软件分析

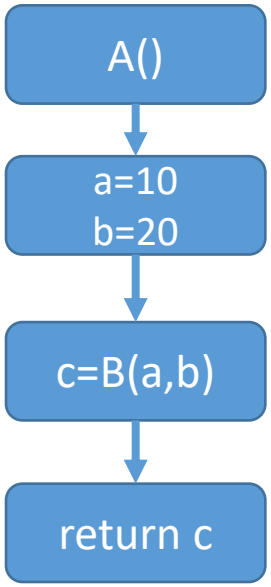
过程间分析

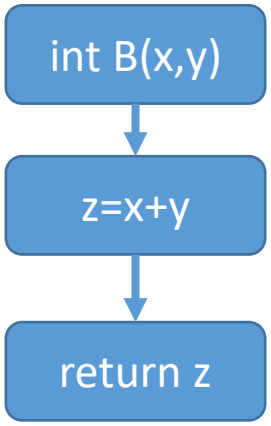
熊英飞
北京大学

过程间分析

- 过程内分析Intra-procedural Analysis
  - 只考虑过程内部语句，不考虑过程调用
  - 目前的所有分析都是过程内的
- 过程间分析Inter-procedural Analysis
  - 考虑过程调用的分析
  - 有时又称为全程序分析Whole Program Analysis
  - 对于C, C++等语言，有时又称为链接时分析Link-time Analysis

