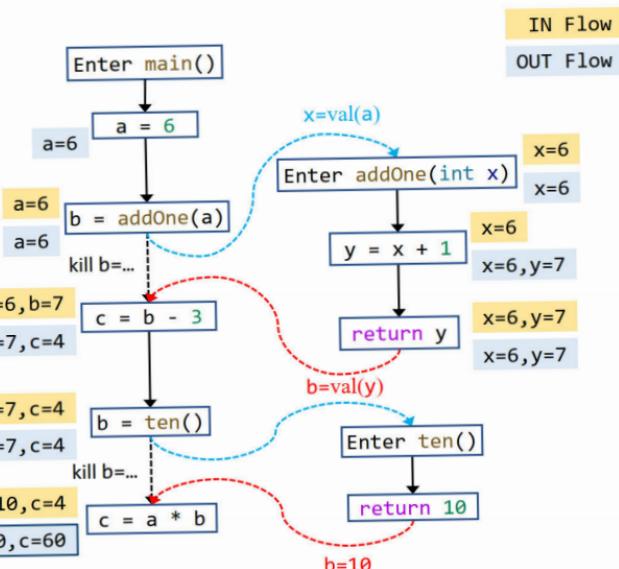
受调用影响的变量的值与返回边上的值不同，则会产生精度损失；如果没有①中的 CTS Edge，local-data flow 就要跟着受调用影响的变量一起出去走一圈，效率太低了。

## Interprocedural Constant Propagation: An Example

```
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}
```

```
static
int addOne(int x) {
    int y = x + 1;
    return y;
}
```

```
static int ten() {
    return 10;
}
```

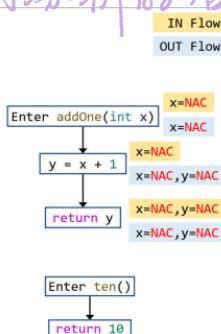


对比仅使用过程中分析的结果：

```
static void main() {
    int a, b, c;
    a = 6;
    b = addOne(a);
    c = b - 3;
    b = ten();
    c = a * b;
}

static
int addOne(int x) {
    int y = x + 1;
    return y;
}

static int ten() {
    return 10;
}
```



相当于只在 CTS 边  
kill 了受调用影响  
的变量，而没有 call &  
Return Edge 的信息流