程序分析-原理

UCSB CS290C课程笔记

0 缘起

程序分析最传统、可能也是最重要的应用应该在编译优化。有关编译的材料，网上相当多，其中最权威的包括龙书、虎书、鲸书和Rice出的那本Optimizing Compilers for Modern Architectures。龙书和虎书覆盖编译了很多基础方面，部分章节也讨论优化；而后两者专门讲优化，程序分析的大部分内容其实都涉及到了。从编译的角度，这份材料可以作为补充参考，此外这门课在两个方面还有点特色：(1) 对程序分析的体系梳理比较到位，内容几乎都很直观，便于理解。可以看完这个材料再去看那几本书，很多内容会更清晰；(2) 这门课的授课老师Ben Hardekopf在指针分析领域有几个重要的工作，对指针分析介绍得比较全面深入。

这份材料主要的对象是那些对程序分析感兴趣但还没有找到合适材料的同仁。说实话，能把这个领域的主要内容压缩在几十页的篇幅自己也有点惊讶，然而事实就是如此。有些内容如果感兴趣希望扩展阅读，后面也列出的重要的参考文献。程序分析是一个相当大的领域，发表的文章可谓浩如烟海，通过浓缩的方式把精要梳理出来，把重要的文献整理出来，相信能帮助节省很多自己摸索的时间。通过这份材料或许能给你的工具箱添加一个新的探索问题的工具。

必须说明这份材料仅仅只是笔记，里面的贴图都来自课程的ppt(网上可以下载)。由于水平有限可能有错误之处，建议结合ppt对照使用。由于这个领域很大，相信有很多内容没有涉及到，因此这个文档没有写完，只是由于自身能力局限，只能写到这里。恳请知道更多内容且有时间的同仁继续补充，最终给大家提供一份这个领域比较全面的资料。

整个材料的提纲如下：

1 导言

1.1 缘起及基本发展

程序分析是一种自动推导程序运行行为的技术，起源于早期以Fortran为代表的高级语言的编译优化需求。计算机的早期发展阶段主要采用接近机器的汇编语言编程。当高级语言最初被引入时，由于当时编译器无法产生高效的代码，高级语言编程的想法曾经受到广泛的质疑，然而以程序分析为基础的编译优化技术随后得到长足发展，高级语言在大多数应用领域取代汇编语言成为人们的常识。编译优化只是程序分析的一种重要应用，程序分析基本思想现已应用到程序安全验证、软件工程、电路综合、操作系统(如微软Singularity项目)以及GPU编程等应用领域。表1.1列举了一些程序分析的应用实例。

表1.1 程序分析举例

分析方法	应用领域	分析目标
常量传播	程序优化	分析所有为常量的变量，化简表达式和程序流图
死代码删除	程序优化	分析程序中不可能执行到的代码路径。一般程序员并非有意写死代码，但通过编译优化或自动程序变换修改了程序的数据流或控制流，从而可能出现死代码。
缓冲区溢出	程序验证	比如数组越界检测
变量使用检查	程序验证	如检测对未初始化变量的使用

1.2 Lingo语言

为了将后续的程序分析技术落实，这里引入一个基本的语言，定义如图1.1所示。首先符号集合的定义，n表示整数集中的元素，b是布尔值(true/false)，x表示程序变量。基本表达式操作符包括算术运算符(+,-,*,/)、关系操作符(<,<=,=,!=)和逻辑运算操作符(&&和||)。表达式包括整数或布尔常量、变量或者各种运算表达式。基本语句包括输入、赋值、NOP(Skip)、循环、选择，此外还定义了语句的组合方法(c;c)，最后一个程序(Prog)就是一个语句(Cmd)。

相对于实际的语言如C，Lingo要简单得多。注意关系操作符中没有>，另外只有两种数据类型(int和bool)，且不支持显式或隐式类型转换。变量定义默认初始化为0/false，除法假设有理想精度。后续为便于讨论各种分析技术还会增加指针和函数调用两种语言特性。

Syntax

$$n \in \mathbb{Z} \qquad b \in \mathbb{B} \qquad x \in Variable$$

$$\oplus \in Op ::= + \mid - \mid * \mid / \mid < \mid <= \mid = \mid != \mid \&\& \mid \mid\mid$$
$$e \in Exp ::= n \mid b \mid x \mid e \oplus e$$
$$c \in Cmd ::= c\ ;\ c \mid x := e \mid \textbf{skip} \mid \textbf{while}\ (e)\ \textbf{do}\ \{c\}$$
$$\mid \textbf{if}\ (e)\ \textbf{then}\ \{c\}\ \textbf{else}\ \{c\} \mid \textbf{input}\ x$$

$$p \in Prog ::= c$$

图1.1 Lingo语言定义

下图1.2给出了一个Lingo的例子程序，直观上很快就给出常量传播(程序中哪些位置的哪些变量肯定是常量?)和死代码检查(哪些语句肯定不会被执行?)的分析结果。可见程序分析来源于程序员理解程序时的经验直觉，经验和直觉只有经过升华才会成为具有一定普适性的指导原则，从后续要陆续展开的各种分析框架可以感受到这一点。写小说的强调要源于生活并高于生活，人类其他学科依然，只是演绎的基本思维元素不同。

Program A

```
[1]x := 5;
[2]if (y < x) then { [3]y := x + 1 }
              else { [4]y := x - 1 } ;
[5]z := y
```

图1.2 Lingo程序举例

1.3 程序语义模型

既然程序分析是提取和挖掘程序行为，因此首要问题是如何严格界定程序行为。有三大类程序语义模型：基于符号的(denotational)、基于抽象操作(operational)和公理化(Axiomatic)证明。符号模型和公理化证明不知为何物，抽象操作模型则相对直观，基本思想是给定一个程序，设想有一个抽象的虚拟机来执行之，依据不同的输入理论上可能得

到无穷个执行轨迹(trace)，每条轨迹对应程序一种可能的运行状态，那么所有执行轨迹的状态总和就构成程序所有可能的行为。

从程序抽象执行轨迹出发来理解程序行为还被用到了其他地方。这里考察的是同一个程序因输入不同可能出现的无穷多执行轨迹，还有一个角度是考察在一台机器(SMP/Multi-core)上运行的多线程程序各个线程间指令的可能交错(interleave)情况，在线程间存在数据竞争的情况下，不同的interleave可能得到不同的执行结果，执行结果对或错的标准由内存模型来界定。尽管解决的问题不同，二者背后都利用了可以设想抽象虚拟机来枚举程序行为这一直觉。

有了虚拟机，那么给定输入，在任意程序点机器的状态就确定了。这里程序点直观上可以理解为程序中语句的位置，如图1.2示例代码中红色数字标识的位置，也就是程序计数器(PC)指向的地址。虚拟机的状态可以用给定PC位置当前所有变量和值的映射关系来描述，如图1.3的定义所示：

Definition (Abstract Machine Configuration)

A *configuration* is a tuple $\langle \mathbf{pc}, \sigma \rangle$

- $\mathbf{pc} \in$ **program counter**: the current statement being executed
- $\sigma \in$ **store**: mapping variables to values

图1.3 给定程序点虚拟机状态

给定程序输入，抽象虚拟机模拟执行每条指令就能得到每个程序点的机器状态。图1.4给出了图1.2所示的程序执行到程序点3并推进到5的机器状态：

Example

Configuration: $\langle \mathbf{pc} = {}^{3}\mathtt{y} := \mathtt{x+1}, \sigma = [x \mapsto 5, y \mapsto 0] \rangle$
or just: $\langle 3, [x \mapsto 5, y \mapsto 0] \rangle$

Evaluation: $[\![\mathtt{y} := \mathtt{x+1}]\!]\sigma \rightarrow \langle \mathbf{pc}' = {}^{5}\mathtt{z} := \mathtt{y}, \sigma = [x \mapsto 5, y \mapsto 6] \rangle$

Thus: $\langle 3, [x \mapsto 5, y \mapsto 0] \rangle \rightarrow \langle 5, [x \mapsto 5, y \mapsto 6] \rangle$

图1.4 机器状态求解过程示例

如果已知虚拟机在每个程序点的状态，那么每步求解过程合起来就构成了程序的执行轨迹。由这种以虚拟机为基础的操作语义出发，给定一个程序理论上可以分析出所有可能的合理(valid)执行轨迹，从而得到程序所有可能发生(may possibly do)或一定发生(must always do)的行为集合。

1.4 程序分析评价指标

对于很多应用场景，程序分析依据一组约束分析用户关注的局部程序行为，但静态程序分析结果经常是不可判定的，如图1.5所示：

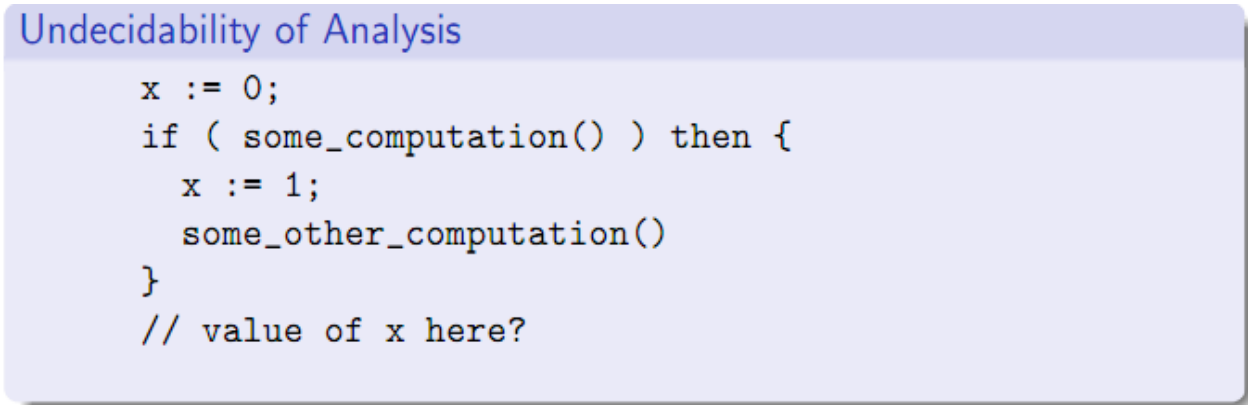

图1.5 程序分析结果的不可判定

图1.5中if语句之后变量x的值是未知，因为有多条路径都可能会修改，在这种情况下程序分析时只能一定程度去近似程序的实际行为，具体有三种操作方式：

A. Sound analysis，对变量的可能取值进行保守假设(over approximation)，如指出变量x取值范围是{0,1}，尽管实际执行结果只会是0。这种情况不会出现误杀，讲究尽可能提高分析的取值范围精度；

B. Complete analysis，和sound analysis相对，力求精确分析(under approximation)，宁信其无，不信其可能有，对图1.5的程序，分析结果变量x的取值范围是{}。这种情况会有误杀情况。

C. 最后一种是不提供正确性保证，对图1.5的程序，分析结果可能是变量x为0，而实际执行结果x可能为1。

后续内容只关注sound analysis。由于这类分析关注某个变量的可能取值范围，当精确分析不可能时，可以对变量值域进行一定抽象分类来分析其取值属性，如{Zero, Non-Zero, Maybe-Zero}、{Even, Odd, Either}等等。

2 控制流分析

2.1 控制流图

控制流图是对程序中分支跳转关系的抽象，描述程序所有可能执行路径，定义如下：

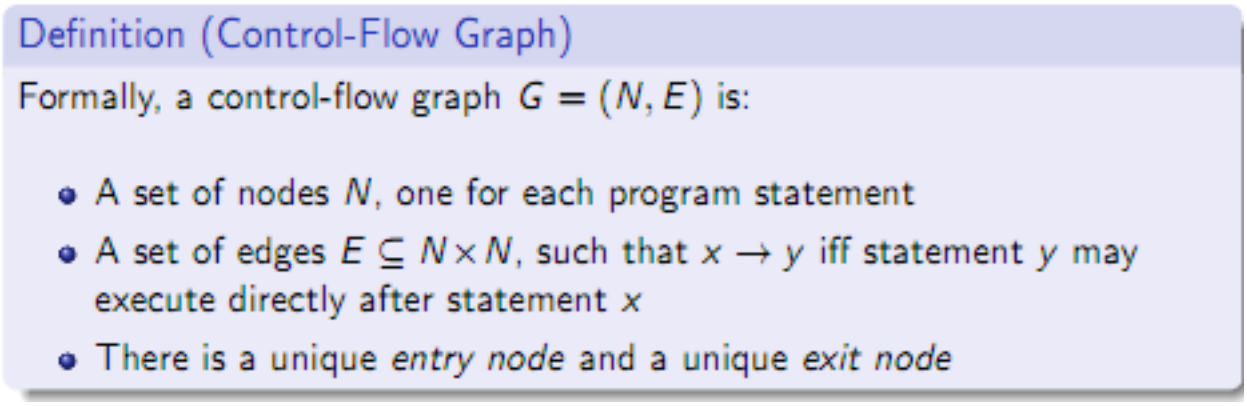

图2.1控制流图

控制流图的节点是语句集合，从A到B的边表示语句A执行完后可能直接执行语句B，整个控制流图必须有唯一的入口和出口。这是以每个语句为节点的控制流图，实际操作的控制流图一般以基本块(basic block)为基本节点，每个基本块包括若干条语句，基本块本身有唯一的入口和出口。

2.2 Dominator和Post-dominator

在控制流图中，Dominator描述节点之间在实际执行轨迹中的顺序关系。如果A是B的Dominator，那么意味着从程序入口执行到B的任意路径则一定经过A，称A Dom B，定义如下：

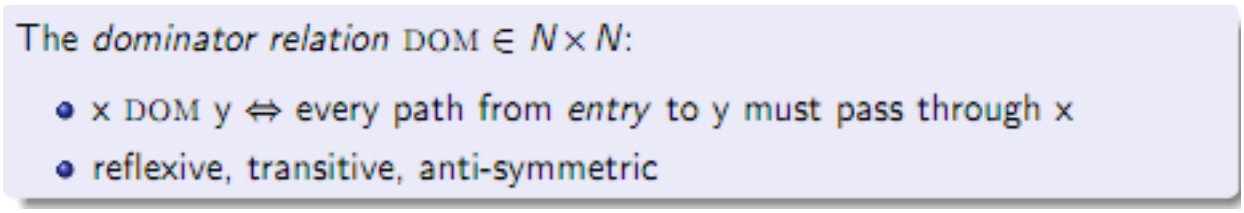

图2.2 Dominance关系定义

Dominance关系具有自反(x Dom x)、传递(x Dom y 且 y Dom z，有x Dom z)和反对称(x Dom y不意味着y Dom x)等性质。如果x Dom y且x!=y，那么称x是y的strict dominator。进一步的，如果x是y的strict dominator(记为Dom!)，并且对任意除x以外的y的任意dominator w，有w Dom x，那么称s是y的immediate dominator(记为IDom)，直观上，immediate dominator就是dominance关系链上最近的一个。Post-Dominator和Dominator是对偶的，如果x PDom y，那么任意从y到程序出口的路径都必须经过x。

考察dominator关系对一些程序分析很有价值。如检测对变量的初始化语句是否出现在所有对该变量的使用之前，或者在多线程程序中检测是否每一个lock都对应unlock。

根据Dominator关系的上述定义，任意一个节点都有唯一immediate dominator。如果把控制流图中所有其余的边删掉，只保留immediate dominator边，就得到一颗dominator树。

给定控制流图，只要求得其dominator树，就能得到节点间的所有dominator关系。下图2.3给出了一个Lingo程序以及对应的dominator树：

Program

```
[1]if (x < 0) then {
   [2]while (y > 0) do { [3]y := y-1 ; [4]x := z+1 }
} else { [5]x := y }
```

Dominator Tree

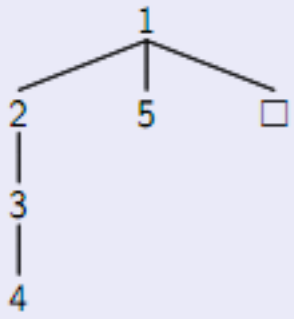

图2.3 dominator树举例

接下来的问题，给定控制流图，如何快速找到所有dominator关系，图2.4是一个基本迭代算法：

Dominator algorithm

$\mathrm{DOM}(entry) = \{entry\}$
for $n \in N \backslash \{entry\}$ **do** $\mathrm{DOM}(n) = N$

repeat
  for $n \in N \backslash \{entry\}$ **do**
    $\mathrm{DOM}(n) = \{n\} \cup (\bigcap_{p \in pred(n)} \mathrm{DOM}(p))$
  until solution doesn't change