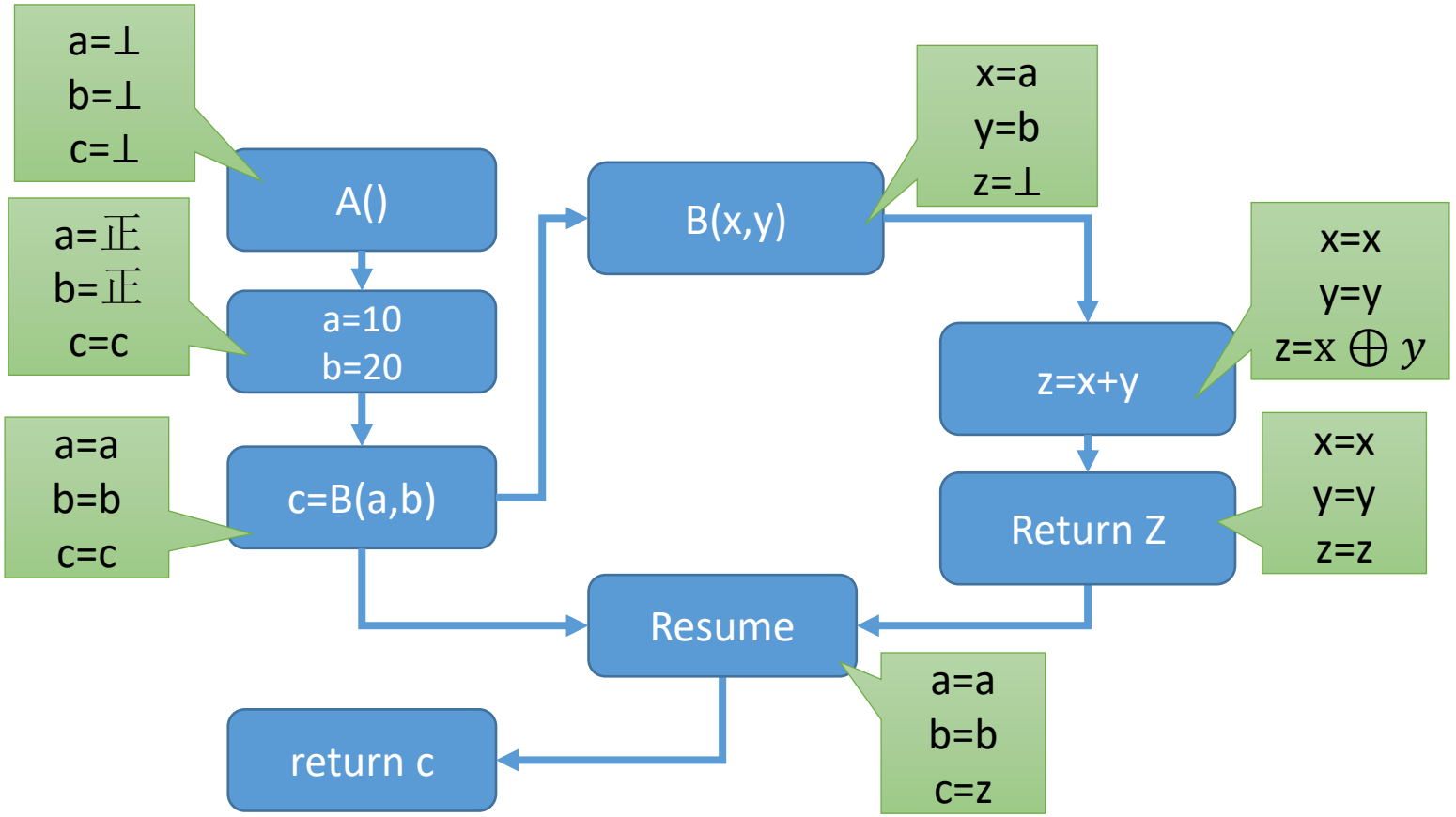
过程间分析-示例





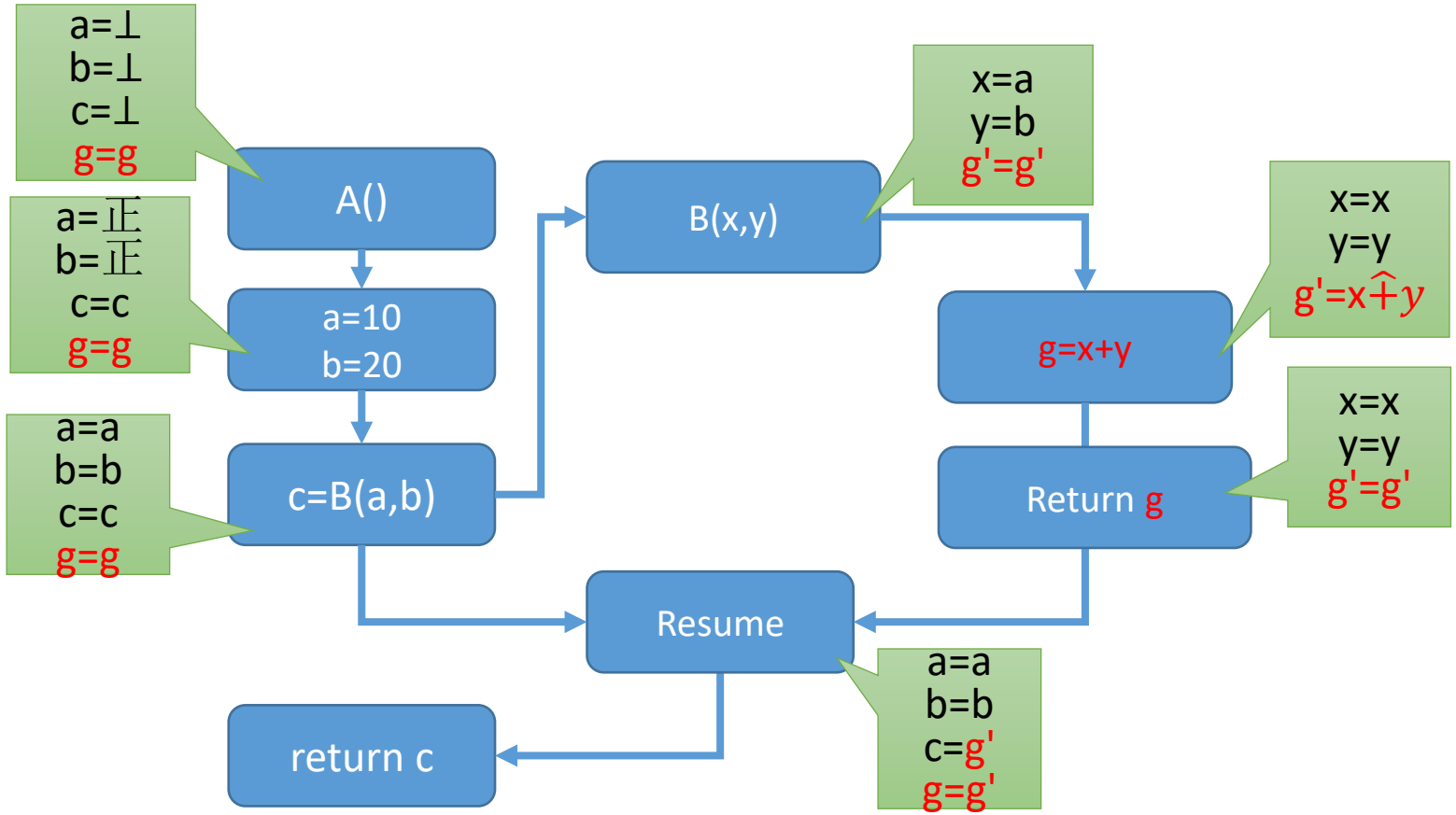
过程间分析-基本思路

- 对不同过程采用不同抽象域
- 在调用和返回的时候添加结点来转换信息



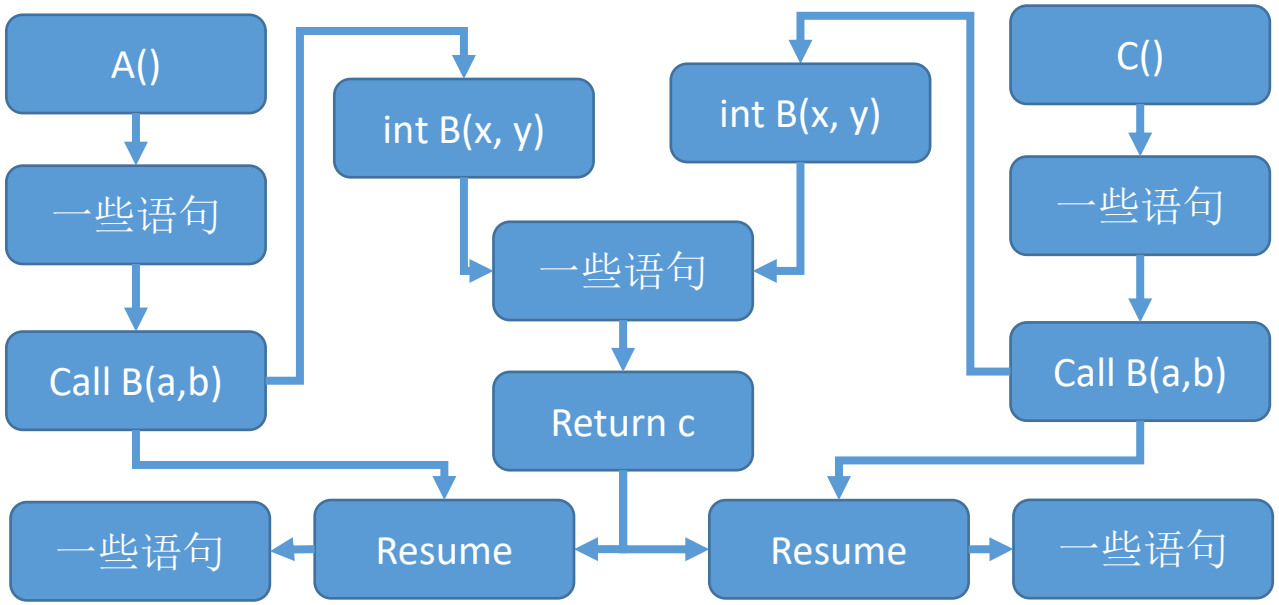
全局变量

- 全局变量需要加到所有使用的过程和这些过程的直接和间接调用者中
- 全局变量带来的开销是过程间静态分析必须考虑的问题



过程间分析-精度问题

- 上述分析方法是否有精度损失?



