第十一章 过程间分析

冯 洋

过程（Procedures）

- 我们已经讨论了过程内分析（intraprocedural analysis）
  - 仅仅是分析一个过程
- 过程间分析（interprocedural analysis ）被用于进一步优化函数间的调用
  - 可以使得分析结果更加精确

调用图（Callgraph）

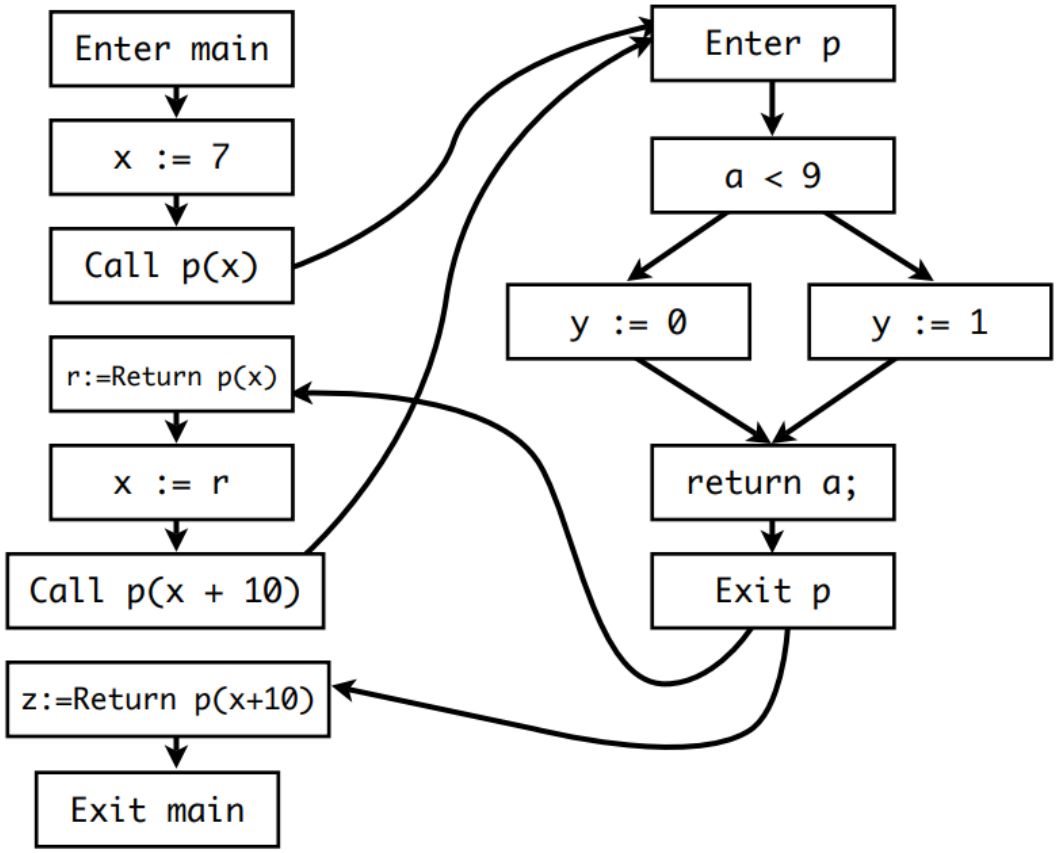
- 程序的调用图，是一个结点和边的集合，并满足：
  - 1. 程序中的所有过程，都有一个结点
  - 2. 对于每个调用点（call site），都有一个结点。所谓的调用点，就是程序中调用某个过程的一个位置
  - 3. 如果调用点c 调用了过程p，就存在一条从c的结点到p的结点的边；

调用图（Callgraph）

- 第一个问题：我们如何知道什么过程在什么地方被调用？
  - 在某些语言中，该问题特别复杂，我们暂时不考虑
  - 本节讨论的所有内容，均假设，我们已经有了一个**静态调用图**