图2.4 dominator关系求解算法

首先初始化，entry节点的dominator集合初始化为一个元素(entry自身)，所有其他节点的集合初始化为整个节点集。接下来进入迭代过程，每次迭代考察每个节点，将该节点所有前驱的集合求交集，整个迭代过程直到解不变为止(即不再有新的Dom关系添加进来)。这个算法不关心不同节点的计算顺序，但事实上由于节点间的依赖关系，合理计算顺序对整个dominator关系求解的性能至关重要。

考虑图2.5的深度优先遍历过程：

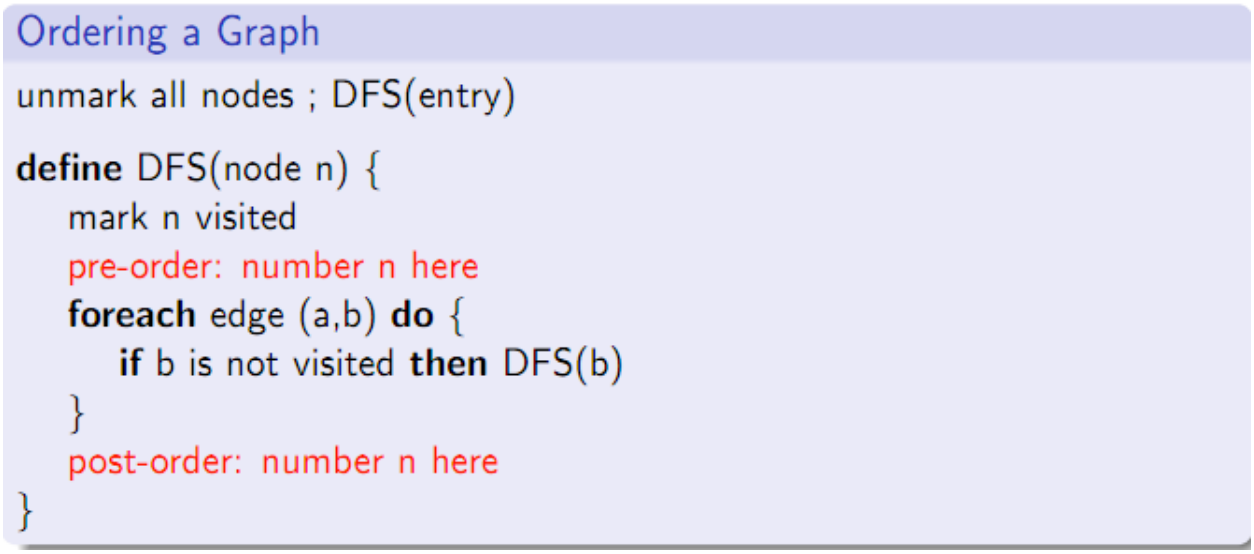

图2.5 DFS过程中的先序和后序编号

遍历过程中对节点升序编号，可见有两种基本的方式：先序(pre-order)和后序(post-order)。直观上，分析dominator关系应该采用先序，因为前驱节点应该尽可能先处理，但对于有环(循环)的控制流图仍可能需要多次迭代。直接采用后序违反直观，对任何节点，前驱节点处理之前无法完成处理该节点。但如果将后序倒过来(reverse post-order)，就能确保任意节点在其后继节点之前处理。reverse post-order不同于pre-order，更多的分析可以参考文献[1]。

Dominance Tree的高效计算方法如图2.6所示。算法基本框架和图2.4一样，需要留意几点细节。首先初始化时，对入口节点外的节点不是初始化为节点全集而是初始化为空(undefined)。进入迭代过程每轮循环需要遍历所有节点更新dominance关系，对所有节点(入口节点无需处理)采用reverse postorder来顺序处理。对选定的待处理节点b，首先选择第一个处理过(processed)的前驱，所谓处理过就是doms[b]不再是Undefined状态的节点，这一点很重要，否则有环时算法可能终止不了,接下来就是遍历所有处理过的前驱寻找b的immediate dominator，这一步通过intersect(b1,b2)实现。

intersect()过程中的变量都表示后序(post order)的节点编号，目标是找到b1和b2最近的公共前驱节点的编号，内层两个while循环按照reverse postorder的顺序找到reverse postorder编号小 (即postorder编号大) 的公共节点。图2.6的算法给每个节点找到其唯一的immediate dominator，所有这些信息合起来构成一颗dominance tree。

图2.4和2.6的算法都是从入口节点(entry node)出发构造dominator关系集合，一组节点的信息被分析出来的前提是从入口可达，因此这两个算法都分析不到死代码(从入口节点无论什么路径都无法到达)内的dominance关系。这点关系不大，因为人们一般只关注死代码的位置，对其内部结构兴趣不大。

```
for all nodes, b /* initialize the dominators array */
    doms[b] ← Undefined
doms[start_node] ← start_node
Changed ← true
while (Changed)
    Changed ← false
    for all nodes, b, in reverse postorder (except start_node)
        new_idom ← first (processed) predecessor of b /* (pick one) */
        for all other predecessors, p, of b
            if doms[p] ≠ Undefined /* i.e., if doms[p] already calculated */
                new_idom ← intersect(p, new_idom)
        if doms[b] ≠ new_idom
            doms[b] ← new_idom
            Changed ← true

function intersect(b1, b2) returns node
    finger1 ← b1
    finger2 ← b2
    while (finger1 ≠ finger2)
        while (finger1 < finger2)
            finger1 = doms[finger1]
        while (finger2 < finger1)
            finger2 = doms[finger2]
    return finger1
```

图2.6 Dominance Tree算法

2.3 控制依赖

控制依赖直观上很好理解，但在控制流图的准确定义存在需要注意的细节，定义如图2.7所示。

Definition (Control Dependence)

y is control-dependent on x ⇔

- ∃ path P from x to y
- $\forall n \in P \,.\, n \neq x \Rightarrow y \text{ PDOM } n$
- $\neg(y \text{ PDOM! } x)$

图2.7控制依赖的定义

这就是说节点y和x之间存在控制依赖需要满足三个条件。第一个条件即存在一条x到y的执行路径P；第二个条件进一步要求x到y之间没有其他控制语句，换言之对路径P上的任意节点n，n到出口的任意路径上一定经过y，也就是说y是n的post dominator。第三个条件要求y一定不是x的strict post dominator，直观上就是说在x和y之间一定有一条流程控制语句，进而使得除了从x到y之外一定还会有跳转到其他地方的路径。例如图2.8所示的程序中，语句5控制依赖于语句4，但语句6和语句4之间没有直接的控制依赖关系，因为4和6直接的语句5是一条控制语句。但另一方面，语句6,7,8都控制依赖于语句5，语句7 PDOM! 语句6，语句8 PDOM! 语句7。

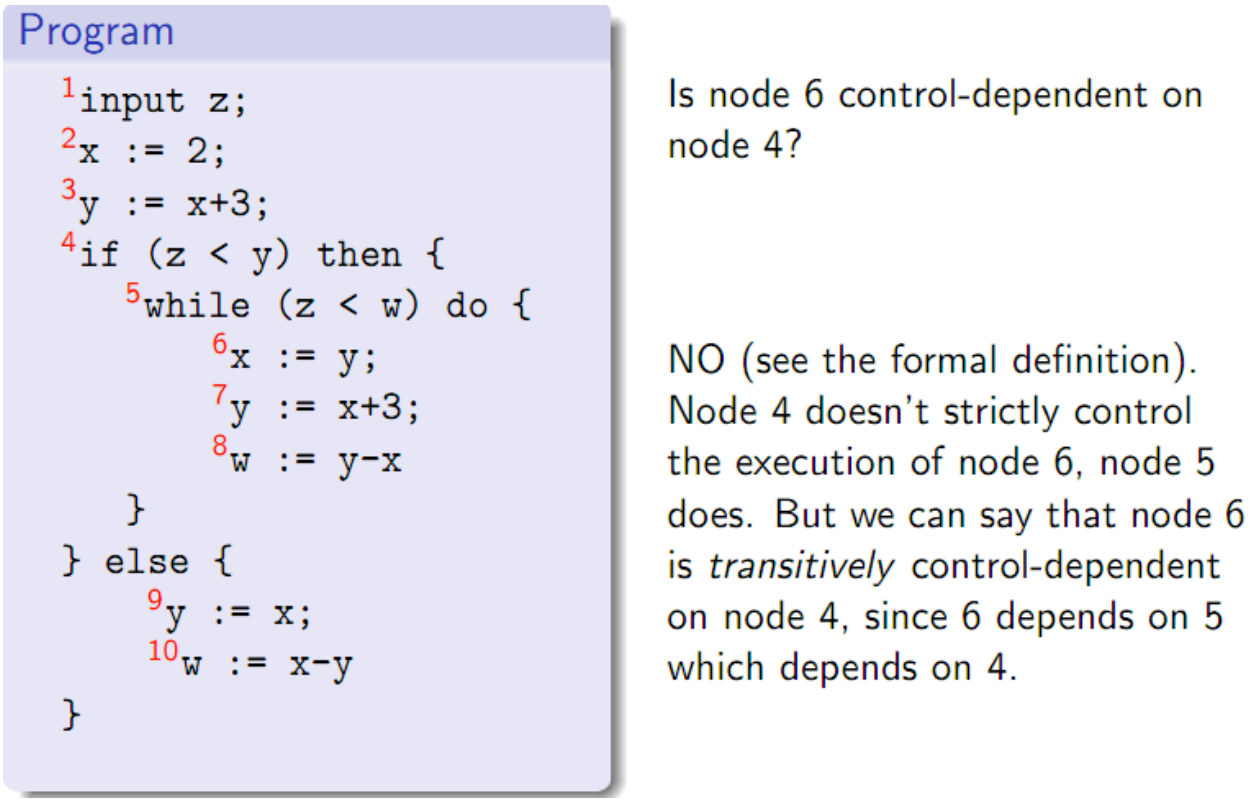

图2.8 控制依赖举例

2.4 循环和强连通子图

控制流分析的另一个应用是检测循环。对控制流图进行深度优先遍历(DFS)能得到一个生成树，生成树中边叫forward edge。另外还有两种边：cross edge和backward edge。backward edge表示控制流图中存在循环。问题是如何区分cross edge和backward edge，下面以图2.9中的程序为例来说明。

图中每条语句用数字标出，显然从语句8到语句3的边回到循环条件，构成回边(backward edge)。语句6到8和语句7到8分别有一条边，这两条边只有一条可能出现在DFS遍历的生

成树中，剩下的一条就是交叉边(cross edge)。假设有一条从A到B的边，判断其是交叉边还是回边的方法是：如果在生成树中可以从B到达A，那么该边是回边；否则是交叉边。

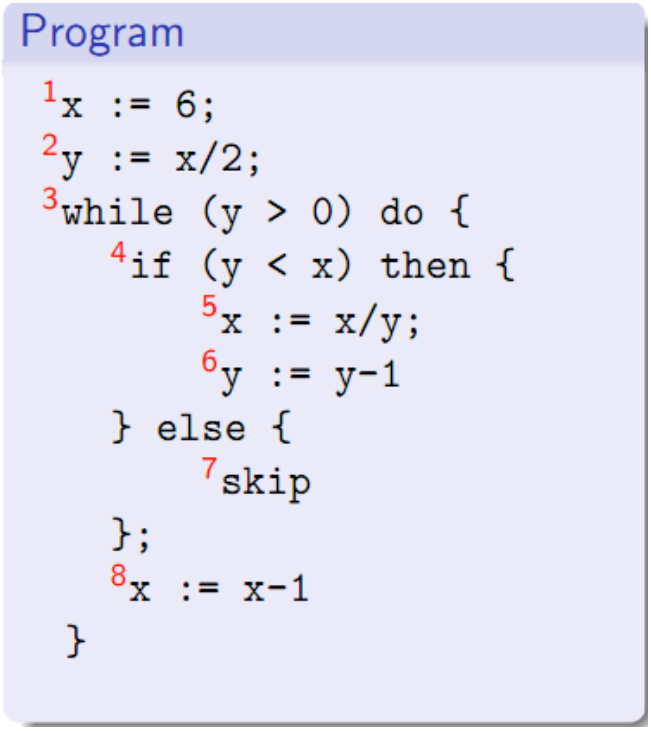

图2.9 循环举例

从另一个角度，控制流图中的循环本质上构成了图中的一个强连通子图(Strong connected component)。检测图中强连通子图的标准算法是Tarjan算法，如图2.10所示。Tarjan算法只需要对图进行一次深度优先遍历即可完成，需要O(n+m)的复杂度，其中n是边数目，m是顶点数目(算法中有对每个顶点的栈操作)。preorder按照优先遍历的顺序对节点编号，n.order表示节点n自身的编号，n.root表示n所在循环根节点的编号，算法先初始化节点所在的根为节点自身的编号。接下来首先顺序处理n所有邻接节点，然后更新n的根节点编号，原因在于可能存在从n的邻接节点到祖先节点(编号可能比n小)的回边，这一步确保能找到最大的强连通子图。递归处理完所有邻接节点后，由每个连通子图的根节点输出该子图的所有节点，每输出一个节点，就将该节点“删除”，这样确保任何节点只可能出现在一个强连通子图中，即不可能出现多个子图交叉的情况。

还有一个实现上的细节：当找到并输出该连通子图的所有节点时，需要子图中每个节点的root域(即让子图中每个节点指向新的根)。如果在图2.10中，repeat/until循环体中增加一个语句才会更准确：v.root = n.order。这样确保连通子图中每个节点的root域指向其所在子图的根节点，否则在存在嵌套循环的时候会有问题。

关于Tarjan算法原理的更多细节请参考文献[2]，关于该算法的实现细节请参考本文第二部分基于Clang的实现参考源码。

Tarjan's Algorithm (notice the embedding of DFS)

```
unmark all nodes ; preorder := 0 ; stack := empty
Tarjan(entry)

define Tarjan(node n) {
    mark n visited
    n.root := n.order := preorder ; preorder++ ; stack.push(n)

    foreach edge (n,v) do
        if v is not visited then Tarjan(v)
        n.root := min(n.root, v.root)

    if n.root = n.order then // n is an SCC root
        repeat // identify SCC nodes
            v := stack.pop()
            delete v from graph
        until v = n
}
```

图2.10 检测强连通子图的Tarjan算法

2.5 自然循环和可化简控制流图

自然循环(natural loops)就是只有单一入口的循环。两个自然循环要么完全不相交，要么嵌套。对于没有goto语句的语言，所有的循环都是自然循环。一般循环可能存在多个到循环体的入口，如通过goto语句直接跳转到一个循环的循环体中。

如果一个控制流图仅通过如下T1和T2两种变换能化简为一个节点，则称该控制流图是可化简的(reducible)：

A. T1: 如果节点n有唯一的前驱，那么将其和其前驱合并为一个节点；

B. T2: 如果节点存在到自身的边，那么将该边删除。

没有循环的控制流图都是可约简的，对存在循环的控制流图，当且仅当所有循环是自然循环时此图是可化简的。有一些程序分析算法只对可化简的控制流图有效。

3 数据流分析

3.1 引子

数据流分析有两种基本类型：一阶(First-Order)分析关注程序状态变化，如常量传播(Constant propagation)和范围传播(range propagation)；二阶(Second-Order)分析关注程序的执行路径变化，如可用表达式分析(available expressions)和reaching definitions。这四种数据流分析各有特点，贯穿这章内容始终，表3.1概要总结了每种分析方法的目的和基本思路。

表3.1 四种基本数据流分析

方法名称	类型	分析目标
Constant Propagation	First Order	在每一个program point，分析出每个一定取值为常量的变量的值，分析结果是(变量,取值)二元组集合。
range propagation	First Order	给定program point，分析每个变量的取值范围，分析结果是(变量，取值范围)二元组集合
available expressions	Second Order	给定program point, 求出所有结果可重用的表达式集合，分析结果是表达式集合
reaching definitions	Second Order	给定program point，求出所有从入口可能到达该处的所有变量定义的集合

constant propagation和range propagation很直观。常量传播是编译器最常用的优化手段之一，目的是在编译阶段尽可能化简表达式，提高目标代码的效率。