- 会考虑不可能路径上的执行轨迹
- 会将A()函数的分析结果引入C()函数，反之亦然

示例：常量分析

- 判断在某个程序点某个变量的值是否是常量

```
int id(int a) { return a; }
void main() {
  int x = id(100);
  int y = id(200);
}
```

- main执行结束时，x和y都应该是常量，但在super CFG中分析不出来这样的结果

精度损失对比

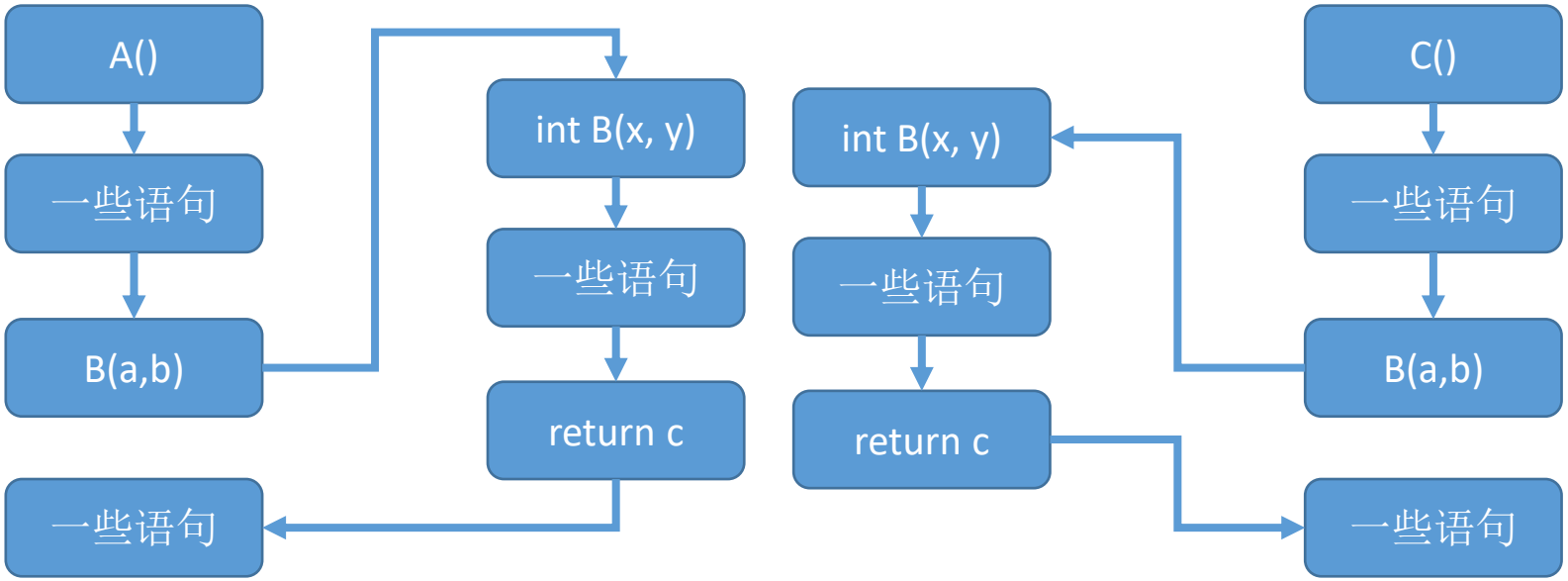
- 在过程内分析时，当条件压缩函数不够精确时，同样会导致不可能的路径上的执行踪迹
  - 为什么可达定值等分析完全忽略这个问题?
- 过程内分析不精确的条件：存在两个条件互斥
- 过程间分析不精确的条件：一个过程被调用两次
- 后者远比前者普遍
- 在面向对象或者函数语言中，过程调用非常频繁，该方法将导致非常不精确的结果

上下文敏感性
Context-sensitivity

- 上下文非敏感分析Context-insensitive analysis
  - 在过程调用的时候忽略调用的上下文
- 上下文敏感分析Context-sensitive analysis
  - 在过程调用的时候考虑调用的上下文

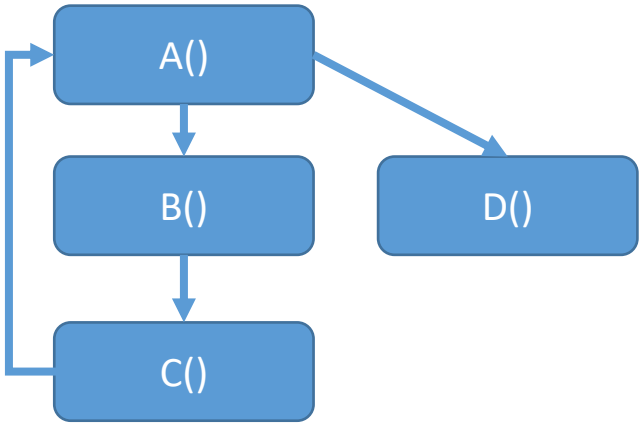
基于克隆的上下文敏感分析
Clone-based Context-Sensitive Analysis

- 给每一处调用创建原函数的一份复制



Call Graph调用图

- 结点表示过程，边表示调用关系



基于克隆的上下文敏感分析

- 练习1：给定下面的程序，请画出克隆之后的函数调用图
  - p() {return q()*q();}
  - q() {return r()+r();}
  - r() {return 100;}

基于克隆的上下文敏感分析

- 练习1：给定下面的程序，请画出克隆之后的函数调用图
  - p() {return q()*q();}
  - q() {return r()+r();}
  - r() {return 100;}

- 练习2：能否画出下面程序的克隆后的函数调用图?
  - p(int n) {return n*p(n-1);}

基于克隆的上下文敏感分析

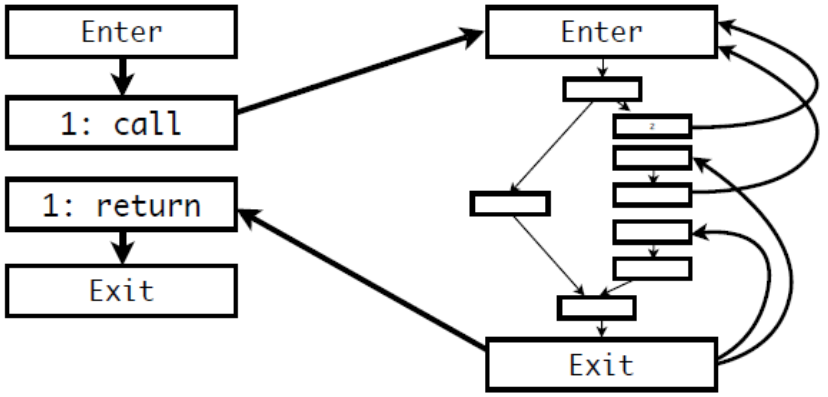
- 调用栈决定了不同的返回位置
- 但调用栈的可能长度是无限的
- 主要问题1：如果有深层次重复函数，就会出现指数爆炸
- 主要问题2：递归调用无法处理