过程间的数据流分析

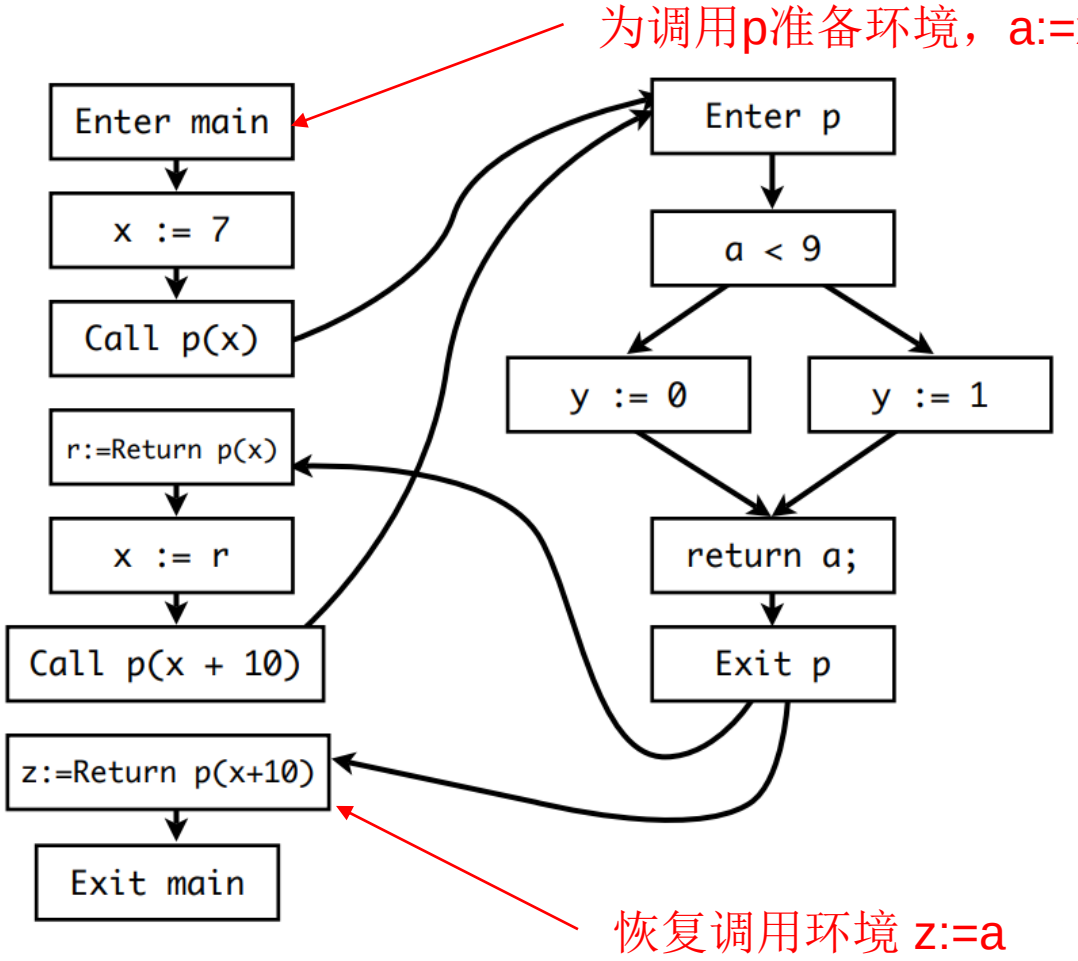
```
main(){
        x := 7;
        r := p(x);
        x := r;
        z := p(x + 10);
}
p(int a){
        if (a < 9)
                y := 0;
        else
                y := 1;
        return a;
}
```



过程间的数据流分析

最简单的思路：构建一个巨大的，CFG

```
main(){
        x := 7;
        r := p(x);
        x := r;
        z := p(x + 10);
}
p(int a){
        if (a < 9)
                y := 0;
        else
                y := 1;
        return a;
}
```

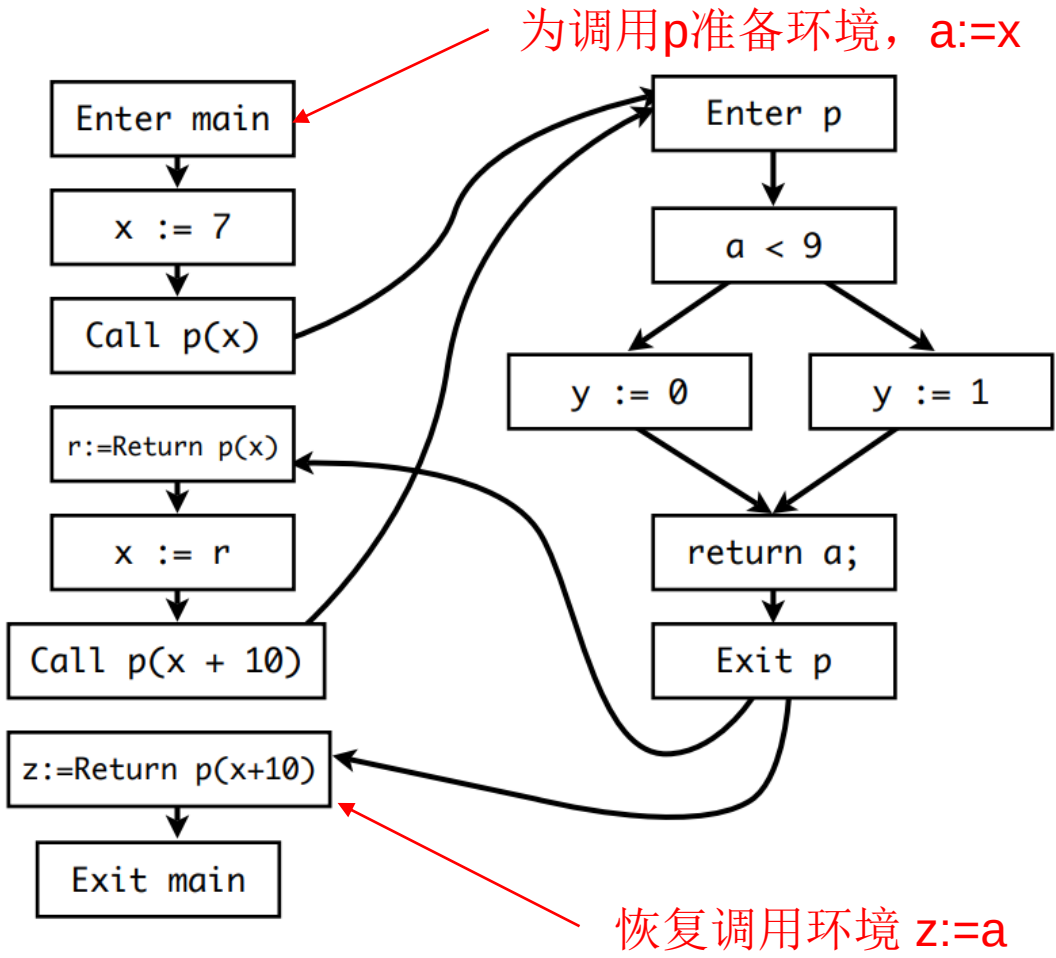


过程间的数据流分析

- CFG 可能需要额外的结点来 handle 函数调用和返回
  - Treat arguments, return values as assingments;
- 一个局部程序变量，可能因为多次调用而代表不同位置的值
- 问题：数据流通过一个调用点污染其他调用点
  - 上面的例子中，p是一个数据流的汇集点