图3.1是available expression的例子，在语句r := a+b; 处的可用表达式集合为:{c+d}。a+b不可用因为变量a可能在r赋值前被修改，同理z>0也不可用。可用表达式优化可用于避免对相同表达式进行重复计算。

图3.2是reaching definitions的例子，可能到达z := x+y;的所有变量定义集合为：{x:=2;x:=3;y:=6}。reaching definitions也是一种基本的分析，比如和上一张提到的控制依赖分析结合起来做程序切片(Program Slicing)，在3.6还会进一步谈到。

Program

```
x := a+b ;
y := c+d ;
if (z > 0) then { a := y }
           else { z := y } ;
r := a+b ;
s := c+d
```

图3.1 available expressions

Program

```
x := 4 ;
y := 6 ;
if (z > 0) then { x := 2 }
           else { x := 3 } ;
z := x + y
```

图3.2 reaching definitions

上述四种分析目标不同，但共同点都在于依据程序固有的控制依赖和数据依赖提取程序行为，因此看上去可以提取一个公共的框架来统一这些分析方法。但上述这些分析方法之间有什么差别？如何评价一个数据流分析方法的优劣？如何设计自己的数据流分析方法(基本套路是什么)？这些问题有些并不是很显然，下面的内容逐步展开说明。

3.2 解集合和格

数据流分析的底线是不能出错，否则一定会破坏程序本身固有的控制或数据依赖关系。例如在图3.1的程序中，给r赋值的时候如果直接重用前面x:=a+b的结果则破坏了程序语义。直观上数据流分析的目的是在不出错的情况下尽可能精确，这样如果用集合来表示某个程序点(program point)的解集合，那么集合之间的包含关系就描述了不同解集合之间的精度关系。对于available expressions来说，解集合越大则分析越精确，但情况并非都如此。对于reaching definitions来说，解集合越小越精确，因为在任何程序点，最保守(安全)的分析结果是“完全不知道，可能所有变量定义都会到达”，也就是变量定义全集，这一点或许不那么直观，下面将会进一步说明。总结起来就是说，解集合之间的包含关系能反应数据流分析精度，但不意味着集合越大越精确，这和具体的分析相关。

可以用集合来表示某个程序点的解，解之间的精度可以用集合关系来描述，这就完成了一点朴素直观的抽象。严格的抽象就是离散数学教科书里的集合偏序和格的概念，当初学这些概念不知道有什么用途，接下来将能看到实际问题和这些抽象概念之间的关联。

首先回顾偏序集合的概念，如图3.3所示：

Definition (Partially Ordered Set)

A *partially ordered set* (**poset**) is a binary relation $\sqsubseteq$ over a set $\mathcal{S}$ that is:

- Reflexive: $x \sqsubseteq x$
- Transitive: $x \sqsubseteq y$ and $y \sqsubseteq z \Rightarrow x \sqsubseteq z$
- Anti-symmetric: $x \sqsubseteq y$ and $y \sqsubseteq x \Rightarrow x = y$

图3.3 偏序集

偏序就是定义在一个集合上的二元关系，满足自反、传递和非对称三个条件。如果集合元素作为一个图的顶点，偏序作为图的边，那么除了到自身这种情况之外图中不能有环。而格就是一种特殊的偏序集合而已，具体要求满足两个条件：

Definition (Lattice)

A **lattice** is a poset such that

- Any 2 elements have a least upper bound (the **join**)
- Any 2 elements have a greatest lower bound (the **meet**)

图3.4 格的定义

第一个条件要求集合中任意两个元素一定有一个公共上界(upper bound)，也就是说对于任何x,y，一定有一个z，使得z>=x并且z>=y。给定两个元素求其上界的操作成为join操作。第二个条件meet对元素关系的定义和join正好相反。直观来说，从格中任意取出两个元素一定有一个上界和一个下界。回忆离散数学里面哈斯(Hasse)图，格一般都有一个最高点(Top)和最低点(Bottom)。举例来说，第2章提到的控制流图的顶点集合，加上Dominance树定义的偏序关系合起来构成格，整数集合上的<=关系，集合{1,2,3,6}上的整除关系。不构成格的偏序关系也很容易构造，如集合{1,2,3}上的整除关系(2和3没有公共上界)，集合{1,2,3,12,18,36}上的整除关系(12和18没有公共下界，2和3没有公共上界)。数据流分析的解集合对应格上的点，格上点之间的偏序关系反应了解集合之间的精度关系。为便于分析理解，在下面有关格的表示中，认为Top表示最精确的解集合，Bottom表示最不精确的解集合，这仅是记号的方便，也可以倒过来理解，格仅是理解分析结果及相互间关系的工具。

问题是对于一些无穷元素的格，如定义在整数数集合上的<=关系，没有确定的Top和Bottom，这种情况下为了分析方便可以人工引入Top和Bottom记号，从而将具有无穷个元素的格变成bounded lattice。很显然所有具有有限个元素的格一定是bounded lattice。

Definition (Complete Lattice)

A **complete lattice** is a bounded lattice for which every subset of the lattice has both a meet (⊓) and a join (⊔).

- The meet and join don't have to be part of the subset
- Contrast with a regular lattice, which only requires binary ⊓ and ⊔

图3.5 Complete Lattice定义

格的定义要求任何两个元素都有唯一的meet和join，complete lattice进一步要求任何元素集合一定有唯一的meet和join。从格的定义，可知任何有有限元素个数的格一定是complete lattice。当元素个数无穷时，格未必一定是complete。如定义在非零整数集合上的整除关系，给定任意两个非零整数，他们一定有最小公倍数和最大公约数，但这个格不是complete的，如对所有正整数构成的集合，他们有最大公约数1，但找不到公倍数，因为0已经在定义时从集合中去掉了。

Definition (Chain)

A lattice **chain** is a totally-ordered subset of lattice elements, i.e., a subset of the lattice such that every member of the subset is comparable with every other member of the subset.

Definition (Finite Ascending (Descending) Chain Condition)

A lattice meets this condition if every chain in the lattice is finite. Intuitively, one can only go up (down) the lattice so far before reaching a bound.

图3.6 Finite Ascending(Descending) Chain

直观理解，所谓chain，就是从格中抽取部分元素，这些元素之间具有全序关系，一个格可能有无穷多个chain。而Finite chain就是要求格中不存在具有无穷个元素组成的chain。这一个有限性约束是任何数据流分析迭代能结束的必要条件。

给定一个数据流分析问题，第一步就是抽象解空间并构造一个complete, finite ascending/descending lattice。解空间的格规定了整个分析过程的边界，从而实现了对实际的数据流分析问题的抽象。表3.2总结了4种基本数据流分析的解空间格。

表3.2 四种基本数据流分析的解空间格

名称	解集合	meet/join 操作定义	Top/Bottom	解空间格性质
Reaching Definitions	变量定义集合	meet(join)操作是对解集合求并(交)集	Top-空集 Bottom:所有变量定义的集合	complete finite AS/DS chain
Available Expressions	可用表达式集合	meet(join)操作是对解集合求交(并)集	Top-所有可用表达式集合 Bottom:空集	complete finite AS/DS chain
Constant Propagation	常量集合	a∩b = ⊥unless a=b a∪b = T unless a=b	Top-uninitialized Bottom:unknown	complete finite AS/DS chain
Range Propagation	取值范围的集合	[a,b]∩[c,d]=[min(a,c),max(b,d)] [a,b]∪[c,d]=[max(a,c),min(b,d)]	Top-空集 Bottom:[-∞,+∞]	complete

理解每种分析的解空间，要点是meet操作的语义。在控制流图上，从程序入口到达某个program point可能存在多个路径，每个路径对应一个解集合，那么怎么合并多个路径上的解集合，就是meet操作。就reaching definitions来说，其分析目的是找出可能到达某程序点的变量定义集合，因此meet就是集合求并，而available expressions分析正好相反。由于解集合是有限的，对于有限元素的格，同时满足complete lattice和finite ascending/descending chain的条件。

常量传播的目的是给定program point，找出能到达该处的所有常量，这里meet操作不是集合交并操作。加入同时到达该处的两条路径有对应某变量的不同赋值，那么从该program point开始该变量的值就是未知(unknown)的，因为不知道实际的执行路径。引入uninitialized一方面是为了构造complete lattice，同时也表示变量定义后的未初始化状态。对于常量传播来说，解空间格有无穷多个元素，但是满足complete和finite AS/DS chain的条件。

range propagation的目的是分析给定program point，分析变量取值范围。在程序实际执行路径未知的情况下，保守方法就是放大可能的取值范围，这就是meet操作，以此可以得出join操作，top/bottom的定义。注意这里解空间格也有无穷个元素，但不具有finite AS/DS chain，因为从Top到Bottom之间可能存在无穷多个解。

3.3 传输函数

至此回答了数据流分析设计的第一个问题：如何抽象解空间并构造格，不同分析的格之间的异同。考虑一个由基本块组成的程序控制流图，meet操作规定了考察基本块内语句之前对所有路径上来的输入集合的处理，接下来的问题是如何处理基本块内的语句并生成新的解集合，这就是传输函数(Transfer Function)。概要来说，传输函数接受一个输入解集合并生成一个输出集合，输入/输出集合都是解空间格上的元素。因此，抽象来说传输函数就是定义在解空间格上的元素之间的映射：

Definition (Transfer Function)

A **transfer function** $F : \mathcal{L} \to \mathcal{L}$ models the effect of a program statement on an abstract domain.

图3.7 传输函数

对于传输函数，从数据流分析的角度关心三个方面：是否monotone，是否distributive以及是否continuous。

Definition (Monotonicity)

A function F is **monotone** iff $x \sqsubseteq y \Rightarrow F(x) \sqsubseteq F(y)$

- It turns out this definition is equivalent to requiring that:
  - $F(x \sqcap y) \sqsubseteq F(x) \sqcap F(y)$

图3.8 传输函数的单调性定义

定义中$\sqsubseteq$不是集合包含，而是定义在解空间格上元素之间的偏序关系，$\cap$不表示集合交而是解空间格上元素之间的meet操作。传输函数的单调性主要表达这样一种直觉(common sense)：如果一个解集合x不如解集合y精确，那么在应用传输函数之后的解集合F(x)同样不应该比F(y)精确。图3.8所示单调性另一种表达形式是说，给定解集合x和y，在x meet y之后应用F得到的解不如分别对x和y应用F后求meet结果得到的解集合更精确。直观上，第一种形式更好理解，而在数据流分析中，对多条可能路径上输入集合先求meet再应用F，和先各自应用F之后求meet得到的结果有重要的差异(后面会详细说明)。传输函数单调性的两种表达方式是等价的。

传输函数的第二个重要性质是distributivity，定义如图3.9所示，是说如果传输函数distributive，那么在应用传输函数前或应用传输函数后meet得到的解集合相同。后面将进一步提到数据流分析中，如果传输函数满足这一条件将能得到相对精确的解集合。

Definition (Distributivity)

A function F is **distributive** iff $F(x \sqcap y) = F(x) \sqcap F(y)$

图3.9 distributivity的定义

continuous的定义是对distributive的推广，类似complete lattice和lattice之间的关系。定义如下：

Definition (Continuity)

A function F is **continuous** iff $F(\sqcap X) = \sqcap F(X)$ for any subset X of lattice elements.

图3.10 continuous

这里区别在于把任意两个解集合推广到格中的任何元素子集。下面依次分析四种基本数据流分析的传输函数及性质。

3.3.1 Reaching Definitions

给定按照语句粒度(每个语句一个节点)构造的控制流图CFG，对CFG中的给定节点k，分析对变量x的reaching definitions传输函数的基本形式如下：

$Fk(IN) = (IN - KILLk) \cup GENk$

分两种情况：

(1) 如果节点k定义了变量x，则

$GENk = \{k\}$

$KILLk = \{n \mid n \in N - \{k\} \wedge n \text{ defines } x\}$

(2) 如果节点k没有定义变量x，则：

$GENk = KILLk = \{\}$

也有文献称Fk(IN)的计算过程为数据流方程，看上去确实有点方程的形式，但和一般数学方程不是一回事。就这一分析来说，如果节点k定义了变量x，那么之前的所有对x的定义已经无效，即KILLk集合，同时产生了一个新的对x的定义，即GENk。传输函数的处理就是依据节点k是否对x赋值修正解集合的内容。

这一传输函数同时满足monotone, distributive, continuous三个条件。直观上理解这一点的线索是KILLk和GENk集合只和节点k本身有关，和输入集合IN无关。

3.3.2 Available Expressions

同样给定语句粒度的控制流图CFG，节点k处传输函数形式如下：

$Fk(IN) = (IN - KILLk) \cup GENk$

同样分两种情况：

(1) 对节点k定义的任何变量x，有

GENk = {E | E is used by k and x ∉ FV(E)}
KILLk = {E | x ∈ FV(E)}
(2) 如果节点k没有定义任何变量，则

GENk = {E | E is used by k}
KILLk = {}
FV(E)表示表达式E用到的变量集合。GENk表示节点k用到的表达式集合。如果节点k定义了变量x，则KILLk表示之前用到了变量x的所有表达式集合，表示这些表达式从节点k开始不再可用。

这一传输函数同时满足monotone, distributive,continuous三个条件。

3.3.3 Constant Propagation

考虑一种简化的情况，在只有+,-,*,/的表达式中进行常量传播。假设对于表达式a+b，直观上只有a和b都是常量的情况下才有意义，其他情况则是未知或未初始化，如图3.11所示。

F_+	$\top$	c_1	$\bot$
$\top$	$\top$	$\top$	$\top$
c_2	$\top$	$c_1 + c_2$	$\bot$
$\bot$	$\top$	$\bot$	$\bot$

F_-	$\top$	c_1	$\bot$
$\top$	$\top$	$\top$	$\top$
c_2	$\top$	$c_1 - c_2$	$\bot$
$\bot$	$\top$	$\bot$	$\bot$

图3.11 加/减表达式的常量传播结果

依据图3.11的定义传输函数满足monotone条件。注意c + T = T规则，如果修改(如定义为c + T = ⊥，尽管看上去很合理：常量加未初始化的数结果为unknown)将破坏monotone。原因如下：

考虑传输函数F(x) = x + c1，则有：

F(c) = c + c1
F(T) = T + c1 = ⊥
注意到c ⊆ T，但c+c1 ⊇ ⊥，于是破坏的单调性。

常量传播的传输函数是否满足distributive？考察图3.12的例子：

Program

```
if (x < 0) then { a := 3 ; b := 2 }
           else { a := 2 ; b := 3 } ;
c := a + b
```

图3.12 常量传播破坏distributive的例子

考察CFG节点c := a+b，有两条可能的路径对应的输入集合：A: {a := 3; b := 2} 和 B: {a := 2; b := 3}，那么：

F(A∩B) = F(⊥) = ⊥，

而F(A) ∩ F(B) = {5}，因此F(A∩B) ⊂ F(A) ∩ F(B)，不满足distributive，因此更不满足continuous性质。

3.3.4 Range Propagation

和常量传播一样，首先需要定义范围传播的基本运算规则，如图3.13所示：

F_+	[a..b]	+	[c..d]	=	$[(a+c)..(b+d)]$
F_-	[a..b]	−	[c..d]	=	$[(a-d)..(b-c)]$
F_*	[a..b]	*	[c..d]	=	$[(a*c)..(b*d)]$
$F_\div$	[a..b]	/	[c..d]	=	$[(a/d)..(b/c)]^*$

图3.13 范围传播的运算规则

和常量传播的情况类似，范围传播只满足单调性。

3.4 数据流分析框架

至此数据流分析的三大要素已经具备：控制流图CFG、解集合格和传输函数，接下来需要解决的问题是通用数据流分析框架的确立。

静态程序分析无法枚举程序在所有可能输入下的行为，因此必须将实际程序行为映射到一定的抽象域(抽象解集合域)上进行，比如将对变量值的分析映射为奇偶性分析，然后从程序入口开始抽象解释(执行)每条可能的路径，最终得到解集合。反应在传输函数上，这种分析要求先对所有输入集合先应用传输函数，最后对结果求meet得到输出集合。以输入为A和B两个集合为例，传输函数的输出为F(A)∩F(B)。这种分析方法称为Meet-Over-All-Paths (MOP)分析，直观上就是说对每条输入路径的解集合分别应用传输函数，最后再通过meet操作来求得最终输出解集合。

MOP方法能得到尽可能精确的分析结果，但存在三方面的缺陷：(1) 分析算法的复杂度通常都是指数级，因为需要对每条输入路径分别应用传输函数；(2) 有些分析无法终止(terminate)，如常量传播遇到循环时可能无穷迭代；(3) MOP分析的一般问题是NP难的。因为这些因素使得MOP分析不具备很强的工程实现价值。

更实际的方法是所谓最大不动点(Maximal Fix Point)分析。Maximal表示尽可能使解集合最精确。Fix Point相对于解集合和传输函数而言，换言之给定传输函数F和解集合X，如

果有F(X) = X，那么则称X为传输函数F在解空间格上的不动点。MFP分析和MOP分析二者之间的差别是直观的，对于控制流图CFG的节点k，有：

MOP(k) = ∩ {Fk ∘ ... ∘ F0 | 0 ... k表示CFG里的路径)

MFP(k) = Fk (∩ {MFP(p) | p ∈ pred(k)})

对于节点k，MOP(k)需要枚举程序入口到k所有路径，对每条路径一次应用传输函数，最后将所有路径上得到的解集合求meet。MFP(k)只考察其前驱节点(pred(k))的输出，求meet之后再应用传输函数求取结果。直观上，MFP能将整个数据流分析控制在多项式复杂度，问题是MFP能多大程度上hold住MOP能获得的精度。

首先对于定义在解空间格上的传输函数F，只要F是单调的，那么不仅存在不动点，而且所有的不动点也构成一个complete lattice，即Knaster-Tarski定理：

Definition (Knaster-Tarski Fixpoint Theorem)

Let $\mathcal{L}$ be a complete lattice and $F : \mathcal{L} \to \mathcal{L}$ be a monotone function. Then the set of fixpoints of F in $\mathcal{L}$ is also a complete lattice.