基于克隆的上下文敏感分析

- 解决方案：
  - 对调用栈做抽象
  - 只使用最近k次调用（以及结束的调用不算）区分上下文
  - 如果最近k次调用的位置都相同，则不复制，否则复制

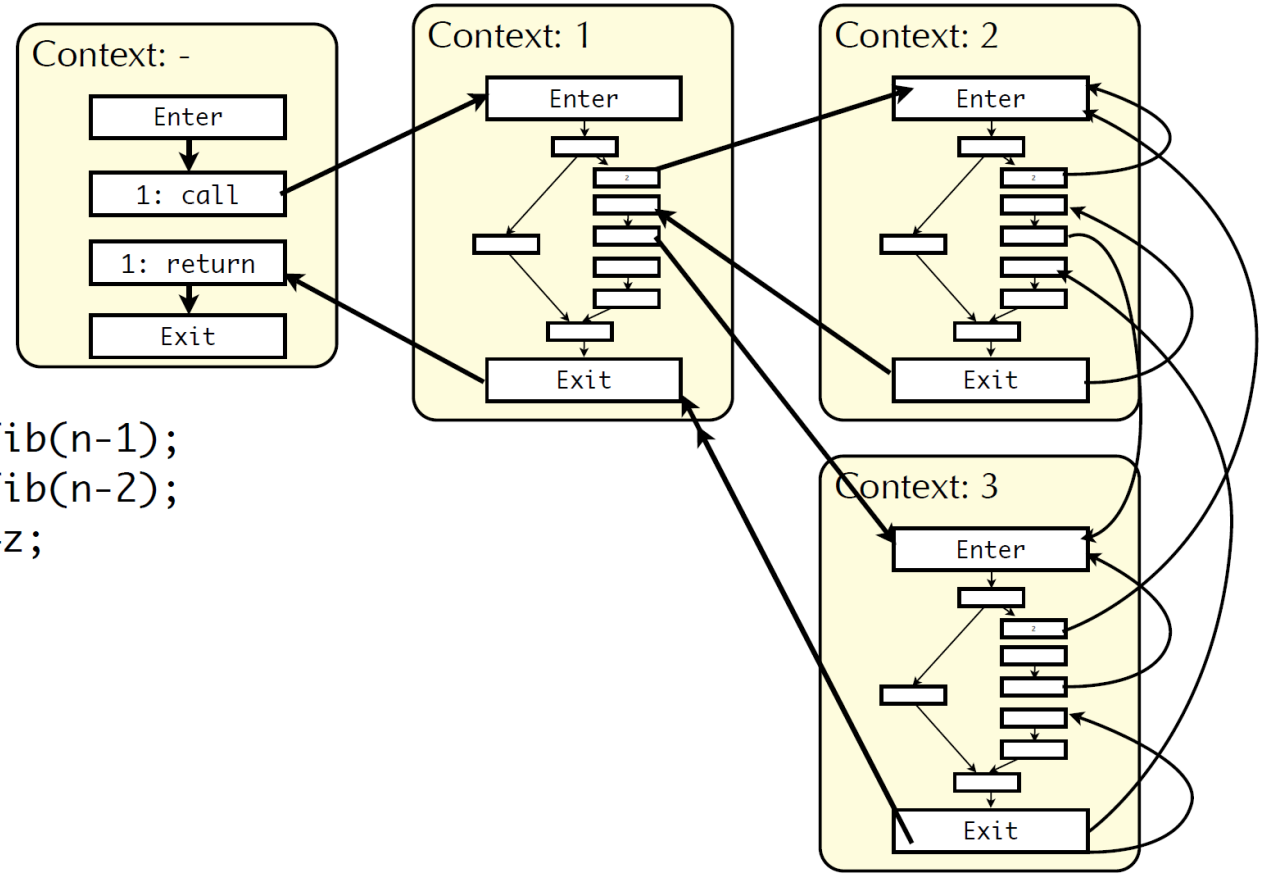
Fibonacci函数示例 --Context-Insensitive

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

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



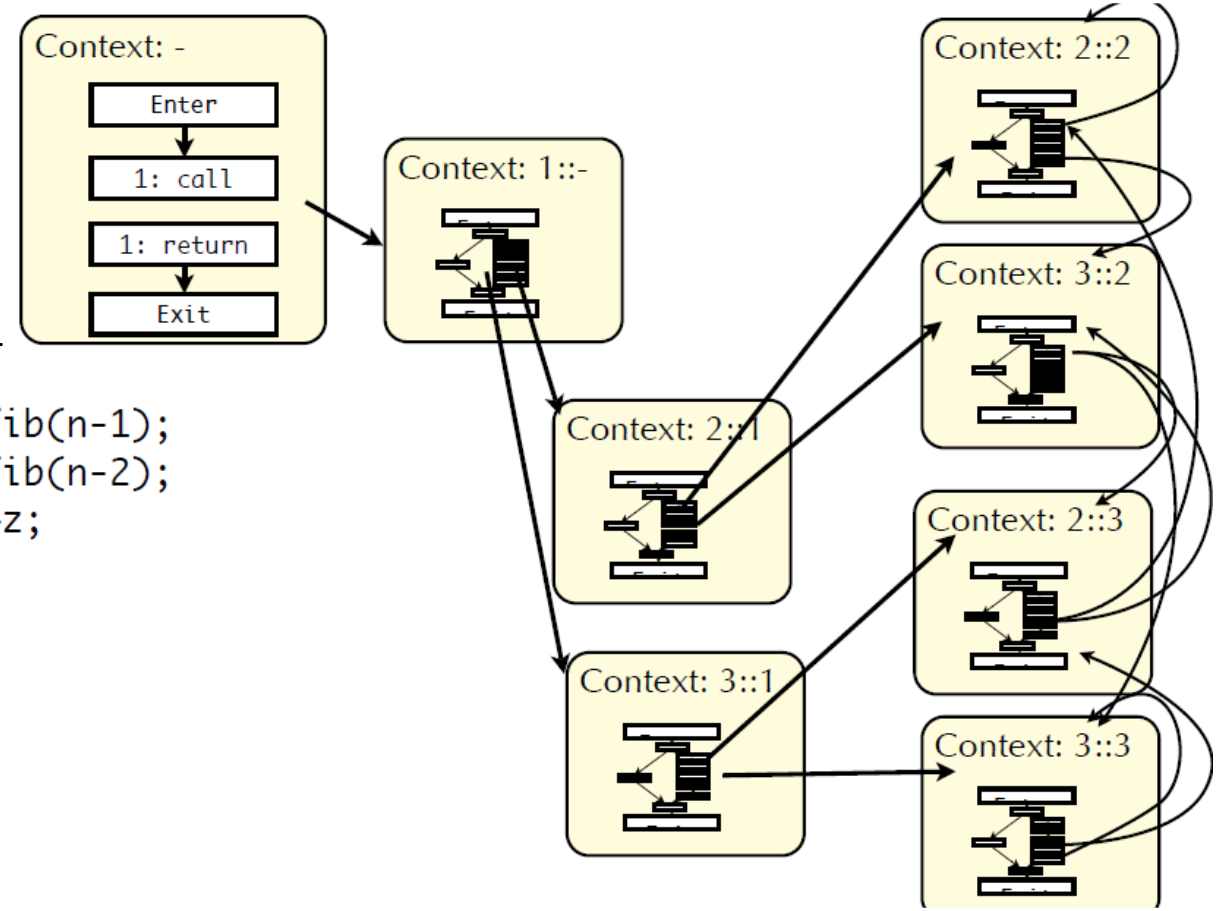
Fibonacci函数示例
--深度1的Clone-based analysis