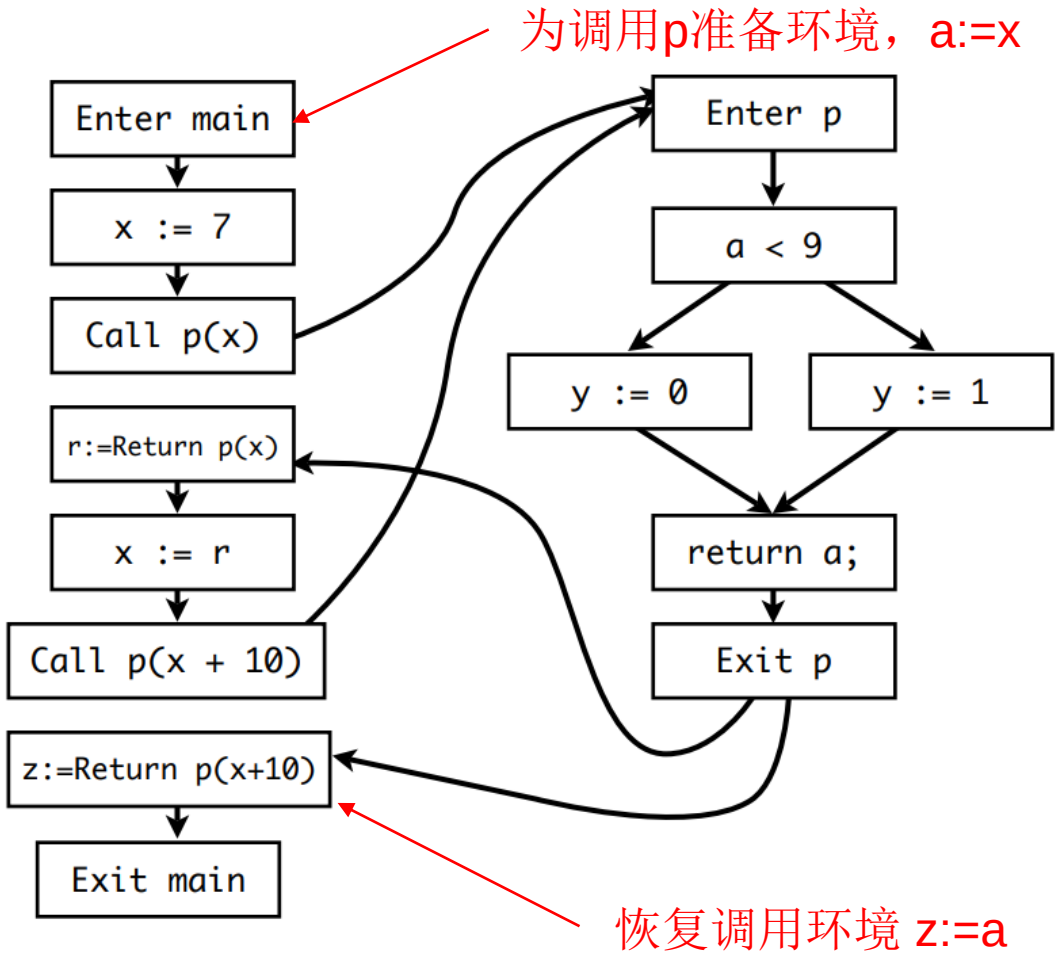
过程间的数据流分析

- CFG 可能需要额外的结点来 handle 函数调用和返回
  - Treat arguments, return values as assingments;
- 一个局部程序变量，可能因为多次调用而代表不同位置的值
- 问题：数据流通过一个调用点污染其他调用点
  - 上面的例子中，p是一个数据流的汇集点