3.14 Knaster Tarski定理

这就是一定要求传输函数单调的原因，否则不能保证解空间格中存在不动点，这意味着无法保证能获得精确的分析结果，因为分析过程可能无法终止。在确定不动点存在的情况下，如何求得不动点，Kleene不动点回答这一问题：

Definition (Kleene Fixpoint Theorem)

Let $\mathcal{L}$ be a complete lattice and $F : \mathcal{L} \to \mathcal{L}$ be a continuous (and hence monotone) function. Then the maximal fixpoint of F is the infimum (GLB) of the descending chain $\top \sqsupseteq F(\top) \sqsupseteq F(F(\top)) \sqsupseteq \ldots$.

图3.15 Kleene不动点定理

这就是说只要传输函数进一步满足连续性(continuous)条件，则不动点可以从Top出发，不断应用传输函数而得到。直观上在解集合上不断应用传输函数使得解从格的顶端(最精确)向下移动，因此只要格满足finite descending/ascending chain的条件，那么这个迭代过程一定能终止，终止的位置得到的解就是分析的最终结果(maximal fix point)。

这意味着，如果传输函数是连续的(reaching definitions, available expressions)，那么MFP分析能得到MOP一样的结果，即总能通过足够多次数的迭代收敛到解空间格的最大不动点。反之，如果传输函数单调但不连续(constant/range propagation)，那么MFP分析也能得到结果，但不能保证是最精确的结果。

Kleene定理意味着可以构建一个通用的数据流分析算法，只要从T开始不断应用传输函数，总能收敛到不动点，如图3.16所示。这个只是原理性算法，效率很低，而且初始化时

不能简单把所有节点的输入集合初始化为T。以reaching definitions分析为例，T表示所有变量的定义集合，算法从CFG的入口节点开始，由于所有输入集合初始化为变量全集，整个算法都无法结束。因此初始化时，入口节点的输入集合必须初始化为空，其他所有节点的输入集合初始化为T。

DFA Algorithm

$\forall k \in N \,.\, \text{IN}_k = \text{OUT}_k = \top$
repeat
  foreach $k \in N$ **do** {
    $\text{IN}_k = \sqcap\{\text{OUT}_p \mid p \in pred(k)\}$
    $\text{OUT}_k = F_k(\text{IN}_k)$
  }
while solution changes

图3.16 通用数据流分析算法

图3.16的算法是低效的，原因是只要有一个节点的解集合有变，在下一轮迭代所有节点要重算，导致那些早就到达不动点(解集合不再变化)的节点重复计算。直观上，当有节点解集合变化时，只有受该节点输出影响的节点需要重新计算解集合，因此有了图3.17所示的工作队列算法。

Worklist Algorithm

$\forall k \in N \,.\, \text{IN}_k = \text{OUT}_k = \top$
$\text{Worklist} = N$
repeat
  $k = \text{SELECT}(\text{Worklist})$
  $\text{IN}_k = \sqcap\{\text{OUT}_p \mid p \in pred(k)\}$
  $\text{OUT}_k = F_k(\text{IN}_k)$
  if OUT_k changed **then** Worklist $\hookleftarrow$ succ(k)
while Worklist not empty

图3.17 基于工作队列的数据流通用分析算法

这个算法是很多实际数据流分析的基础，在实践部分会介绍Clang中的两个数据流分析例子，二者都基于这一框架。有两点要注意：(1) 工作队列如何组织，常用方法是FIFO队列、优先级队列、或者按照reverse post order等顺序等等；(2) pred(k)未必一定是CFG图中的前驱节点，对于后向数据流分析(live variables)，pred指的是CFG图中的后继。

对于range propagation，由于不满足finite AS/DS chain条件，图3.17的算法无法终止。这种情况下可以引入所谓的widening函数，直观上就是把解空间格稍微调整一下(代价就是结果更不精确)，使得其满足finite AS/DS chain的条件，例如该函数可以这样定义：当区间[a,b]足够小的时候直接就用(a+b)/2表示；当区间足够大时，直接用(-∞, +∞)(⊥)表示。足够小/足够大的量化参数依据精度需求和迭代速度折中。

数据流分析依据其一些基本特点可以进行分类。一种分类就是所谓separable/non-separable分析。比如reaching definitions, available expressions，可以按照不同的变量单独分析，最后把结果合起来，这样操作和考虑所有变量同时分析结果相同，也就是说对不同变量的分析过程是完全独立的，这种分析就是separable分析。而如常量/范围传播分析则属于non-separable分析。

依据数据流分析过程中传输函数的数据流向，可以分为前向分析(forward)和后向分析(backward)分析。前面讨论的四种分析过程都是前向的，即从程序入口节点开始以此往后迭代；典型的后向分析是live variables分析，给定程序点，求将来可能用到的变量集合，用于编译器的寄存器分配和死代码删除等优化，这种分析主要考察“将来”是否会用到变量，因此需要从程序出口逆向分析，即所谓backward分析的由来。后向分析的过程和前向分析一样，只要把解空间格倒过来，用一样的方法迭代即可。

3.5 数据流分析评估指标

数据流分析的评估指标包括四个方面：复杂度、精度(precision)、迭代是否能结束(termination)和结果的正确性(soundness)。

数据流分析算法的复杂度是O(e*h*f)，其中e是CFG中边的数目，h表示解空间格的高度(即从TOP到Bottom的距离)，f是传输函数本身的复杂度。对于available expressions和reaching definitions，复杂度为O(e^3)，常量传播的复杂度为O(e*v^2)，其中v为变量的数目。

数据流分析的精度主要考察MFP能多大程度接近MOP的精度，这主要取决于传输函数本身的性质。如果传输函数连续，则MFP和MOP精度一样；否则对于单调函数MFP相对于MOP会损失精度。

数据流分析迭代终止的条件是传输函数单调且满足finite AS/DS chain条件。

给定一个数据流分析算法，怎么知道其得到的结果是正确的？虽然直观上就是觉得正确，但在理论上严格说清楚并非容易的事。soundness定义如图3.18所示，即在任何program point计算得到解集合必须构成对程序实际行为的保守近似(over approximation)。例如常量传播分析，保守近似意味着分析的常量集合必须是实际程序执行到该程序点可能出现的

常量集合的子集，也就是说可以有一些常量没有发现，但不能误报，否则就破坏了程序语义；而对于reaching definitions分析，保守近似意味着实际可能到达该program point的变量定义集合必须是解集合的子集。soundness提出的问题是为什么给定的数据流分析方法能保证保守近似的关系。

Definition (Soundness)

The computed abstract value at a given program point is an over-approximation of the concrete collecting semantics at that program point.

图3.18 soundness

严格回答soundness的理论工具是Galois Connection，详细内容见参考文献：abstract interpretation相关论文。基本思想是直观的，给定：

描述具体程序执行行为的格(concrete domain)：C

描述抽象域解空间的格(abstract domain)：A

从具体域到抽象域的抽象函数：α: C->A

从抽象域到具体域的映射函数：β: A->C

具体域(concrete domain)的运算符：$[.]_C$: C->C

抽象域操作符：$[.]_A$: A->A

满足soundness条件等价于：给定$c \in C$: $\beta([\alpha(c)]_A) \subseteq [c]_C$。具体域运算符就是程序中表达式运算符，如+,-,*,-,指针，sizeof等等，抽象域运算符是在抽象值域对这些具体操作符语义的抽象，如图3.11 constant propagation分析对+/-运算符的抽象。这一条件就是说，给定具体域的值c，将其映射到抽象域(α(c)，然后在抽象域中应用抽象操作符($[\alpha(c)]_A$)，再将结果映射回具体域($\beta([\alpha(c)]_A)$)，其结果一定不比直接在具体域应用操作符($[c]_C$)更精确。

3.6 更多数据流分析举例

设计一个数据流分析的基本过程：(1) 设计抽象域解空间格；(2) 定义传输函数；(3) 分析方法的复杂度、精度、终止条件及soundness分析。下面是几个实际中常用的例子。

3.6.1 Definitely Initialized Variables

在编译优化中，这一分析可以用来警告对未初始化变量的使用。首先构造抽象解空间格。格中的元素，即解集合是(初始化过的)变量集合。这是前向分析，meet操作是集合求交集(∩)，因此Bottom是空集，Top是所有变量的集合。

再来看传输函数，考虑语句粒度的控制流图(节点是语句)，在节点k，如果节点定义了变量x，则GENk = {x}，否则GENk={}。KILLk 始终是空集。传输函数满足monotone, distributive, continuous。
由于是对程序中变量分析，因此分析复杂度上界O(e * v^2)；因为传输函数满足distributive条件，故MFP能实现MOP一样的精度；由于解空间格满足complete, finite AS/DS chain条件，故分析一定能终止。此分析还满足soundness条件。
实际实现基于工作队列的迭代算法时，程序入口节点的输入集合初始化为空，所有其他节点的输入解集合初始化为Top，即变量全集。

3.6.2 Live Variables

在编译优化中，这一分析最常用来删除死代码和寄存器分配。首先构造抽象解空间格。格中的元素，即解集合是(活动的)变量集合。注意这是逆向分析，meet操作是集合求并操作，就是说在某个程序点，只要变量在后续任何路径上被使用，那么该变量就是活动变量(Live)。解空间格Top是空集，Bottom是所有变量集合。
再看传输函数。同样考虑语句粒度的控制流图，在节点k，如果节点定义了变量x，则直观上相当于告诉前面的节点该变量被重新赋值(死)了，KILLk={x}，否则KILLk={}。GENk是节点k用到的变量集合(不包括同时被赋值的变量，例如对于语句x++，不能将x添加到GENk集合)。
同样由于是分析程序变量，故分析复杂度上界O(e * v^2)；因为传输函数满足distributive条件，故MFP能实现MOP一样的精度；由于解空间格满足complete, finite AS/DS chain条件，故分析一定能终止。该分析同样满足soundness条件。
实际实现时，所有节点的输入解集合初始化为空(Top)。

3.6.3 Program Slicing

程序切片在调试、软件工程等很多方面都有用。问题是这样的：给定一个程序点(program point)，找出所有对该程序点计算结果有贡献的所有语句集合。从这个问题陈述来看，显然是逆向分析。切片问题也可以换一种前向分析的提法，这里以逆向分析的情况为例。
首先构造解空间的格。格中元素，即解集合显然是语句集合。meet操作是集合并，因为只要和目标程序点计算结果相关的所有可能路径上的指令都需要被包含在内。一旦明确meet操作，则解空间格的Top是空集，Bottom是所有语句集合。
再看传输函数。考虑语句粒度的控制流图，在节点k，只有两种情况：节点k的语句是否需要被添加到解集合中，这需要考察节点k和传输函数输入解集合中所有语句之间的依赖关

系。如果节点k没有定义任何变量，那么显然不需要将节点k添加到输出解集合中，于是只需要考虑k对变量有定义这种情形，设被定义的变量为x。
考察传输函数的输入集合，给定集合中的任意节点p，如果p使用了变量x，而且k是p的reaching definition，则节点k需要被添加到解集合中。还有一种情况，如果节点p对节点k有控制依赖(p is control dependent on k)，那么节点k也需要被添加到输出解集合中。控制依赖的严格定义见前文控制流分析部分。程序切片问题相对复杂，需要同时用到reaching definitions和控制依赖分析的结果。
程序切片是对语句的分析，在reaching definitions和control dependence结果已经得到的情况下，复杂度为O(e^3)。这里传输函数同样满足distributive条件，虽然输出集合和输入集合的内容相关，但传输函数对输入集合中所有语句的处理之间是独立的，因此MFP能实现MOP一样的精度。由于解空间格是有限的，满足complete, finite AS/DS chain条件，故分析过程一定能结束。程序切片分析满足soundness条件。

4 稀疏(sparse)分析

所谓稀疏分析主要指基于SSA的分析方法。对有些应用，SSA的引入能大幅提高分析效率。现代的编译器，如GCC, LLVM，后端优化都基于SSA来做。SSA相关研究的基础性论文是Cytron et al “Efficiently Computing Static Single Assignment Form and the Control Dependance Graph”.下面讨论的所有内容都没有越过这篇文章的范围。
为什么需要SSA? SSA是什么? 用什么算法来计算SSA? 下面说明这些问题。

4.1 why SSA?

简单的数据流分析有很严重的效率问题，图4.1是一个常量传播的例子。语句1发现x的值是常量，于是传播给所有后继，图中即节点2，其实节点2只是定义了变量y，和变量x没任何关系，对常量传播来说，直观上更有效的方法是把对变量的定义分析直接传播给对变量的使用节点进行进一步分析。由此可知，简单的DFA存在冗余信息传播问题，对于大型程序分析(如对整个Linux内核做指针分析)，导致的效果是高存储开销、低效的传输函数(解集合包含冗余元素)、昂贵的解集合运算(如求meet)。关于这个问题的进一步说明请参考这篇论文Ben et al “Semi-Sparse Flow Sensitive Pointer Analysis”。

Example (Constant Propagation)

```
[1]x := 4 ; [2]y := 1 ;
[3]if (_) then { [4]y := 3 ; [5]z := x } else { [6]z := 3 } ;
[7]a := y+z
```

图4.1 DFA的效率问题

解决这个问题简单方法就是构造def-use链，对每个变量，跟踪代码中对其的定义和使用，从而可以依据这一依赖关系构造依赖关系图。数据流分析的信息传播依据该依赖关系图来传播而不是盲目将解集合传播给控制流图中的后继。def-use链的构造方法分两步：(1) 计算reaching definitions，具体内容见前面数据流分析部分；(2) 考察变量x，对任何定义变量的节点Nd和任何使用该变量的节点Nu，如果Nd在Nu的reaching definitions集合中，则添加一条从Nd到Nu的边。

问题是，简单def-use链可能导致依赖关系边数目迅速膨胀，从而抵消掉所有因减少冗余信息传播带来的好处。图4.2是一个依赖关系边数目膨胀的例子。

Example

```
if (_)      then { x := _ }
else if (_) then { x := _ }
else             { x := _ } ;

if (_)      then { _ := x }
else if (_) then { _ := x }
else             { _ := x }
```

图4.2 def-use关系存在的问题

根据def-use关系的定义，每个对变量x定义和使用节点之间都必须有一条边，对这个例子就是3x3，实际程序中这个问题会严重得多。为解决DFA的效率问题需要减少冗余信息传播，而简单def-use链会引入大量额外依赖关系边，这是引入SSA的动机。本质上来说，SSA还是def-use链，但是对程序进行合理的变换控制def-use边的数目。

4.2 Static Single Assignment (SSA)

SSA的定义如图4.3，两个条件：(1) 任何变量只在一个语句被定义；(2) 对变量的任何使用都只有一个reaching definition。对程序进行SSA变换的一般处理就是对变量重命名，如图中的例子：每次赋值重命名一次变量，保证每次使用只看到一个对该变量的定义，这里变量重命名的思想和CPU中寄存器重命名的道理一样，只是解决的问题略有不同。这

里的目的是将少def-use边，诀窍是通过重命名每次变量使用都只看到一个变量定义；而寄存器重命名的目的是消除寄存器名字冲突带来的伪依赖。

Definition (Static Single Assignment Form)

A program is in **Static Single Assignment (SSA) Form** if every variable is defined with exactly one program statement and every use of a variable has exactly one reaching definition.