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

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



Fibonacci函数示例
--深度2的Clone-based analysis

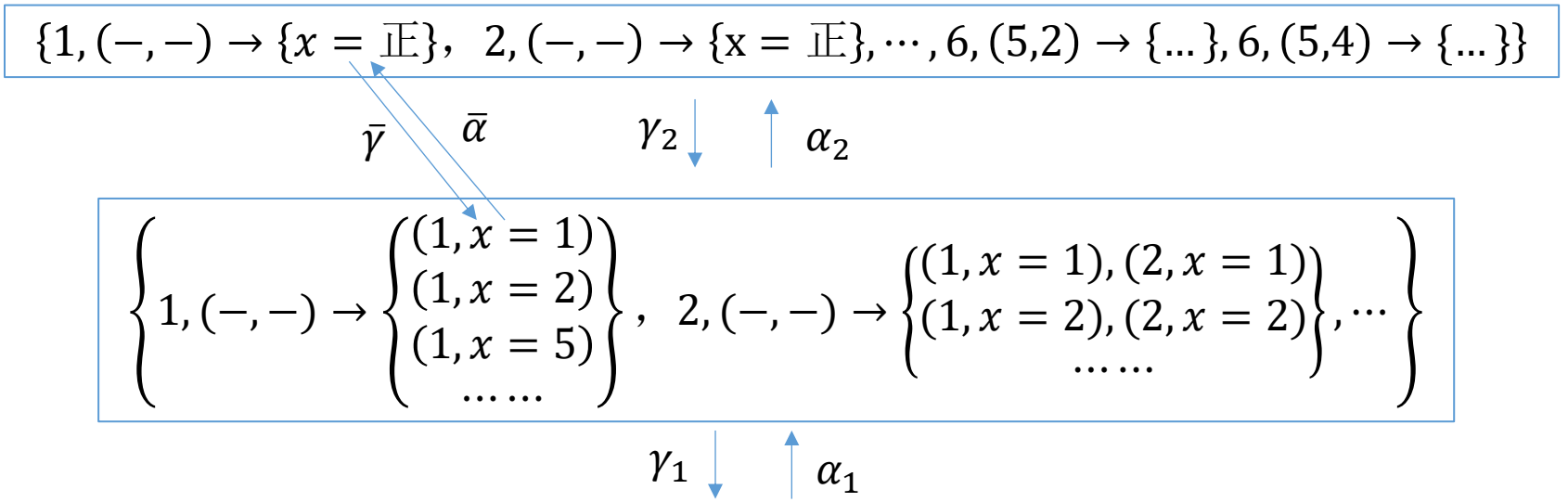
```
main() {
  1: fib(7);
}

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



其他可能的上下文

- 基于函数名字而不是调用位置的上下文
  - 在Fib的例子中，2::2和2::3都变成了fib::fib
  - 不如调用位置精确，但能减少复制量
- 基于对象类型的类型的上下文
  - 在OO语言中，对于x.p()的调用，根据x的不同类型区分上下文
  - 可以对OO函数重载进行一些更精细的分析
- 基于系统状态的上下文
  - 根据分析需要，对系统的当前状态进行分类
  - 当函数以不同状态调用时，对函数复制

从抽象解释的角度


{1, (−, −) → {x = 正}, 2, (−, −) → {x = 正}, ⋯, 6, (5,2) → {…}, 6, (5,4) → {…}}
γ̄
ᾱ
γ2
α2
{1, (−, −) → {(1, x = 1), (1, x = 2), (1, x = 5), ……}, 2, (−, −) → {(1, x = 1), (2, x = 1), (1, x = 2), (2, x = 2), ……}, ⋯}
γ1
α1
{(1, ( ), x = 1), (2, ( ), x = 1), (5, (2), x = 2), (6, (5,2), x = 2)
(1, ( ), x = 2), (2, ( ), x = 2), (5, (2), x = 3), (6, (5,2), x = 3)
(1, ( ), x = 5), (3, (3), x = 5), (4, ( ), x = 6), (5, (4), x = 15), (6, (5,4), x = 15)
……}

克隆背后的主要思想

- 原始数据流分析按结束的控制流节点对具体直接轨迹分类，每类对应一个抽象值
  - 标准控制流图
- 流非敏感分析不分类，总共只有一个抽象值
  - 所有节点合并成一个的控制流图
- 上下文敏感进一步用最近k次调用来分类
  - 每个节点基于上下文分裂成多个

克隆思想用于过程内分析

- 实例1：
  - 已知c1和c2的条件互斥，给定如下的代码
    - if(x > 0) x++; else x--; y=x;if (y > 0) y++; else y=-y;
  - 该代码上的路径敏感区间分析有何不精确？
    - 会得出y的范围是[0, ∞]，但精确值是[1, ∞]
    - 两条路径汇合时会丢失精度
  - 如何通过克隆手段来解决该不精确？
    - Contex: 前一个if走的是true还是false分支
    - 等价于把代码变换成
      if(x > 0)
      {x++; y=x; if (y>0) y++; else y=-y;}
      else
      {x++; y=x; if (y>0) y++; else y=-y;}
    - 由于y=x复制了两次，不会有汇合产生的精度损失

克隆思想用于过程内分析

- 实例2：对于循环，可以展开一定层数k，这样对于前k层会有比较精确的结果
  - 上下文：{第一次循环，第二次循环，…，第k次和更多次循环}

- 实例3：根据对系统状态的分类同样可以在语句级别而不是过程级别进行复制

内联Inline

- 另一种实现克隆的方法是内联
- 内联：把被调函数的代码嵌入到调用函数中，对参数进行改名替换
- 实际效果和克隆等效

精确的上下文敏感分析