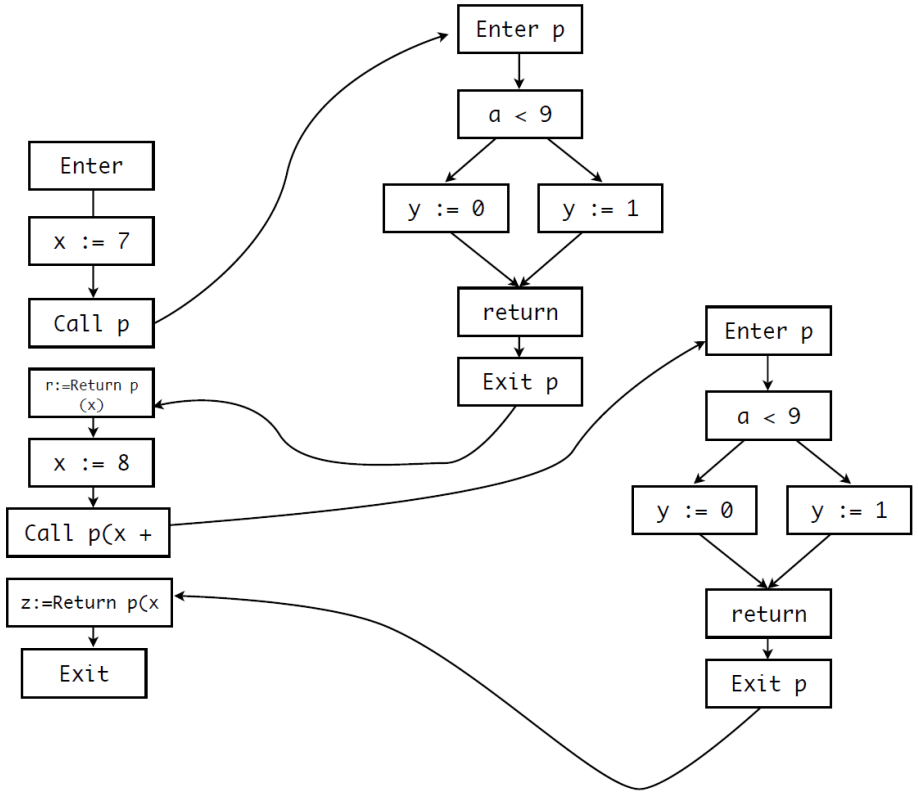
过程间的数据流分析

- CFG 可能需要额外的结点来 handle 函数调用和返回
  - Treat arguments, return values as assingments;
- 一个局部程序变量，可能因为多次调用而代表不同位置的值
- 问题：数据流通过一个调用点污染其他调用点
  - 上面的例子中，p是一个数据流的汇集点



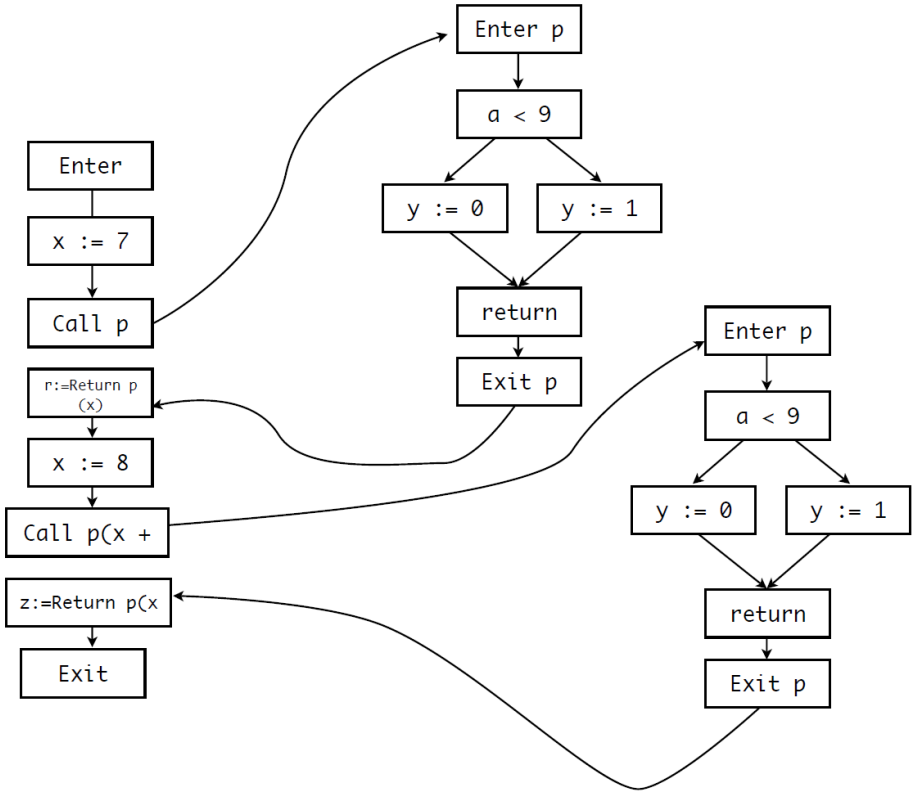
内联（Inlining）

- 内联
  - 在每个CFG的调用点，生成一个被调用函数的**副本**



内联（Inlining）

- 内联
  - 在每个CFG的调用点，生成一个被调用函数的**副本**
- 可能存在的问题
  - 代价可能很高？会随着CFG的大小成指数级增长。
  - 递归如何处理？