Example (Original Program)

```
x := 4 ;
y := x ;
x := 6 ;
y := x
```

Example (SSA Form)

```
x1 := 4  ;
y1 := x1 ;
x2 := 6  ;
y2 := x2
```

图4.3 SSA

static single assignment中的"static"的意义在于这种表示形式不关心动态运行行为，只关注静态程序结构，如while(1) do { x = x +1; }本身是SSA，尽管变量x在运行时可能被定义很多次。

对于图4.2的例子，不能通过简单重命名变量来实现SSA，因为对变量的使用来自对变量的多个定义位置，因为存在到达变量使用的不同执行路径。为此需要引入一个选择符phi，定义如图4.4所示。

Definition (ϕ Function)

For a given variable x for which multiple definitions reach a program point p, a ϕ **function** combines the set of incoming definitions into a single new definition.

The ϕ function is a *choice operator* that multiplexes the incoming definitions.

图4.4 phi操作

phi操作是一个路径选择符号，当有多个变量定义都能到达某程序点，那么就在该程序点放置一个phi符号，以此来合并到达此处的多个变量定义，phi操作返回程序执行时实际到达该处的变量定义。引入phi操作后，图4.2的SSA形式如图4.5所示。

Example

```
if (_)      then { x1 := _ }
else if (_) then { x2 := _ }
else             { x3 := _ } ;
x4 := φ(x1, x2, x3) ;
if (_)      then { _ := x4 }
else if (_) then { _ := x4 }
else             { _ := x4 }
```

图 4.5 phi操作的路径选择

4.3 SSA的构造算法

SSA的构造算法有两步：(1) 在合适程序点放置phi操作，解决办法是引入dominance frontier；(2) 重命名变量，结合reaching definitions分析完成。

4.3.1 放置phi节点

简单方法是分析控制流图，对每个执行路径的汇合点，对每个变量放置一个phi节点。尽管正确，但过于保守。更精确的方式：如果节点A和B都定义了变量x，且存在从A和B到节点C的不相交的两条执行路径，那么在C为变量x放置一个phi节点。由于从A到C和从B到C的路径不相交，因此A和B都不是C的dominator (dominance相关内容见前文控制流分析部分)，直观上引入dominance frontier的定义：

Definition (Dominance Frontier (DF))

$$\mathrm{DF}(n) = \{x \mid \exists p \in pred(x) \,.\, n \ \mathrm{DOM}\ p \wedge \neg(n \ \mathrm{DOM!}\ x)\}$$

图 4.6 dominance frontier定义

对于节点n，其dominance frontier (DF)是一个集合，对集合中的任何元素p，n不是p的strict dominator，但n一定是p某个前驱的dominator，也就是说DF(n)描述一组"恰好"不被n dominates的节点集合。对任何节点n，DF(n)肯定是多条路径的合并点，也就是说DF(n)表达的位置是phi节点应该存在的地方。

图4.7形象地表达了DF的意义。图中蓝色部分表示被节点5 dominates的节点，红色部分就是节点5的dominance frontier，其中每个节点都是一个多路径合并点，因此是phi操作的放置地点。需要注意，节点5属于自己的dominance frontier，因此节点不严格 (strictly)

dominates自身，但节点5有一条到节点8的回边，5是8的immediate dominator。DF定义中的strict dominates关系约束用来包含存在循环条件下的合并点，如图中的节点5。

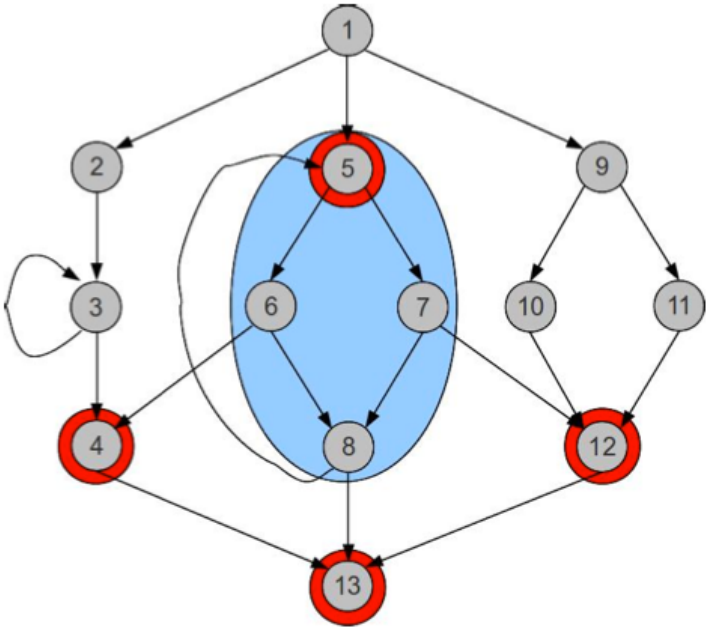

图4.7 Dominance Frontier 图示

有了dominance frontier，只要从变量的定义集合出发不断应用DF操作，每次迭代将结果添加到输入集合进行下次迭代，直到集合不再变化位置，集合中每个节点就是需要为该变量放置phi节点的所有位置。

Definition (DF Algorithm)

foreach $x \in N$ **do**
  if $|pred(x)| > 1$ **then**
    foreach $p \in pred(x)$ **do**
      $n := p$
      while $n \neq \text{IDOM}(x)$ **do**
        add x to $\text{DF}(n)$
        $n := \text{IDOM}(n)$

图4.8 DF求解算法

DF求解算法如图4.8所示。对CFG中的每个节点x，内部的foreach循环找到所谓n使得x属于集合DF(n)。首先如果x只有一个前驱，意味着从前驱只可能有一条路径到达x，无需处理；如果x有多个前驱，则依次遍历寻找x和合并点的所有可能，主要while条件用到IDOM关系，因此该算法需要先构造dominance树。dominance tree是比dominance frontier更基础更有用的分析方法，LLVM和Clang都有完全的实现。

dominance frontier还有个妙用，如果倒过来看DF关系，恰好符合control dependance的定义(见前文控制流分析部分)。因此，可以把CFG倒置，然后对这个倒置的图求DF关系，直接就能得到所有的控制依赖关系。详细内容请参考Cytron等人的那篇经典论文。

4.3.2 变量重命名

变量重命名分三步实现：

A. 在每次对变量的定义处，重命名该变量，如前面图中所示给每个定义一个下标表示每次对变量的定义；

B. 计算reaching definitions，由于每次变量定义被重命名过，因此每次对变量的使用都只有一个定义，依次重命名phi节点中的被合并的每个变量；

C. 同理由于每次变量使用只有一个变量定义可达，在每次变量使用位置重命名变量。

将SSA变回普通表示形式主要是移除phi节点，简单的处理方式是将phi节点退化为赋值语句然后插入到合并前的每条路径。

4.4 基于SSA的程序分析举例

基于SSA的数据流分析通过def-use链传播解集合信息，同时由于phi节点的引入压缩了def-use边的数目，因而成为优化的重要工具。图4.9是一个基于SSA进行常量传播的例子，可以初步体会SSA对提高分析效率的作用。

SSCP Algorithm

Worklist := {}

foreach $v \in V$ **do**
  if v is a constant k **then** set v to k and add v to Worklist
  else set v to $\top$
repeat
  v := SELECT(Worklist)
  foreach variable x defined using v **do**
    compute new value for x
    if x's value changed **then** add x to Worklist
while Worklist isn't empty

图4.9 基于SSA的常量传播

算法基本框架和DFA相同，都是基于工作队列(worklist)迭代。首先初始化所有常量赋值，对非常量赋值的变量初始化为TOP，然后进入迭代循环。从工作队列中获取常量v，对所

有使用v的变量定义(依据def-use关系传播解集合)，重新计算变量x，如果结果发生变化则将x添加到worklist，循环直到worklist为空为止。

这个算法有些细节要注意，下面举一个例子来说明。算法初始化时，worklist = {x1 = 2; y1 = 3; z1 = 4; x2 = 5; x3 = 6; y2 = 7; }，其余变量初始化为TOP。这里phi操作按常量传播的meet处理。在循环的第一个迭代中，处理变量x4和x5 (对其他变量的处理过程没有要注意的细节故略)时，由于x5用到x1和x4，x1 = 2，x4 = TOP(初始化时不是常量的所有变量初始化为TOP)，因此x5 = x1 meet x4 = 2 meet TOP =2，由于x5从TOP变为2，故将x5 = 2加入worklist。处理x4时，x4 = x2 meet x3 = 5 meet 6 = BOTTOM，故x4从TOP变为BOTTOM，故将x4加入worklist。下一轮循环时，由于x5 = phi(x1, x4)用到x4，故x5需要重新计算，此时x5 = x1 meet x4 = 2 meet BOTTOM = BOTTOM，x5从2变为BOTTOM，因此将x5重新添加到worklist，重新循环，此时解空间不再变化。

Program

```
x1 := 2 ;
y1 := 3 ;
z1 := 4 ;
if (_) then {
   if (_) then { x2 := 5 } else { x3 := 6 } ;
   x4 := φ(x2, x3) ;
   a1 := z1
} else { y2 := 7 } ;
x5 := φ(x1, x4) ;
y3 := φ(y1, y2) ;
a2 := φ(0, a1) ;
a3 := x5 ;
b1 := y3 ;
c1 := z1
```

图4.10 常量传播举例

5 指针分析

5.1 Lingo的指针扩展

扩展指针后的Lingo语法如图5.1所示，主要有四个方面：(1) 类型系统需要增加指针类型(ref type)，但这里并不区分指针的间接级别，比如a = ref b; b = ref a;是允许的，在C语言

里不能这么操作；(2) 这里指针操作符用!表示，对指针指向的变量赋值：!x = rhs (right hand size expression)；(3) 指针的初始化方式，可以直接用ref v赋值，或者用new操作符动态分配内存地址，这里ref相对于C的取地址&；(4) 对指针指向变量的引用!x。

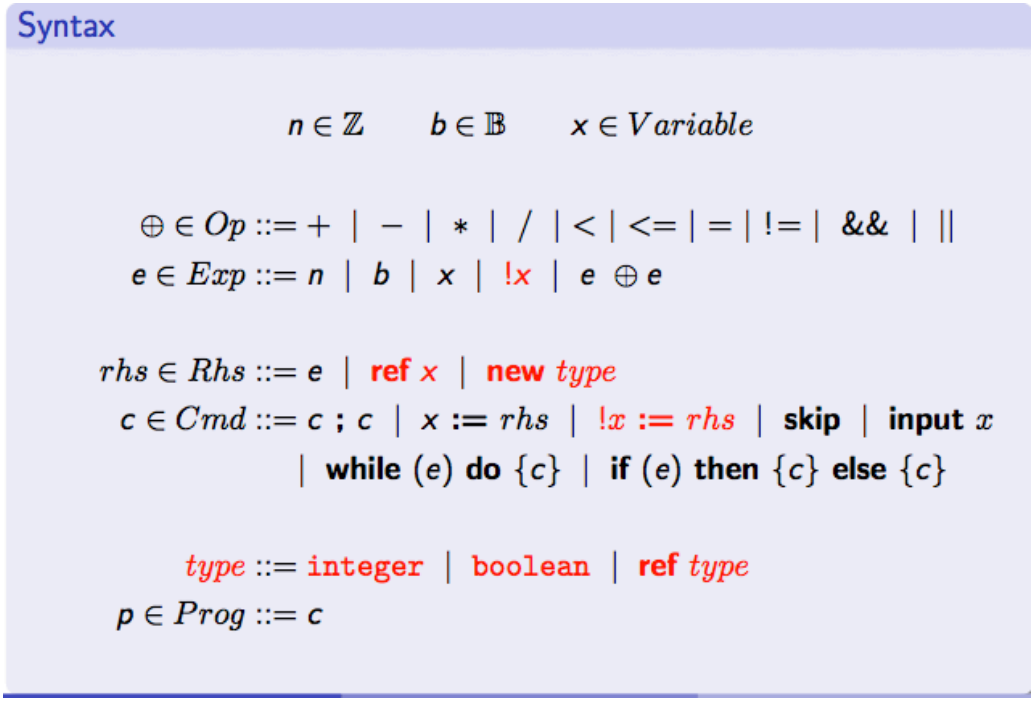

图5.1 Lingo 的指针扩展

一旦引入指针，很多数据流分析方法(available expression, reaching definitions, constant propagation, range propagation, program slicing, etc)都需要重新修正。

5.2 基本指针分析

这章内容考察的指针分析都基于数据流分析框架，从而是路径相关(flow sensitive)的分析，在后续内容会讨论路径无关的指针分析，以及考察函数调用上下文的分析(context sensitive)。

直观上说，指针分析的目的就是通过考察程序行为估计指针可能指向哪些变量的地址。指针实际指向的变量地址只有在运行时可知，指针分析期望尽可能准确估计指针指向的变量集合。在最保守的情况下，可以假设任何指针可能指向任何变量，从而直接得到指针分析结果为变量全集，虽然简单，但价值很有限，指针分析的难点在如何高效、如何更精确。

更准确一点，指针分析就是要通过考察程序行为获得所有指针尽可能精确的points-to关系，如果x points-to y (x->y)，那么x指向变量y的地址。

接下来从数据流分析的角度，设计一个基本的指针分析，三个方面：解空间格、传输函数和指针分析复杂度。

解空间格，即指针分析可能的解空间，就是对每个指针变量求出所有可能的points-to关系构成的集合，直观上，分析结束能知道每个指针变量指向的变量地址集合。对于指针分析，MEET操作是集合求并(JOIN操作是集合交)，因而格的TOP是空集，BOTTOM是所有可能的points-to关系全集。由于解空间格是有限的(finite)，因此是complete，并且有finite ascending/descending chains。

传输函数的基本形式都是：OUT = (IN - KILL) U GEN，对Lingo语言而言，需要针对指针相关的几种语句分情况讨论：

A. x = ref y. 建立一个x到y的指针关系(x->y)，KILL集合即从IN中删除x之前的指针关系，GEN集合即往输出集合OUT中添加x->y。传输函数记为：OUT = (IN - {x->?}) U {x->y}。{x->?}表示IN集合中x指向的所有指针关系；

B. x = y。将y的指针关系复制给x，KILL集合即从IN中删除x之前的指针关系，GEN集合即往输出集合OUT中添加如下指针关系：对任何变量a，如果y->a,则x->a，相应的传输函数为：OUT = (IN - {x->?}) U {x->a | y->a ∈ IN}；

C. x = !y。将!y的指针关系赋值给x，同样需要删除所有x之前的指针关系，GEN往OUT添加如下指针关系：对任何a,b，如果y->a且a->b (!y->b)，则添加x->b，相应的传输函数为：OUT = (IN - {x->?}) U {x->b | !y->b ∈ IN}；

D. !x = y。将y的指针关系复制给!x。GEN往OUT集合添加如下指针关系：对任何a,b，如果x->a且y->b，则添加a->b。KILL集合需要分情况考察：(1) 如果x有且只指向一个变量x->a，即所谓的strong updates，那么这种情况下删除a之前的指针关系，相应传输函数为：OUT = (IN - {a->? | x->a ∈ IN}) U {a->b | {x->a, y->b} ⊆ IN}；(2) 如果x有多个指针关系，即所谓的weak updates，那么无法确定将y的指针关系复制为x指向的哪个指针变量，这种情况下只能保守认为KILL集合为空，这种情况下可能损失精度，相应传输函数为：OUT = IN U {a->b | {x->a, y->b} ⊆ IN }；