- 精确的上下文敏感分析
  - =考虑最近任意多次调用的上下文
  - =对于任意分析中考虑的路径，路径中的调用边和返回边全部匹配（称为可行路径）

- 能否做到精确的上下文敏感分析?
  - 历史上提出多种不同的模型
    - CFL可达性分析
    - 函数摘要技术

CFL可达性分析

- 1995年由Reps等人提出
  - 理解上比较直观
  - 能够优化出高效算法
  - 能覆盖任意的具备分配性的数据流分析（但不能覆盖所有已有方法）
  - 基于该模型讨论清楚了若干可判定性问题

发明人：威斯康星大学
Thomas Reps教授
Susan Horwitz教授

Dyck-CFL

括号匹配的上下文无关语言

$S \rightarrow \{_1\ S\ \}_1$

$\quad | \ \{_2\ S\ \}_2$

$\quad | \ \ldots$

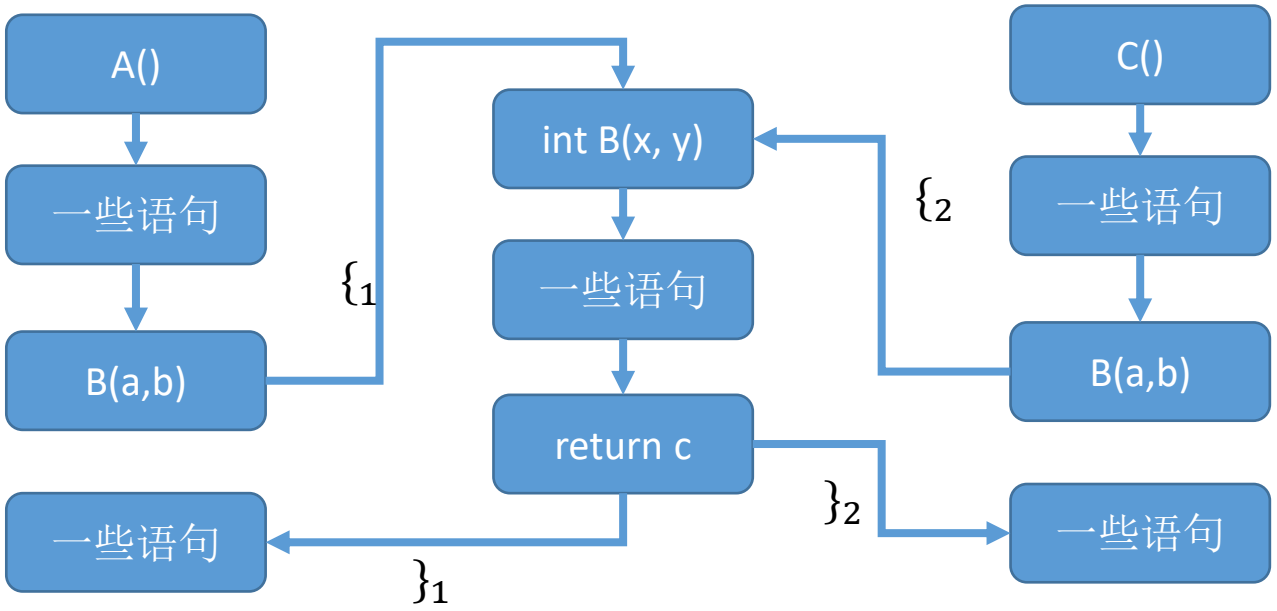
$\quad | \ SS$

$\quad | \ \epsilon$

注意区分上下文无关语言和上下文敏感性中的上下文

Dyck-CFL与上下文敏感性

- 给系统中的每一处调用分配唯一一对括号



- 给定一条路径，如果该路径上的符号组成Dyck-CFL的句子，则该路径是可行路径

复习：数据流分析的分配性 Distributivity

- 什么叫数据流分析的分配性?

- 一个数据流分析满足分配性，如果
  - $\forall v \in V, x, y \in S: f_v(x) \sqcup f_v(y) = f_v(x \sqcup y)$

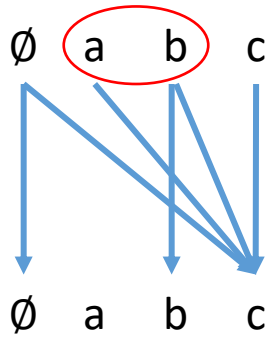
- 符号分析是否满足分配性?
- GEN/KILL标准型$f(S) = (S - KILL) \cup \text{GEN}$是否满足分配性?

数据流分析的分配性

- 推论
  - 给定任意满足分配性的数据流分析X，其entry结点的初值是I^X。
  - 令Y和Z为仅有初值不同的数据流分析，其中Y和Z的初值分别是I^Y、I^Z，且$I^X = I^Y \sqcup I^Z$。
  - 令OUT_v^X为通过X分析得出的v结点的值
  - 则：$\mathrm{OUT}_v^X = \mathrm{OUT}_v^Y \sqcup \mathrm{OUT}_v^Z$
- 证明
  - 在任意执行路径上，结论成立
  - 根据数据流分配性的性质，在所有执行路径上，结论成立

转换函数vs图可达性

- 对于可分配函数f(X)=(X-{a})∪{c}，其中X为{a,b,c}的子集
- 可以表示成图

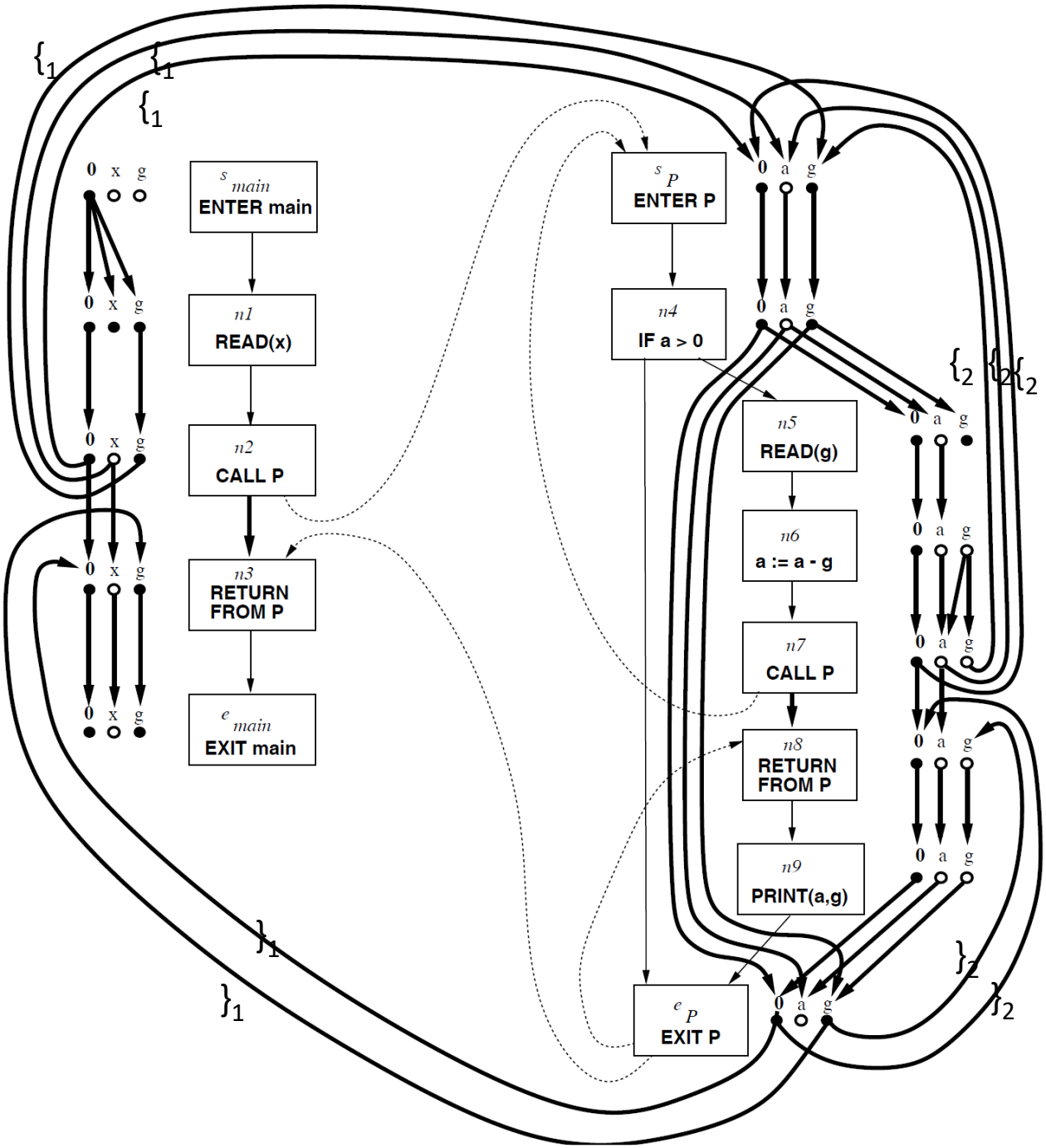


- 求解f({a,b})变成了从 {a,b}出发的可达性问题
- 结合上页推论我们可以把整个数据流分析转成可达性问题

分配性与图可达性

- 分配性是图可达性的充分条件
- 没有分配性可能也可以转成图
  - 考虑上节课作业的符号分析
- 分配性保证结果相等，没有分配性的时候，结果可能更精确，但计算量可能更大
  - 比如符号分析，一方面拆了之后的图包括抽象域中所有值，另一方面对于一些本来计算一个值的情况（如：踩）现在需要计算多个值求并（如：正、负）

例：未初始化变量分析


s main
ENTER main
n1
READ(x)
n2
CALL P
n3
RETURN FROM P
e main
EXIT main
s P
ENTER P
n4
IF a > 0
n5
READ(g)
n6
a := a - g
n7
CALL P
n8
RETURN FROM P
n9
PRINT(a,g)
e P
EXIT P
0 x g
0 a g

上下文无关语言可达性问题

- 给定一个图，其中每条边上有标签
- 给定一个用上下文无关文法描述的语言L