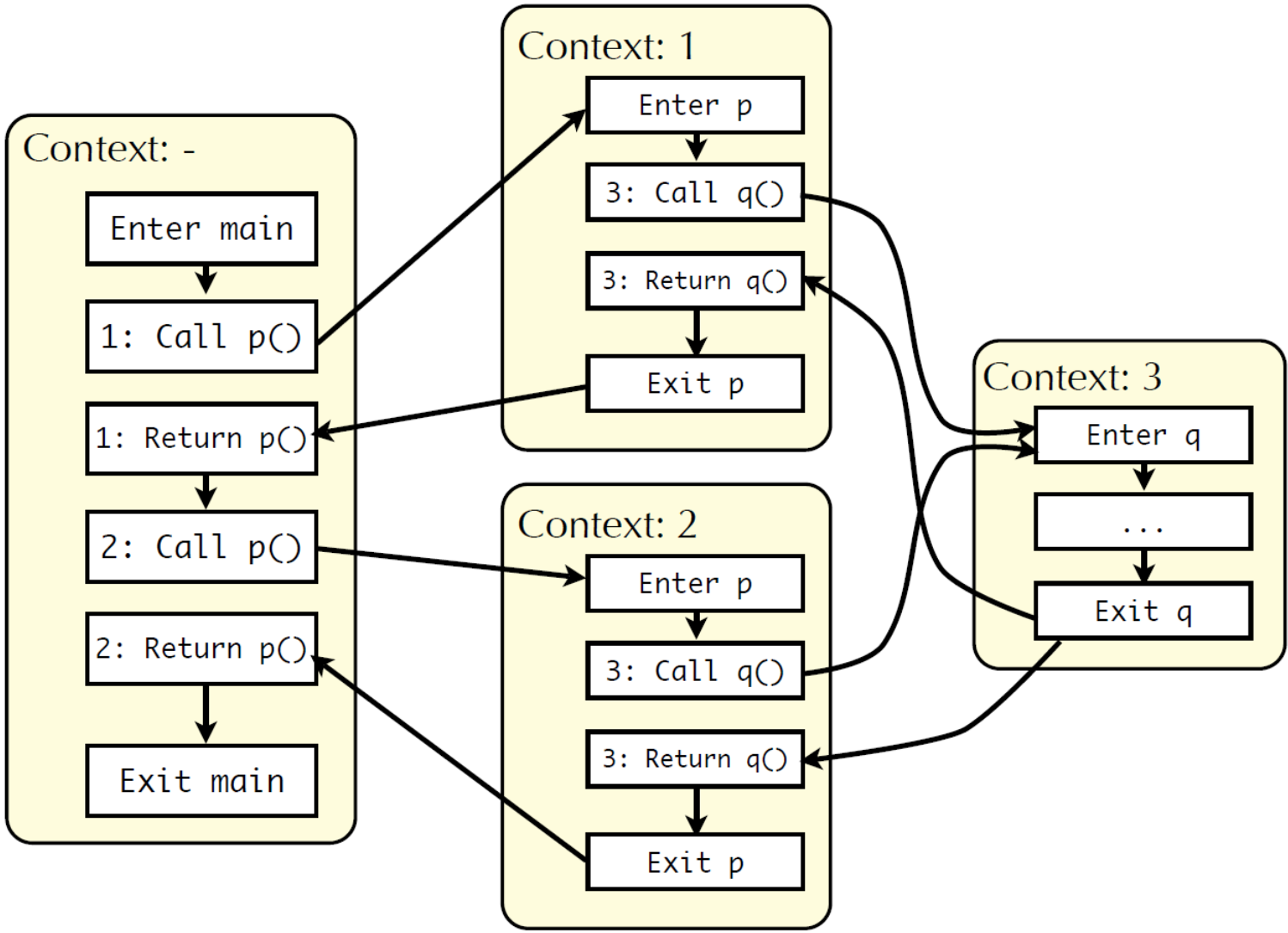


解决方案：上下文敏感

- 上线文敏感（Context Sensitivity）
- 通过构建**有限数量**的副本
- 使用上下文信息决定什么时候、什么状态需要生成一个副本
- 选择是用什么信息，是一个精度（precision）和可扩展性（scalability）

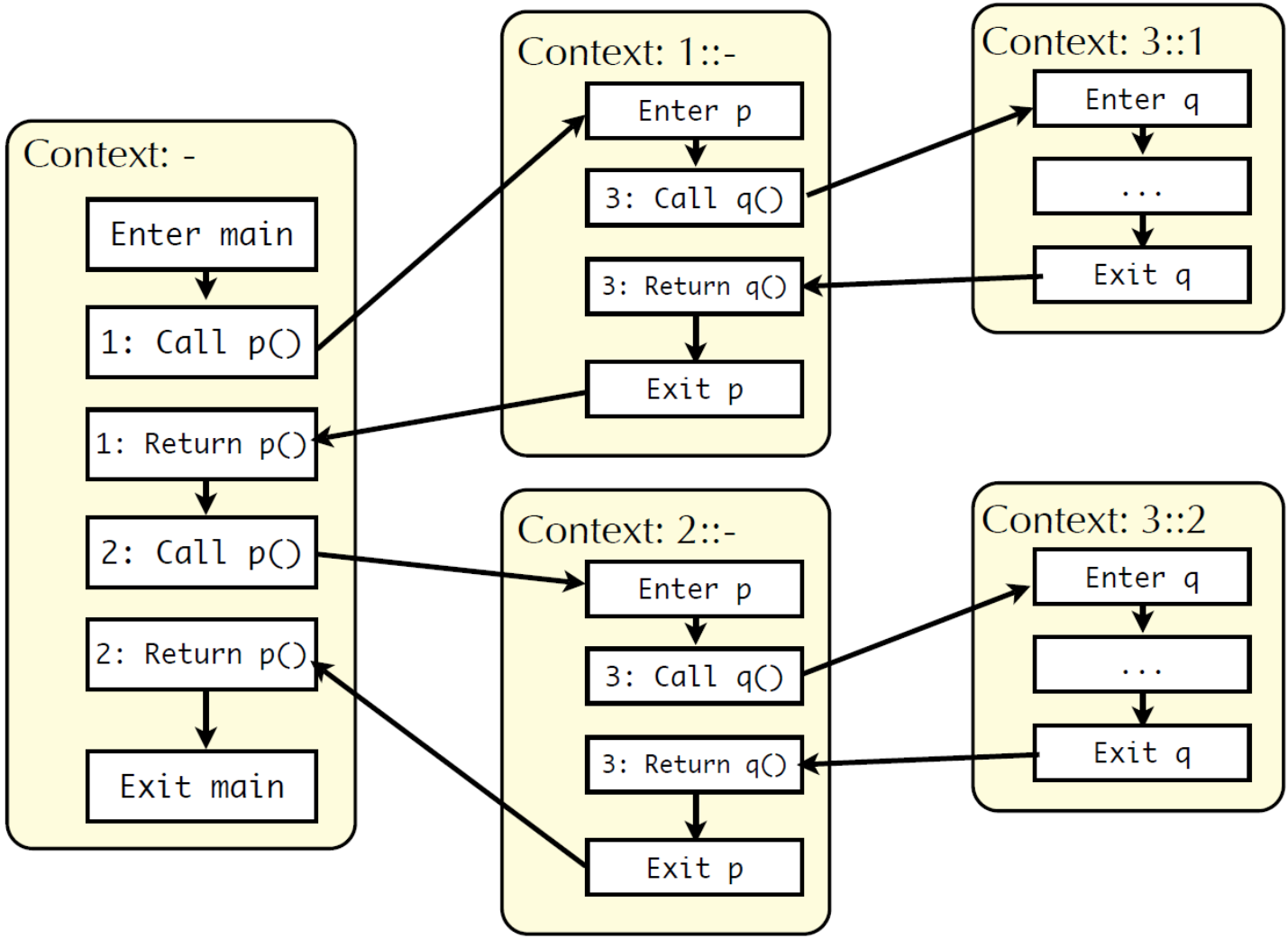
解决方案：上下文敏感

```
main() {
  1: p();
  2: p();
}

p() {
  3: q();
}
q() {
  …
}
```