E. !x = !y。复制y指向所有变量的指针关系。GEN往OUT集合添加如下指针关系：对任何a,b,c，如果x->a，且y->b,b->c，则添加a->c。KILL同样需要分两种情况讨论：(1) 对于strong update x->a，则删除a之前的所有指针关系，这种情况下相应的传输函数为：OUT = (IN - {a->? | x->a ∈ IN}) U {a->c | {x->a, y->b, b->c} ⊆ IN}； (2) 对于weak update，即x可能指向多个变量，为不出错则不能删除任何指针关系，即KILL为空，传输函数为：OUT = IN U {a->c | {x->a, y->b, b->c} ⊆ IN }。

现在看传输函数的性质，主要考虑是否单调，以及是否distributive。首先该函数是单调的(monotone)，即对任何两个解集合A和B，如果A⊆B，则F(A)⊆F(B)，直观上可以理解，因为在上述五种情况下在考察KILL集合时都只采用的保守假设，即在weak updates的情况

下不删除任何指针关系，其他情况下，在确知指针变量被赋值时(如情况A,B,C,以及D,E中strong updates)删除指针关系是安全的。传输函数的单调性，保证了通用数据流迭代算法一定能终止，也就说能收敛到解空间格上的不动点。

其次该传输函数不满足distributive条件，即对解空间集合A和B，F(A) MEET F(B) != F(A MEET B)，原因是存在weakly updates。假设程序点A,B,C，从A和B分别有到C的路径，假设从A到C的路径上建立{x->a, y->b}的指针关系，从B到C的路径上建立{x->c, y->d}的指针关系，现在假设C上有语句：!x = y。当分别考察A->C的路径或B->C的路径时，对x的处理可以按照strong updates进行，KILL集合中分别包括{a->?}和{c->?}，GEN集合分别添加{a->b}和{c->d}；而当对两条路径上的解集合求MEET之后应用传输函数时，首先KILL集合为空，因为是weakly updates，其次GEN集合需要考虑枚举所有x和y的指针关系，因此GEN集合为：{a->b,a->d,c->d,c->b}。可见无论KILL集合还是GEN集合结果都不相同，因此不满足distributive条件，这意味着按照通用数据流框架进行迭代求MFP (Maximum Fix Point)解集合相对于MOP(Meet Over All Path)求解集合会损失精度。关于MFP和MOP的更多内容请参考数据流分析部分，或者参考Kildall的文章：”A Unified Approach to Global Program Optimization”。

指针分析的复杂度如何？回顾通用数据流分析复杂度为O(E * H * C)，其中E为控制流图边数，H为格的高度，C为传输函数复杂度。对指针分析，E为边的数目N，H为变量之间的指针关系，假设变量数目为V，则最坏情况下H为O(V*V)，C是传输函数复杂度，考虑weakly updates的情况下，需要对两个变量枚举所有可能的指针关系，从而最坏情况下复杂度也是O(V*V)，因此指针分析的复杂度为：O(N * V^4)。仅从粗略分析，即可看出指针分析相对于前面讨论的数据流分析代价更高。

至此就建立起一套基本的指针分析方法，只要将这些内容套入通用数据流分析框架，就能进行指针分析，并且获得静态分析情况下尽可能精确的结果(Maximum Fix Point)。由于MOP复杂度高且可能不收敛，因此这实质上已经是实际静态指针分析能获得最精确的结果了。由于路径敏感的指针分析复杂度仍然很高(O(N * V^4))，因此有了更实际但也更不精确的路径不敏感分析(flow insensitive analysis)，因为分析代价低很多因此在GCC/LLVM等编译器中采用，后面会谈到这类分析的原理。由于路径敏感分析精度相对较高，以及指针分析在所有程序分析方法中的基础性地位，使得这方面仍是当前一个挺重要的研究课题(即如何在保持精度的情况下提高分析效率)。

在存在指针的情况下，基本数据流分析需要依据指针关系进行微调，细节不再展开。本节最后还需要谈几个实际指针分析中的重要问题：

A. 空指针的deref问题。考虑程序点A,B,C，从A和B分别有到C的路径，从A到C有一个未初始化的指针x，从B到C有x->a，如何处理C直接deref x的问题？换言之就是从A到C的路径x的指针关系集合应该是什么内容？简单处理是，考虑最保守的情况，即让x指向所有变量的集合，但这样会让结果非常不精确。更好的处理是直接让A到C路径上的x的指针关系集合为空，从在C按照strong update来处理，实际执行路径如果从A到C，那么deref空指针会发生segmentation fault，实际分析可以利用这一点直接按x指针关系为空集处理。

B. 动态内存分配问题。指针分析必须考虑动态内存分配，如C中的malloc/free，C++中的new操作符等等，问题是如何处理动态分配的内存对象？和变量定义不同，动态分配的内存对象没有名字，因此这个问题本质上就是如何给动态分配的内存命名的问题，在指针分析中叫Heap Model。一般有三种处理方法：(1) 最保守的方式，将整个heap看做一个“变量”，只要两个指针都指向动态分配的heap空间，即认为两个指针指向相同对象；(2) 最激进的方式，进行数据流迭代时没遇到一次动态空间分配都重新命名，这样最精确，但是存在循环的情况下可能导致整个迭代不收敛；(3) 折中的方式，把所有在程序中相同语句分配的空间命相同的名字，这样仍存在不精确的问题，但相对于把整个heap处理为一个变量名要精确多了，同时也不存在无法收敛的问题。因此实际分析中最常用，由于执行heap的指针很可能执行多个区域，因此对heap指针的处理都按weakly updates进行。

C. 指针运算。指针分析假设指针运算不会超过内存中相同对象的边界，比如对结构体的指针偏移假设仍执行该结构体内部。这一假设符合C标准，但存在不遵守规范的实际程序，通过指针运算指向不同的对象，这类程序通常很难移植。实际指针分析一般都维持这一假设。

D. 数组，和指针运算类似，将整个数组视为一个对象；

E. 弱类型，对C这样的语言，允许整形和指针之间相互转换，因此任何变量都可能是指针

5.3 控制流图的化简和SSA的指针扩展

在通用数据流分析框架下优化程序分析有两种基本方法：控制流图化简和SSA。控制流图化简的出发点和SSA一样，都为了尽可能减少不必要的信息传播，两种优化都对指针分析的改进有重要作用。这一节描述控制流图化简算法，以及存在指针情况下如何构造SSA，下一节专门说指针分析的优化。

5.3.1 控制流图的化简

在引入SSA时，谈到数据流分析的缺点，即冗余信息传播。以前面提到的指针分析为例，在节点A处应用传输函数得到的输出解集合传播给A的所有后继，不管是否用到解集合的信息。这就是说，如果A有后继B，B节点没有任何指针操作，在数据流分析框架中仍然需要把指针信息从A传播到B节点，在B应用传输函数，对解集合没有改变，然后继续把信息传播后继节点。

解决这一问题的标准做法就是构造SSA，但存在指针的情况下，构造SSA的过程本身就需要指针信息，因此指针分析本身不好使用SSA(但并不是不可能，见5.4节)。在不构造SSA的情况下，另一个办法就是控制流图的化简。还以指针分析为例，只要构造一个新的控制流图，在不影响精度的情况下消去原控制流图中的和指针分析无关的节点。这种化简的控制流图叫Sparse Evaluation Graph (SEG)，或Sparse Evaluation Representations，更多内容请参考这篇文章：Ramalingom “On Sparse Evaluation Representations”，这里介绍大略。

在原始控制流图的基础上构造SEG首先需要对节点分类。把和目标分析有关的节点，即这些节点的传输函数可能会改变解集合，称为m-节点(modify nodes)；而把和目标分析无关的节点成为p-节点(preserve nodes)。以指针分析为例，涉及指针运算的节点(即可能包含五类指令：x = ref y; x = y; x = !y; !x = y; !x = !y)归为m节点，其余节点成为p节点。以reaching definitions为例，假设需要分析变量x，那么把所有对x有定义的节点成为m节点，其余节点归为p节点。

在控制流图上定义两个基本变换T2和T4。T2的定义是：假设有p节点A且A只有一个前驱，那么将A和其前驱合并为一个节点C，修改所有到A的边到新节点C，修改所有从A起始的边为从C起始。T4的定义是：假设存在一个只包含p节点的最大强连通分量(strong connected component)，那么将整个连通分量合并为单个节点D，修改所谓到连通分量的边到新节点D，修改所有从连通分量起始的边为从D起始。T2和T4两个变换很直观，是压缩控制流图构造SEG的基础。强连通分量(SCC)的构造算法有O(n)的复杂度，具体内容见前文控制流分析部分。有了T2和T4两个操作，SEG的构造算法包括三步：

A. 对原始控制流图G，设所有p节点构成的子图为Gp，用标准SCC构造算法找到Gp的所有强连通分量，按拓扑顺序记为X1, X2, ..., Xk。
B. 对所有Xi应用T4变换，设X1, X2, ..., Xk对应的压缩后的节点为w1, w2, ..., wk；
C. 顺序访问w1, w2, ..., wk，如果可以应用T2则应用T2变换，最终得到的结果就是化简后的SEG图，记为(T2+T4)*(G)，即对G应用若干次(T2+T4)化简得到的结果。

图5.2 是一个控制流图化简的例子。假设对变量x进行reaching definitions分析，节点r,c,g为m节点，其余为p节点，图(i)表示连通分量构造结束后的控制流图，即{a,d,e}, {f}, {i}, {j},

{h}分别是连通分量；图(ii)执行算法第二步，压缩连通分量，这里{a,d,e}被压缩为一个节点{w}，其余分量只有一个节点，无需处理；图(iii)到图(vi)是算法的第三步，循环应用T2直到图不能被化简位置，结果就是SEG。

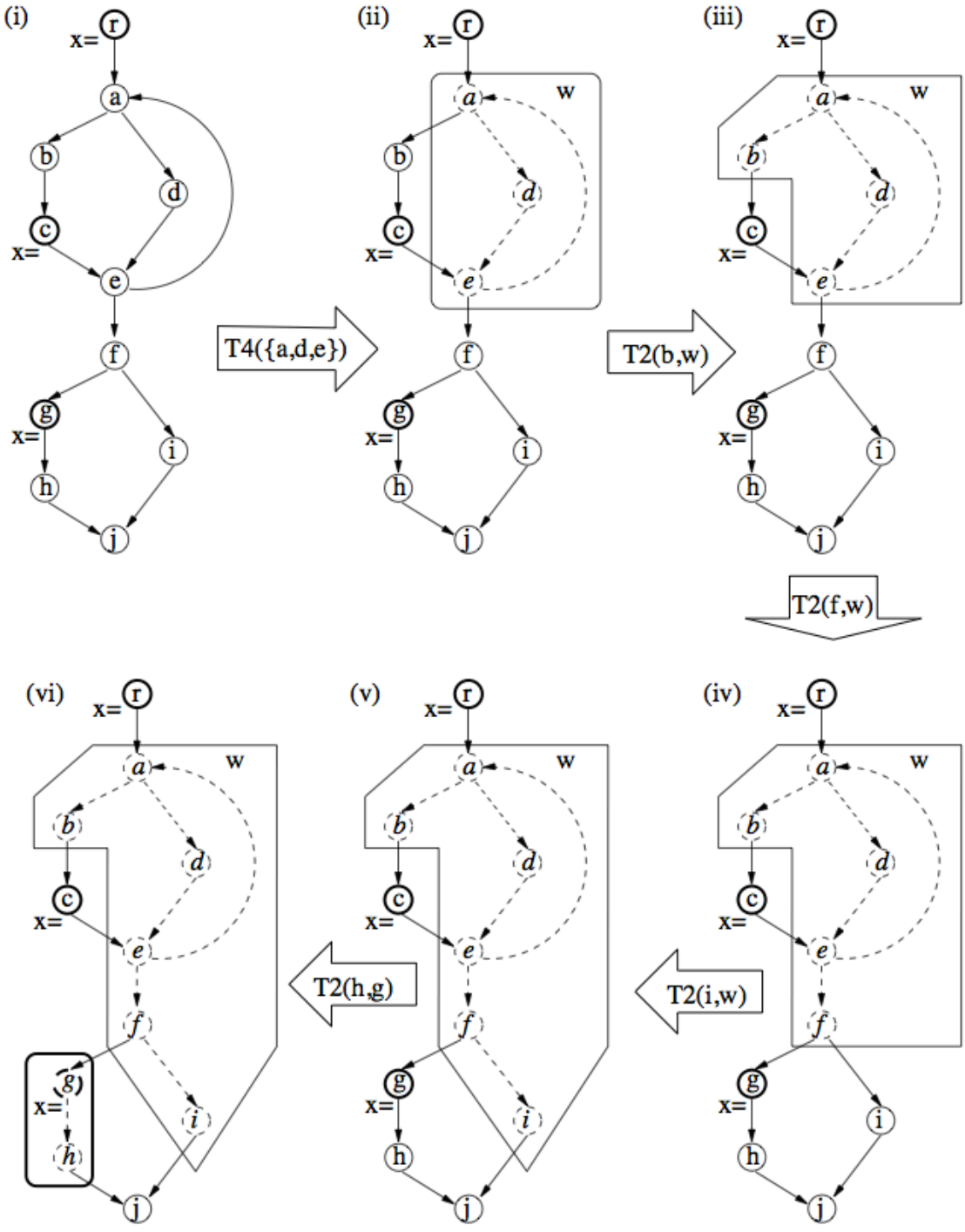

图5.2 SEG图构造举例

5.3.2 SSA的指针扩展

回答这个问题：如果指针分析结果已知，如何微调SSA以处理指针的情况？思想很直观，详细内容请参考论文：Chow, et al “Effective Representation of Aliases and Indirect Memory Operations.”。

SSA指针扩展的难点是处理对变量的间接定义和访问(!x = ? 和 ? = !x)，为此在SSA已有PHI函数的基础上添加两个函数：

A. u(x)：表示变量x可能被使用(may be used)，如果通过指针分析发现在节点A处某变量x可能被使用，则在A处添加u(x)节点；

B. x = X(x)：表示变量x可能被定义(may be defined)，如果通过指针分析发现在节点A变量x可能被定义，添加x = X(x)节点。

在引入u和X函数后，就可以用标准SSA构造算法，但存在一个严重问题：最终SSA中可能引入大量的额外def-use边，实际边的数目取决于指针分析的精度。对于涉及大量指针操作的复杂控制流图，这种方法可能导致def-use边数目爆炸，从而很大程度抵消了SSA本身的好处。另一个问题也可能限制这种方法的应用：引入u和X函数后，使得整个SSA分析变得非常复杂。

在SSA上处理指针的另一种方法是partial SSA。将所有变量分成两类：top level variables和Address Taken Variables。TLV变量不可能被指针间接操作，判断一个变量是否是TLV是看该变量是否被取地址；ATV是可能通过指针间接操作的变量。构造SSA时针对TLV和ATV分开处理，对TLV按照标准算法构造SSA，而对ATV不构造SSA。实际编译器,如LLVM和GCC，都采用这种处理方式。以LLVM为例，前端生成LLVM IR后，通过一个memory to register promotion的优化过程构造SSA，ATV变量除外，然后基于SSA做大量后端优化，第二部分会结合LLVM分析更多相关内容。

5.4 稀疏(SSA)指针分析

这里介绍Ben的两个指针分析工作，应该代表了指针分析领域的最近进展，论文见参考文献。第一个基于partial SSA做精确指针分析；第二个基于完全SSA做指针分析。这里仅提到基本原理，考虑到指针分析实际实现要比基本原理复杂得多，因此结合代码的实现分析放在第二部分。

5.4.1 基于partial SSA的指针分析

基本思想：从5.2节的内容可以看到，利用基本的数据流分析框架虽然可以用来做指针分析，但复杂度太高，解决问题的根本出发点还是让分析稀疏化。既然LLVM缺省就是一种

partial SSA，那么能否基于这种半稀疏表示方式做指针分析？工作的创新有两点：(1) 找到了在稀疏CFG上做指针分析的优化方法，大大减少了冗余信息传播。而对Address taken变量通过构造SEG化简原始控制流图。作者提出一个Dataflow Graph的概念，把TLV的SSA和ATV的SEG结合，从指针分析的角度把过去的经验进行恰当总结和运用。实验结果相对于前人工作分析效率有显著提高；(2) 数据流分析高度依赖集合运算，用一种新的数据结构(Binary Design Diagram)来表示集合和做集合操作，替代传统的位向量运算，效率有进一步提高。