- 对于图中任意结点v1、v2，确定是否存在从v1到v2的路径p，使得该路径上的标签组成了L中的句子。

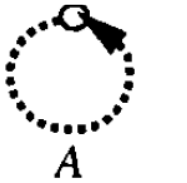
计算方法

- 把原文法改写为右边只有最多两个符号的形式

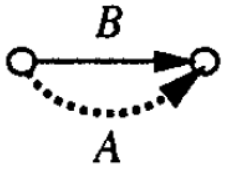
$$
\begin{aligned}
S \rightarrow\ & \{_1 E_1 && E_1 \rightarrow S\}_1 \\
\mid\ & \{_2 E_2 && E_2 \rightarrow S\}_2 \\
\mid\ & \ldots \\
\mid\ & \epsilon
\end{aligned}
$$

计算方法

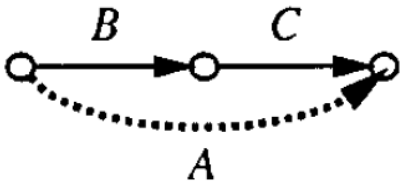
- 按如下三种模式不断添加边，直到没有边需要添加



(a) $A \rightarrow \varepsilon$



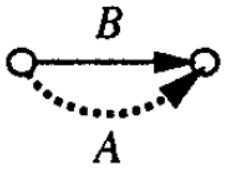
(b) $A \rightarrow B$



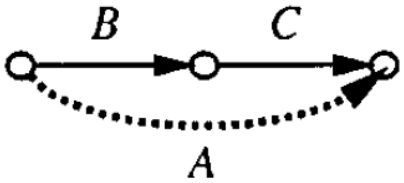
(c) $A \rightarrow B\ C$

例子

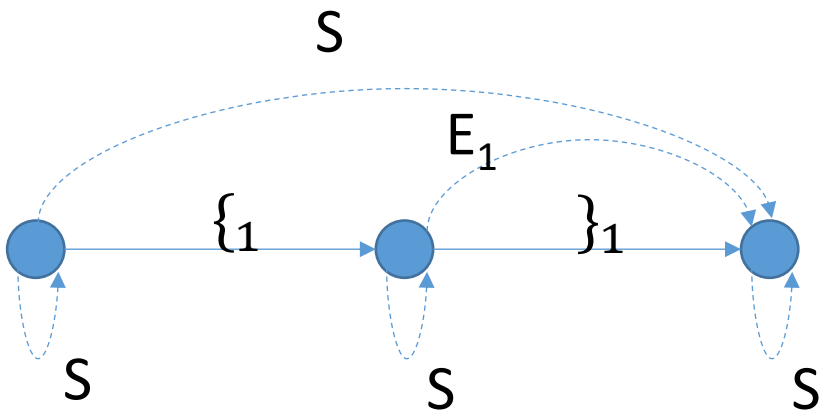
(a) $A \rightarrow \varepsilon$



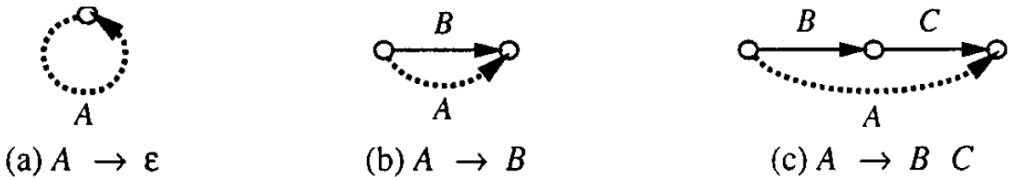
(b) $A \rightarrow B$



(c) $A \rightarrow B\ C$



伪码



For(each node) {
  根据规则(a)加边
}
ToVisit ← 所有边
While(ToVisit.size > 0) {
  从ToVisit中取出任意边
  根据规则(b)加边
  查看前后结点的边的组合，根据规则(c)加边
  以上两步新加边加入ToVisit
}

安全性

- 给定任意控制流图上的可行路径，该路径上算出的结果一定包含在CFL-Reachability算出的结果中。
  - 证明：
    - 根据之前的分析，对于单条路径，CFL-Reachability算出的可达性一定等价于传递函数算出的值
    - 由于该路径是可行路径，所以起结果一定仍然可达

精确性

- 给定任意n，以前n次调用位置作为上下文的基于克隆的分析所产生的结果一定包括CFL-Reachability分析所产生的结果
  - 证明：
    - CFL-Reachability上的任意一条可达路径所产生的结果一定包含在克隆分析中
    - CFL-Reachability分析的结果就是所有可达路径的结果的合并

交集的情况

- 当数据流分析对应的合并操作为∩时，单位元素就变成了全集和全集中去掉一个元素对应的所有集合。用同样的方式可以组合得到所有集合。

术语：IFDS

- 可以用CFL可达性解决的问题又称为IFDS问题
  - Interprocedural, finite, distributive, subset problems
- 后来通常称CFL可达性解决数据流问题的方法称为IFDS框架或者IFDS方法
  - 一般指采用下节课介绍的优化之后的算法

精确的上下文敏感分析意义有多大?

- 学术界有过多次试验，结论不完全一致
- 总体观点
  - 在面向对象程序中，上下文敏感分析比上下文非敏感分析精确很多
  - 精确的上下文敏感分析比不精确的上下文敏感分析通常更精确一些
  - 精确的上下文敏感分析所额外增加的开销是可接受的

作业1

- 设计一个过程间的Reaching Definition分析
  - 给出半格定义和变换函数定义，特别是call语句和return语句的变换函数
  - 参数传递不认为是定值。比如，下面的程序中，标号2位置的c是由标号1的语句定值的而不是标号3的语句定值的