解决方案：上下文敏感

```
main() {
  1: p();
  2: p();
}

p() {
  3: q();
}
q() {
  …
}
```

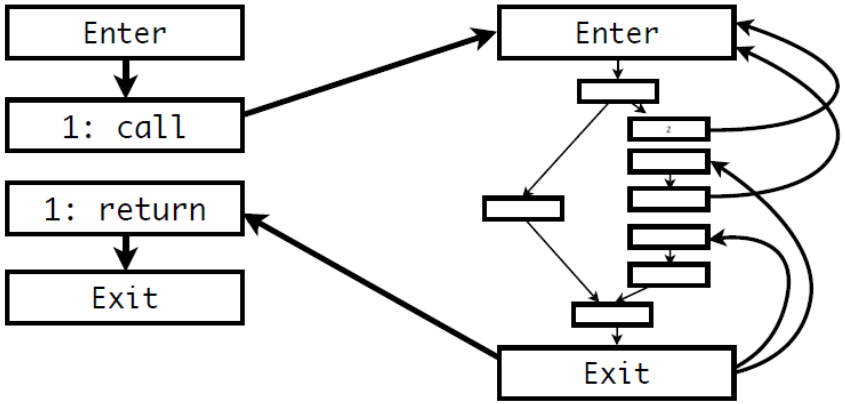


一个简单的例子

```
main(){
  1：       fib(7);
}
fib(int a){
            if (n <= 1)
                        x := 0;
            else
   2:                   y := fib(n-1);
   3:                   z := fib(n-2);
                        x := y + z;
            return x;
}
```

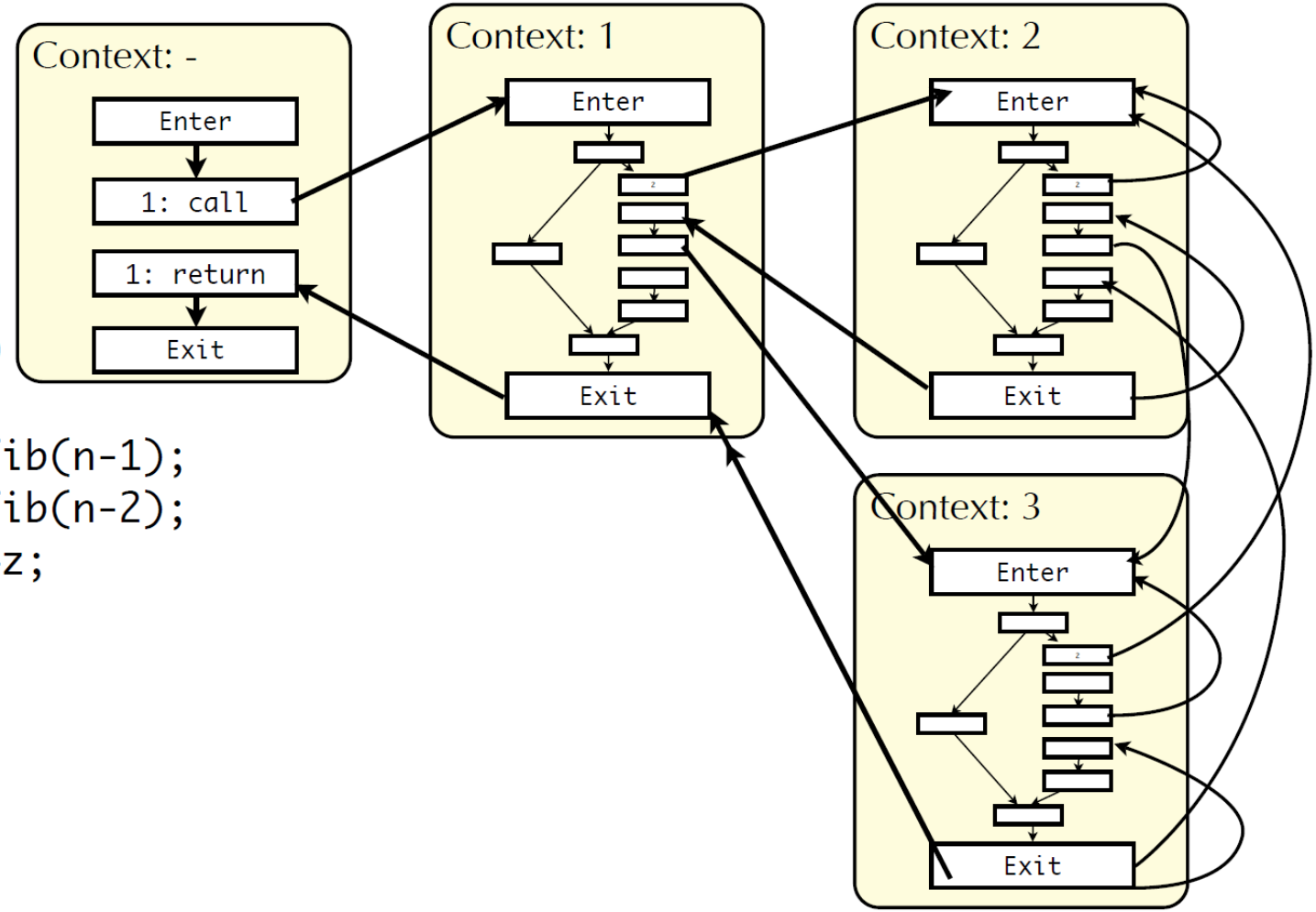
一个简单的例子

```
main(){
  1：        fib(7);
}
fib(int a){
            if (n <= 1)
                        x := 0;
            else
   2:                   y := fib(n-1);
   3:                   z := fib(n-2);
                        x := y + z;
            return x;
}
```