结果：这个工作对路径相关的指针分析有很大推动，如表5.1所示。能看出这个工作相对于前人把分析速度提高了上百倍，把能分析的代码规模提高到几十万行代码级别。但对大程序分析速度还是很慢，从几十秒到几百秒，有些程序如(gdb, ghostscript)还是分析不了。

Name	*bitmap*						*BDD*					
	IFS		SS		SSO		IFS		SS		SSO	
	time	mem	time	mem	time	mem	time	mem	time	mem	time	mem
197.parser	80.25	888	1.28	53	0.52	15	7.24	142	0.64	142	0.48	142
ex-050325	—	OOM	15.74	198	7.33	39	7.95	142	0.66	143	0.46	142
300.twolf	72.28	415	0.82	32	0.34	12	6.41	143	0.46	144	0.32	143
255.vortex	—	OOM	33.37	1,275	11.70	81	14.39	150	0.97	151	0.78	150
sendmail-8.11.6	—	OOM	—	OOM	86.38	258	38.51	150	2.16	154	1.40	152
254.gap	—	OOM	—	OOM	191.72	518	68.66	167	2.50	168	2.34	166
253.perlbmk	—	OOM	—	OOM	—	OOM	1,477.05	280	50.22	182	21.25	177
vim-7.1	—	OOM	—	OOM	—	OOM	4,759.37	535	573.28	300	112.16	263
nethack-3.4.3	—	OOM	—	OOM	4,762.07	1,648	3,435.48	423	13.68	225	5.37	220
176.gcc	—	OOM	—	OOM	—	OOM	2,445.27	595	39.71	234	9.37	226
gdb-6.7.1	—	OOM	—	OOM	—	OOM	OOT	—	OOT	—	OOT	—
ghostscript-8.15	—	OOM	—	OOM	—	OOM	OOT	—	OOT	—	OOT	—

表 5.1 2009年路径相关指针分析的水平

5.4.2 基于完全SSA的指针分析

基本思想：基于partial SSA分析虽然能应对几十万行代码的规模，但对于ATV只能基于SEG做通用数据流分析，对于大型程序分析还是性能瓶颈。解决办法就是构造完全SSA，这里作者解决了两个技术挑战，从而拿到2011年CGO的best paper。首先构造SSA本身要指针分析，因此在进行精确分析之前先做一次不那么精确的路径无关的指针分析，然后依据该不精确的分析结果构造完全SSA，这就是说为了做精确指针分析先做一次不精确但快速的指针分析，提出了分级分析的概念，如处理器微结构中的分级分支预测。其次，因为完全SSA基于不精确的指针分析结果，因此引入大量额外def-use边，作者提出Access Equivalence概念，把def-use边按照变量分类，如果多个变量有完全相同的def-use边组成(如果从节点A到B有一条变量x的def-use边，一定存在一条变量y的def-use边)，于是可以把def-use边数目压缩。

结果：这个工作将分析规模推到百万行代码规模。结果如表5.2所示。值得指出的是，一些在partial SSA模型下无法分析出结果的大程序如ghostscript, gimp, tshark都能得到结果，但性能很低。由此可见精确指针分析的代价，由于编译器一般都很强调分析速度，因此这些算法都不用于编译器，而是一些对分析速度要求不高但追求精度的场合。

Name	SSO		SFS					Time	Mem
	Time	Mem	Prelim	Prep	Solve	Total Time	Mem	Comp	Comp
197.parser	0.41	138	0.29	0.07	0.008	0.37	275	1.11	0.50
300.twolf	0.23	140	0.34	0.07	0.004	0.41	281	0.56	0.50
ex	0.35	141	0.29	0.10	0.008	0.40	277	0.88	0.51
255.vortex	0.60	144	0.45	0.14	0.028	0.62	285	0.97	0.51
254.gap	1.28	155	0.94	0.33	0.016	1.29	307	0.99	0.50
sendmail	1.21	147	0.70	0.27	0.032	1.00	301	1.21	0.49
253.perlbmk	2.30	158	1.05	0.50	0.020	1.57	312	1.46	0.51
nethack	3.16	197	1.72	0.82	0.096	2.64	349	1.20	0.56
python	120.16	346	4.04	2.02	0.564	6.62	404	18.15	0.86
175.gcc	3.74	189	2.00	1.42	0.040	3.46	370	1.08	0.51
vim	61.85	238	2.93	2.44	0.160	5.53	436	11.18	0.55
pine	347.53	640	13.42	21.25	47.330	82.00	876	4.24	0.73
svn	185.10	233	5.40	5.07	0.216	10.69	418	17.32	0.56
ghostscript	OOT	—	42.98	86.13	1787.184	1916.29	2359	∞	0
gimp	OOT	—	90.59	105.87	1025.824	1222.28	3273	∞	0
tshark	OOT	—	232.54	219.83	376.096	828.47	6378	∞	0

表5.2 2011年路径相关指针分析的水平

6 过程间分析

6.1 过程间控制流图

实际程序分析都需要处理过程调用，因此第一个问题是如何对控制流图进行合理扩展以处理函数调用和返回。解决方法之一是函数调用图(Call Graph)，定义如下：

给定程序P，其调用图为G(N,E)，其中N为函数集合，E ⊆ NxN为边集合，图G中存在边x->y 当且仅当函数x调用函数y。

Program

```
def id = fun (p) { q := p ; return q }

x := 1      ;
y := 2      ;
a := id(x) ;
b := id(y)
```

图 6.1 函数调用举例

调用图是对函数调用的一种粗略表示，比如无法从调用图得知一个函数调用了多少个函数，以及这些函数调用具体发生的位置。调用图可能有环，表示递归调用。对于图6.1的LIngo程序，其调用图节点集合为{main, id}，这里假设main为顶层函数，边集合为{main->id}，也就是说只有一条边，因为main只调用了id函数。由此可见单纯调用图只能给出有限的信息。

更精确处理过程间调用的方法是过程间控制流图(Interprocedural CFG, ICFG)。ICFG以过程内CFG为基础，针对过程调用增加如下四类边：

A. 将程序中Call节点分离为两个独立节点：Call node and return node

B. 增加从主调函数(Call node)到被调函数(Callee)入口的边；

C. 增加从被调函数(Callee)出口返回到主调函数(Ret node)的边；

D. 在分离的Call节点的Call node和Ret node之间构造一条虚拟边。

图6.1的过程间控制流图如图6.2所示，图中每个节点表示一个程序语句。

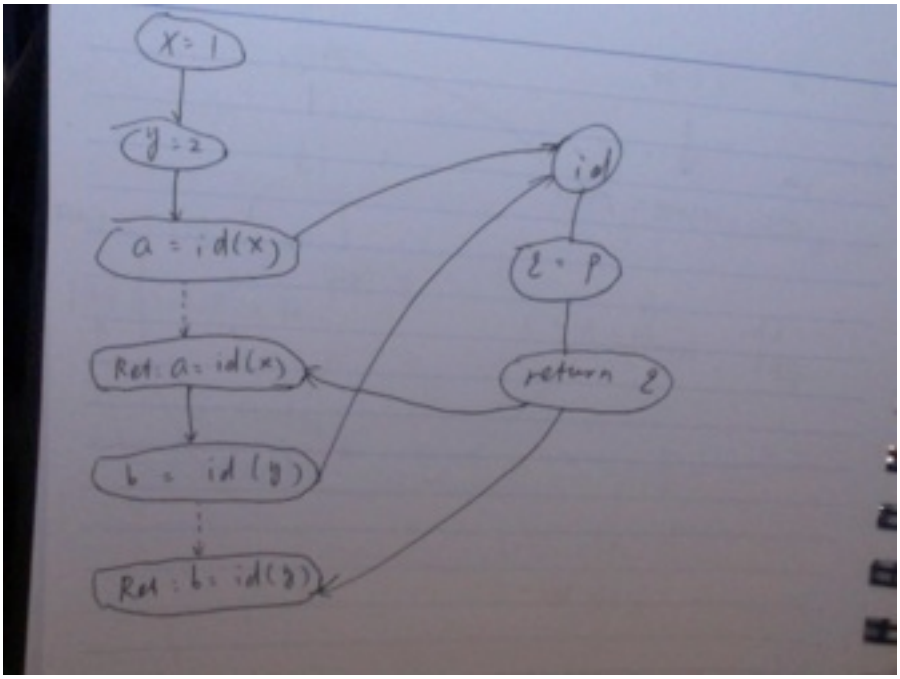

图6.2 ICFG举例

ICFG是对程序实际可能执行路径的保守表示，和CFG不同，ICFG一些路径组合在实际程序执行路径中不会出现，如图6.2中存在一条a=id(x)->id->return q->Ret: b = id(y)的路径，这条路径在程序实际执行时不可能出现。这就引入上下文敏感的问题，程序分析如果能利用这种函数调用的上下文信息，则结果更精确。

6.2 上下文敏感分析

6.2.1 为什么需要上下文敏感分析

在过程间程序分析中，依据是否利用函数调用上下文可以分为context sensitive和context insensitive两类。上下文不敏感的分析相对简单，即将ICFG看做一个整体控制流图，然后应用数据流分析框架来迭代，这里Call/Ret退化为goto语句。这种方法本质上相对于把所有函数的CFG合成了一个大CFG，分析方法和过程内分析完全一样，优点是易于实现，缺点是忽略了上下文信息，可能大大损失分析精度。

仍以图6.2的ICFG为例，考虑过程间常量传播问题。在第一次调用函数id返回时，q的结果传播给所有调用点({a = id(x); b = id(y);})，从而{a = 1, b = 1}。同理在第二次函数调用时，返回值也传播给两个调用点，对两次迭代的结果求MEET最后的结果是{a = Bottom, b = Bottom}。事实上，直观上很容易看出常量传播的结果应为{a = 1, b = 2}，这就需要上下文敏感分析来防止从函数出口到调用点的冗余传播。

6.2.2 Call String和K-limiting算法

上下文敏感分析有两种基本类型：以控制流为基础的Call String和以数据流为基础的Functional context sensitivity。先看Call String。

直观上Call String用来模拟程序调用栈，假设在分析函数A时的Call String为CS1, 如果在某个点调用函数B，这样分析函数B时Call String为{CS1 + “B”}，等B分析完返回调用点时修正Call String为CS1继续分析A中的后续代码。这样对每个函数的分析结果都取决于当次分析的函数调用上下文(Call String)，函数出口返回时只返回结果为Call String一致的调用节点。

可以通过修正ICFG的方法来实现Call String同样的效果，即invocation graph (IG)。修正方法是：(1) 在ICFG每次函数调用的位置，复制一份被调函数的控制流图；(2) 在每次被调函数返回的位置，只构造到上下文对应的调用点的返回边。可以看出，在构造invocation graph之后，对其做context insensitive分析的结果和对ICFG做Call String context sensitive分析的结果相同。也就是说通过invocation graph将一个上下文敏感分析的问题转化为上下文不敏感的分析。

Call String和Invocation Graph存在的问题是递归。解决办法两个：(1) 当存在递归调用时，ICFG的调用节点和返回节点构成环路，因此只要用Tarjan算法找出所有连通分量(SCC)，SCC内部的函数采用上下文不敏感分析，其余函数采用上下文敏感分析。(2) K-limiting，即最多保存K次函数调用的上下文信息，也就是将长度限制为K的调用栈。接下来详述K-limiting。

设cs表示代表函数调用上下文的Call String，cs是一个LIFO的栈操作字符串，K-limiting需要四个辅助操作：

A. push(cs, n)：将n入栈，返回入栈后的新Call String，即n+cs；

B. head(cs)：返回栈顶元素，即当前正在分析的被调函数；

C. tail(cs)：返回除栈顶之外剩余调用上下文(cs - head(cs))，代表进入当前正在分析的被调函数之前的调用历史；

D. klim(cs)：返回调用栈cs前k个元素。cs代表全部的函数调用历史，klim(cs)表示cs的前k个元素。

对数据流分析框架做如下调整：

A. 工作队列(worklist)的每个任务表示为二元组(n, cs)，其中n表示程序点，实际操作可处理为和分析相关的语句，cs表示调用上下文。worklist初始化为整个程序的入口；

B. 传输函数的IN/OUT集合扩展为每个上下文一组解集合。

整个算法工作过程就是从工作队列中提取待处理节点，针对节点类型分类处理直到队列为空为止。一共包括五种情况：

A. Entry Nodes，即每个函数的入口节点。处理分三步：

    A. n.IN[klim(cs)] <- head(cs).IN[klim(tail(cs))]，即将调用节点的OUT集合传播给被调函数入口。由于调用节点本身不修改集合内容，因此传播其IN集合内容。对于被调函数来说，tail(cs)表示调用函数上下文，head(cs)表示主调函数，二者合起来能得到调用节点处的解集合；

    B. 调整n.IN[klim(cs)]，处理参数传递，根据调用节点传递的参数，修正入口节点的IN集合；

    C. 计算入口节点的OUT集合n.OUT[klim(cs)]。

B. Exit Nodes，每个函数的出口。每个函数出口节点维护一个调用上下文集合callstrings，每个元素保存完整的Call String，而不是klimit(cs)，处理分两步：

    A. 计算n.IN[klim(cs)]，对n的每个前驱p.OUT[klim(cs)]，将klim(cs)上下文相关的OUT集合求MEET，和基本数据流分析处理过程相同；

    B. 考察n.callstrings集合，将OUT集合传播给所有klimit(s) = klimit(cs)的调用节点。对于上下文不敏感分析，这里相对于k为0，因此将解集合传播给所有调用节点；对于完全的上下文敏感分析，相对于k为无穷大，因此只将结果传播给和上下文对应的唯一调用节点。

C. Call nodes，函数调用节点n。设Fentry和Fexit分别表示被调函数的入口和出口节点，设m为和当前调用节点对应的Ret节点(每个调用节点对应一个Ret节点，见前面ICFG定义)。操作分三步：

A. 计算n.IN[klim(cs)]，对n的每个前驱p.OUT[klim(cs)]，将klim(cs)上下文相关的OUT集合求MEET；

B. 在工作队列添加(Fentry, push(cs,n))和(m, cs)两个待处理节点；

C. 往Fexit的callstrings集合中添加push(cs,n)：Fexit.callstrings <- push(cs,n)。

D. Ret Nodes，ICFG中和调用节点对应的Ret节点。设Fexit为被调函数的出口节点，m为和该Ret节点对应的调用节点，处理分三步：

A. n.IN[klim(cs)] <- Fexit.IN[klim(push(cs,m))]，此处实现被调函数出口解集合到调用节点的传播；

B. n.OUT[klim(cs)] <- n.IN[klim(cs)]，同时对于有返回值的调用，调整n.OUT[klim(cs)]以处理被调函数的返回值。

E. 其他节点。在klim(cs)上下文中和数据流分析处理过程相同。

6.2.3 Functional Context Sensitivity

和Call String主要关注控制流不同，这类上下文敏感分析从数据流的角度定义上下文的相关性。任何程序分析都将程序的实际运行行为映射到抽象域，然后在抽象域不同应用传输函数进行工作队列迭代，知道整个解空间到达不动点。这样从程序分析的角度，一个函数调用可以抽象为根据一组抽象的输入状态应用传输函数得到其输出状态。Functional上下文敏感分析即保存函数每次调用的输入和输出状态集合，之后每次分析该函数之前比较输入状态，如果之前已经分析过就直接得到应用传输函数后的输出状态。Functional的意义即把一次函数调用理解为从输入到输出的映射，输入相同则输出相同，这样对函数调用维持一个输入到输出的映射表，本质上是希望在多次分析过程中重用分析结果。

相对于Call String，在函数输入抽象状态有限的情况下，Functional上下文分析可以自动处理递归过程，上下文展开是在分析过程中动态展开完成的。其复杂度主要取决于函数调用的输入/输出状态空间的大小。相对于Call String，Functional上下文敏感分析一般更为精确，但实现更复杂。注意这两种上下文相关分析方法其实是正交的，可以合起来使用以达到更高的精度。

6.3 过程间指针分析

图6.3是对Lingo进行完成函数和指针扩展后的结果。对程序分析来说，函数和指针并不是相互独立的语言特性。指针对过程间分析的影响体现在：(1) 间接过程调用，即函数指针，对指针分析精度有很大影响；(2) 调用参数和返回值存在指针情况下的处理(即函数调用时的forward binding，以及函数返回时backward binding)。