```
void main() {
  int a = 100; //1
  b(a) //3
}
void b(int c) {
  print(c);//2
}
```

  - 假设过程调用时不能传表达式，只能传变量，即
    - f(a+b)不合法，f(a)合法

作业1

- 根据前一页定义的Reaching Definition分析，采用基于克隆的方法分析下面的程序，k至少应该设置成多少分析结果才能和精确的上下文敏感分析等价?
  - int g;
    void main(){
      int a = 10;
      g=a+1;
      h(a); h(a);
    }
    void h(int a) {
      x(a); g++; x(a);
    }
    void x(int a) {
      output(a);
    }

作业2

- 对右边这个程序做符号分析，考虑条件压缩函数，画出CFL-reachability的图。
  - 要求：根据输入值x的符号，在这个程序上能分析出返回值完全精确的符号结果
- 假设int是无界整数，分析是否存在一个足够大的k，使得基于克隆的方法返回精确的结果。如果存在，请给出这样的k；如果不存在，请给出简要证明

```
int f(int x){
  int ret;
  if(x < 0){
    x = x+ 1;
    if(x == 0) ret = 2024;
    else ret = f(x);
  }
  else if(x > 0) {
    x = x - 1;
    if(x == 0) ret = -2024;
    else ret=f(x);
  }
  else ret = 0;
  return ret;
}
```

参考资料

- Two Approaches to Interprocedural Data Flow Analysis
  - Micha Sharir and Amir Pnueli
  - New York Univerisity Technical Report, 1978
- Precise Interprocedural Dataflow Analysis via Graph Reachability
  - Thomas W. Reps, Susan Horwitz, Shmuel Sagiv
  - POPL 1995