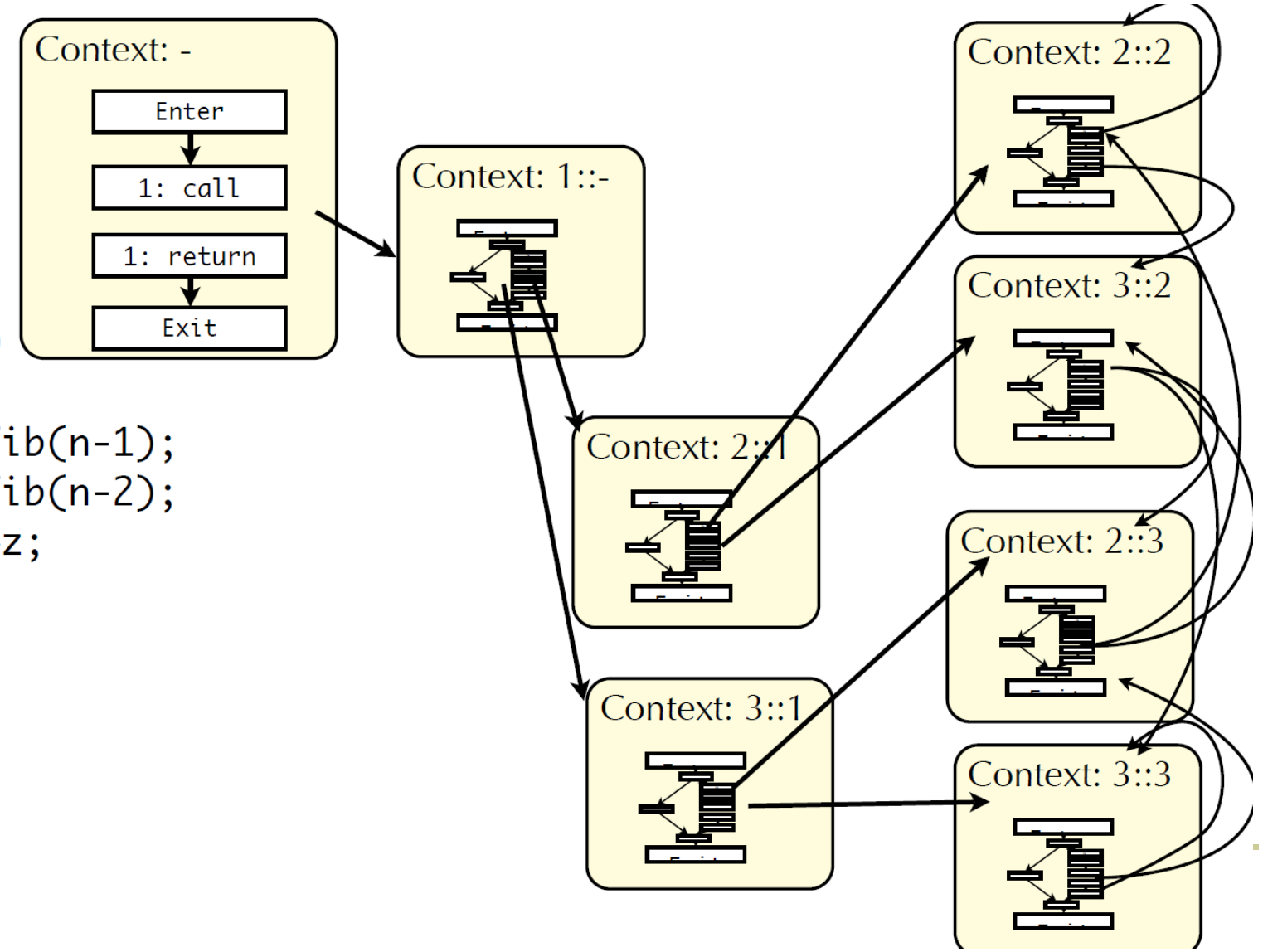
一个简单的例子—栈深度为1

```
main() {
  1: fib(7);
}

fib(int n) {
  if n <= 1
       x := 0
  else
    2: y := fib(n-1);
    3: z := fib(n-2);
       x:= y+z;
  return x;
}
```



一个简单的例子—栈深度为2

```
main() {
  1: fib(7);
}

fib(int n) {
  if n <= 1
       x := 0
  else
    2: y := fib(n-1);
    3: z := fib(n-2);
       x:= y+z;
  return x;
}
```



其他的上下文信息（context）

- 上下文敏感信息可以区分出同一个过程的不同调用
  - 对于敏感信息的选择，可以决定哪些调用会被区分出来
- 其实可以选择其他的上线文信息
  - 调用方的栈信息
    - 与调用点栈信息相比，精度较低
    - 例如，上面的例子中上下文 “2::2” 与 “2::3” 均会被转化为 “fib::fib”
  - 对象敏感信息：哪个对象是方法调用的目标
    - 用于面向对象语言
    - 对于一些常见的面向对象patterns，可以维持较好的精度
    - 需要使用指针分析以获得可能是目标的对象

其他的上下文信息（context）

- 上下文敏感信息可以区分出同一个过程的不同调用
  - 对于敏感信息的选择，可以决定哪些调用会被区分出来
- 其实可以选择其他的上线文信息
  - 调用方的栈信息
    - 与调用点栈信息相比，精度较低
    - 例如，上面的例子中上下文 “2::2” 与 “2::3” 均会被转化为 “fib::fib”
  - 对象敏感信息：哪个对象是方法调用的目标
    - 用于面向对象语言
    - 对于一些常见的面向对象patterns，可以维持较好的精度
    - 需要使用指针分析以获得可能是目标的对象
  - 更多的选择
    - 多种上线文信息的组合

过程摘要（Procedure Summaries）

- 实践中，通常不会构建单个的CFG并进行数据流分析
- 通过对过程进行摘要，用以后续的分析
- 当在上下文C中，通过输入 D 调用 p，我们可以先检查 p 的过程摘要是否在上下文C中存在

过程摘要（Procedure Summaries）

- 实践中，通常不会构建单个的CFG并进行数据流分析
- 通过对过程进行摘要，用以后续的分析
- 当在上下文C中，通过输入 D 调用 p，我们可以先检查 p 的过程摘要是否在上下文C中存在
  - 如果不存在，则在上下文C中，通过D，正常处理 p
  - 如果存在，发现该摘要是用于输入 M 的，并且与M得到了输出E，则
    - 如果 M 是 D 的子集，那么我们可以直接使用输出 E
    - 如果 M 不是 D 的子集，那么我们只需要处理 D - M
  - 如果摘要有所改变，必须重新处理摘要
  - 对递归的处理依然是一个重要问题

过程间分析的应用

- 为什么要使用过程间分析
  - 虚方法的调用
  - 指针别名分析
  - 软件错误和漏洞的（自动化）检测
  - SQL 注入检测
  - 缓冲区溢出检测

过程间分析的应用

- 虚方法（Virtual Function）的调用
  - 面向对象程序设计中（运行时）多态的重要组成部分
  - 虚函数可以给出目标函数的定义，但该目标的具体指向在编译期可能无法确定
  - 不同语言有不同的定义
    - C++：Virtual 关键字
    - Java： 所有的方法默认都是"虚函数". 只有以关键字 final 标记的方法才是非虚函数.
    - C#：对基类中的任何虚方法必须用 virtual 修饰，而派生类中由基类继承而来的重载方法必须用 override 修饰
  - 通过上下文分析，完成内联

过程间分析的应用

- 指针别名分析
- 多个指针是否可坑互为别名？
  - 我们的例子

```
*p = 1;
*q = 2;
x = *p;
```

如果不知道p与q是否可能指向同一位置，也就是说，他们是否可能互为别名，那么就不能确定x在基本块的结尾处是否为1。

软件错误和漏洞的（自动化）检测

- 静态分析可用于检测代码中是否存在常见的错误模式
- 检测工具的可靠性（或译为健全性，Soundness）与完备性（Completeness）
  - Soundness：对程序进行了over-approximate过拟合，不会漏报（有false positives误报）
  - Completeness：对程序进行了under-approximate欠拟合，不会误报（有false negatives漏报）
  - 当前所有的检查工具既不是可靠的也不是完备的：即，既不能够找到所有的错误，也不是报告的所有警告都是错误
  - 一个矛盾：如果让工具尽可能地报告所有的可疑错误，那么大量的假报警（False Positive）会使得工具无法使用；如果采用尽量保守的做法，那么很可能漏报一些真实存在的错误

SQL 注入检测

- 一种极为常见的攻击方法
- 例子：
  - sql=’select * from users where user=’&user&’ and passwd=’&passwd& ‘
  - 以 admin 为用户名；‘1’ or ‘a’=‘a’ 为密码，则以上SQL 转化为：

select * from users where user= ‘admin’ and passwd=’1 ‘ or ’a’=’a’

缓冲区溢出检测

- 当一个由用户提供的，精心制作的，数据被写到了预想的缓冲区之外，并操纵程序的执行时，就发生了缓冲区溢出攻击（buffer overflow attack）
- 例子：

```
void foo(void){
    int a, *p;
    p = (int*)((char *)&a + 12);  //让p指向main函数调用foo时入栈的返回地址，等效于p = (int*)(&a + 3);
    *p += 12;   //修改该地址的值，使其指向一条指令的起始地址
 }
int main(void){
    foo();
    printf("First printf call\n");
    printf("Second printf call\n");
    return 0;
}
```

缓冲区溢出检测

- 例子：

```
void foo(void){
    int a, *p;
    p = (int*)((char *)&a + 12);  //让p指向main函数调用foo时入栈的返回地址，等效于p = (int*)(&a + 3);
    *p += 12;    //修改该地址的值，使其指向一条指令的起始地址
 }

int main(void){
    foo();
    printf("First printf call\n");
    printf("Second printf call\n");
    return 0;
}
```

缓冲区溢出检测

- 例子：

```
void foo(void){
    int a, *p;
    p = (int*)((char *)&a + 12);  //让p指向main函数调用foo时入栈的返回地址，等效于p = (int*)(&a + 3);
    *p += 12;    //修改该地址的值，使其指向一条指令的起始地址
 }

int main(void){
    foo();
    printf("First printf call\n");
    printf("Second printf call\n");
    return 0;
}
```

结果输出Second printf call，未输出First printf call