函数对指针分析的影响体现在：(1) 上下文敏感的heap模型。 在指针分析部分曾提到实际实现一般采用的heap模型是：对程序中相同语句分配的动态内存指针统一采用相同的名字， 引入过程间分析后， 可以在考虑函数调用上下文的情况下对动态分配的指针命名，从而可以进一步提高精度；(2) 递归调用时的strong updates问题。回顾前面的指针分析的内容， 对于*x = y语句的处理， 当x只指向唯一的变量(即strong updates)可以从解集合中去掉一些不必要的指针关系从而能提高分析精度。但在存在递归调用的情况下， 即使x只指向唯一的变量也不能按strong updates处理， 因为递归多次函数调用的局部变量地址其实位于堆栈中的不同位置。

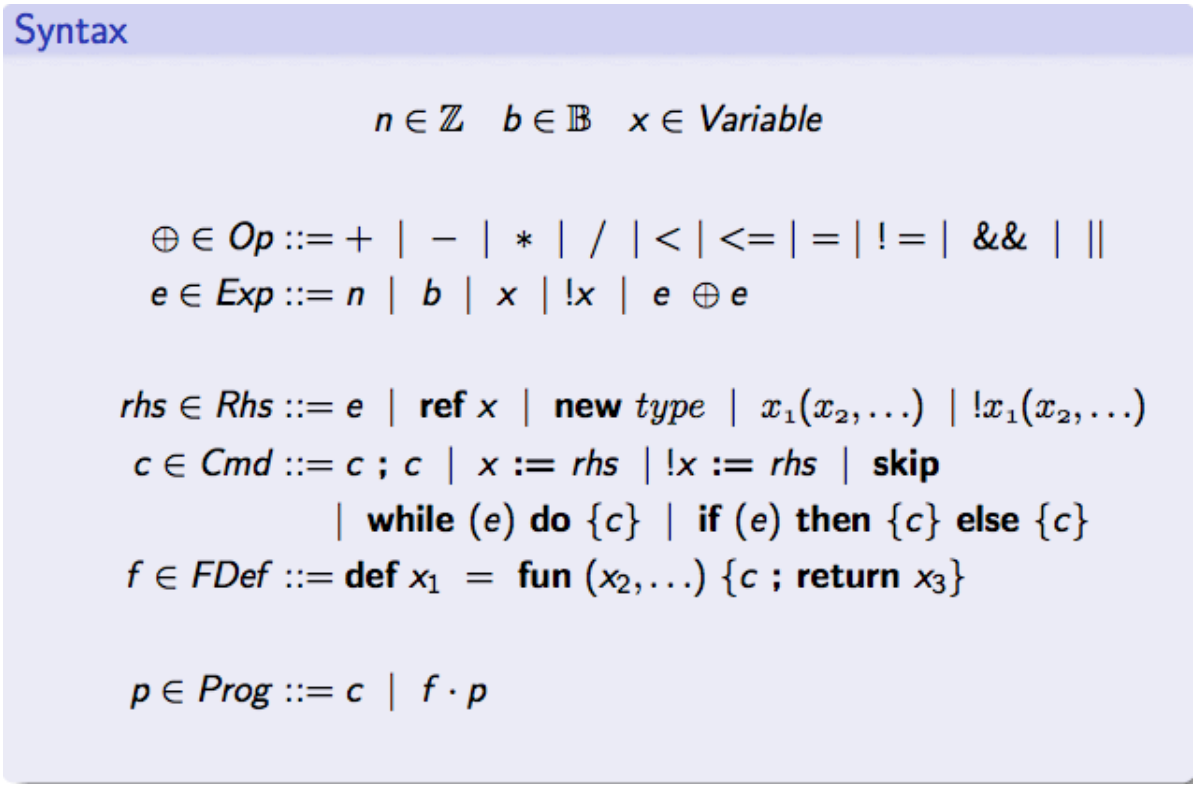

图6.3 完整的Lingo语言

7约束分析

7.1 集合约束

约束分析(constraint based analysis)的基础是约束描述语言(constraint language)， 通过约束语言精确描述程序分析解空间必须满足的约束条件， 然后通过不断化简和推导， 直到得到解集合。约束描述语言有很多种， 这里使用集合约束， 其定义如图7.1所示。

集合约束以集合运算为基础，包括集合变量(variable)、构造函数、表达式和语句等几个元素。集合变量x表示一个集合，其中0表示空集，1表示全集。集合表达式可以是变量、0、1、constructor或者表达式之间的非、取并集和交集等运算。单个集合语句表达集合表达式之间的包含关系，复合集合语句描述多个语句的联合约束。

Definition (Set Constraint Language)

$$x \in Variable \quad c \in Constructor \quad \mathbf{0} = \text{empty set} \quad \mathbf{1} = \text{universal set}$$

$$E \in Expr ::= x \mid \mathbf{0} \mid \mathbf{1} \mid \neg E \mid E_1 \cup E_2 \mid E_1 \cap E_2 \mid c(E_1, \ldots, E_{a(c)})$$
$$S \in Stmt ::= E_1 \subseteq E_2 \mid S_1 \wedge S_2$$

图7.1 集合约束语言

图7.2是一组集合语句的例子，c1~c3表示集合构造函数，因为没有参数，故省去括号。构造函数的目的是根据已有集合变量构造新集合，后面会有更多说明。注意构造函数、集合表达式的基本操作(非、求并集交集)，以及集合表达式之间的包含操作等都需要依据要解决的目标问题具体定义其语义。

Example (Set Constraint Statement)

$$\begin{aligned} \{c_1\} &\subseteq x & \wedge \\ \{c_1, c_2, c_3\} &\subseteq y \cup x & \wedge \\ x \cap y &\subseteq \{c_2\} & \wedge \\ z &\subseteq \neg(x \cap y) \end{aligned}$$

图7.2 集合约束举例

构造函数的一个很好例子是函数语言里的类型构造。基本数据类型，如Int, String，本身是构造函数，无需参数；数组类型，如int[]，接受一个类型参数，返回一个该类型的数组类型；二元构造函数，比如Int->String，有两个类型参数Int和String，该类型构造函数将Int类型的参数映射为String类型。

构造函数相对于每个参数要么是单调递增(monotone)或递减(anti-monotone)，也就是说如果把构造函数给每个参数求“偏导”，结果要么>=0或者<=0。如果一个构造函数相对某个参数来说是递增的，那么就称该参数为covariant，否则就称该参数为contravariant。举例来说，考察将类型X映射为类型Y的构造函数lam(**X**,Y): {f | x∈X ⇒ f(x) ∈ Y}。可见lam函

数是一个函数集合，每个函数将X集合中的元素映射到Y集合。本文在写构造函数时，将contravariant参数以加粗和下划线的方式突出表示，如lam函数中的X。

以lam参数为例，由于X参数是contravariant，那么就是说增大集合X回使得满足条件的函数集合变小，即X ⊆ X1 ⇒ lam(**X**,Y) ⊇ lam(**X1**,Y)，同理，由于Y参数是covariant，那么增加集合Y使得满足条件的函数集合变大，即Y ⊆ Y1 ⇒ lam(**X**,Y) ⊆ lam(X,**Y1**)。

具体来说，假设X={1,2}，X1 = {1,2,3}，Y={1}, Y1 = {1,2,3},

函数f1: f1(1) = 1, f1(2)=1, f1(3) = 3

函数f2: f2(1) = 1, f2(2)=1, f2(3) = 1

函数f3: f3(1) = 1, f3(2)=2, f3(3) = 3

lam(**X**,Y)的解{f1, f2}，lam(**X1**,Y)的解{f2}，lam(X,**Y1**)的解{f1,f2,f3}。

7.2 基于约束的程序分析

基于约束的程序分析包括两步：

A. 约束的生成(constraint generation)，遍历程序生成一条集合约束语句。一条集合约束语句是一组集合约束的集合：S: S1 ^ S2 ^ S3 ... ^ Sn，每条语句表示一个集合约束表达式：E: E1 ⊆ E2。

B. 约束求解(constraint resolution)，给定一个集合约束语句，找到满足所有集合约束的解集合。

以reaching definitions为例来说明，首先定义基本集合变量即构造函数，在约束分析中，基本集合变量及构造函数的定义称为Universe of Discourse，对RD来说，定义如下：

A. 对每个程序点k，定义集合变量INk和OUTk；

B. 每个程序点k，定义无参数构造函数k；

C. 对每个变量x，定义无参数构造函数x；

D. 二元构造函数def(x,k)，表示变量x在程序点k被定义

RD的约束生成如图7.3所示。每种语句对应一条规则，⇒左边部分表示语句，右边表示语句对应的约束规则。说明如下：

A. 对程序点k的赋值x = e，从INk中减去所有对变量x的定义，然后把剩下的元素传播(⊆)到OUTk中，同时需要将def(x,k)表示的集合添加到OUTk中；

B. 对程序点k的if(-)else语句，首先递归生成两个分支c1和c2的约束规则，将INk传播给INk1和INk2，将OUTk1_和OUTk2_传播给OUTk。k1(k2)表示c1(c2)中的第一条语句，k1_(k2_)表示c1(c2)中最后一条语句。

C. 对程序点k的while语句，和if/else类似，先递归生成循环体c的约束规则，将INk传播给INk1，将OUTk2传播给OUTk

D. 程序点k1和k2之间顺序语句的解集合传播，顺序生成c1和c2的约束规则，将k1的输出集合OUTk1传播给k2的输入集合。

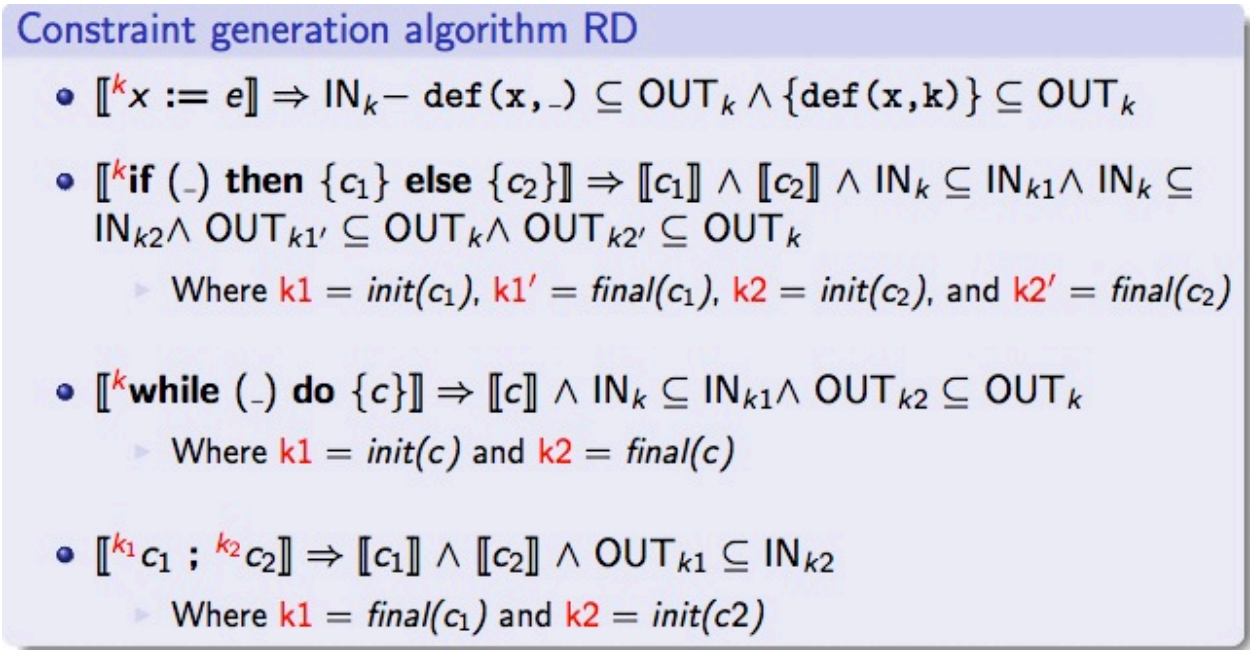

图7.3 Reaching Definitions的约束规则

剩下的问题的是约束求解，下一节以指针分析为例展开具体的求解过程。这里仅以RD为例说明约束构造过程。

7.3 Andersen-style指针分析

7.3.1 基本原理

Andersen-style指针分析是流不敏感(flow insensitive)指针分析家族中的重要成员。之前讨论过的指针分析都是流敏感(flow sensitive)分析，对于大型程序效率很低，因此一般只用于特定的程序分析场合，不在编译器中使用。和流敏感指针分析中求出每个程序点变量的指针关系(point-to sets)不同，流不敏感指针分析针对整个程序对每个变量找到一个最精确的指针关系集合。因为每个变量的指针集合表示的是整个程序所有可能路径的保守结果，因此流不敏感指针分析一定不如流敏感分析精确，但一般而言效率要高得多，因为这一原因，GCC/LLVM都采用流不敏感的指针分析来辅助程序优化。

Andersen-style指针分析基于简化的集合约束语言，定义如图7.4所示。相对前面的集合约束语言，主要简化集合表达式，去掉了集合操作(非、集合交集和并集)。对于

Andersen-style指针分析，所有约束规则都表示为集合包含关系，因此这类指针分析也称为inclusion-based指针分析。

Definition (Restricted Set Constraint Language)

$$x \in Variable \quad c \in Constructor \quad \mathbf{0} = \text{empty set} \quad \mathbf{1} = \text{universal set}$$

$$E \in Expr ::= x \mid \mathbf{0} \mid \mathbf{1} \mid c(E_1, \ldots, E_{a(c)})$$
$$S \in Stmt ::= E_1 \subseteq E_2 \mid S_1 \wedge S_2$$

图7.4 简化集合约束语言

对Andersen-style指针分析而言，约束规则的求解过程基于一组约束化简规则(ReWrite Rules)，如图7.5所示。对每条化简规则说明如下：

A. 由于x⊆x，因此S ∧ x ⊆ x ⇒ S；

B. 由于E ⊆ I，因此S ∧ E⊆I ⇒ S；

C. 由于0 ⊆ E，因此S ∧ 0 ⊆ E ⇒ S；

D. 构造函数的化简，即如果有c(E1, ..., Ek) ⊆ c(E1', ..., Ek')，则依据构造函数每个参数的单调性展开，如第i个参数是covariant，则Ei⊆Ei'，否则Ei' ⊆ Ei；

E. 集合关系的传递，即如果c(E1, ..., Ek)⊆x且x⊆y成立(S中包含此约束)，则c(E1,...,Ek)⊆y成立。

Rewrite Rules

$$S \wedge x \subseteq x \rightarrow S \qquad S \wedge E \subseteq \mathbf{1} \rightarrow S \qquad S \wedge \mathbf{0} \subseteq E \rightarrow S$$

$$c(E_1, \ldots, E_k) \subseteq c(E_1', \ldots, E_k') \rightarrow \bigwedge_{i \in [0..k]} \begin{cases} E_i \subseteq E_i' & i \text{ is covariant} \\ E_i' \subseteq E_i & i \text{ is contravariant} \end{cases}$$

$$c(E_1, \ldots, E_k) \subseteq x \wedge x \subseteq y \in S \rightarrow S \wedge c(E_1, \ldots, E_k) \subseteq y$$

图7.5 约束化简规则

有了化简规则就可以定义Andersen-style指针分析的约束规则，并进行求解。先定义初始集合变量和构造函数(Universe of Discourse)：

A. 每个变量定义一个无参数构造函数x，即用此构造函数x来表示变量x；

B. 给定变量x，用X表示x可能指向的变量集合，即x的points-to集合；

C. 定义一个三元构造函数ref(x,X,**X**)，定义此构造函数将变量x和其可能指向的变量集合X关联起来，构造函数第二个参数是covariant参数，即输入集合IN，第三个参数是contravariant参数，用来构造输出集合OUT。直观上，该构造函数表达的意义是：从covariant参数X获取输入指针信息，将结果存入contravariant参数指向的指针关系集合。

接下来给每种程序语句定义约束规则(constraint generation)，如图7.6所示，依次说明如下：

A. 对于顺序语句c1; c2，则依次对c1和c2构造约束规则；

B. 对于if/else语句，依次对两个分支c1和c2构造约束规则，可见这里的处理方式和顺序语句完全一样，前面已指出Andersen-style指针分析是流不敏感分析，给整个程序求解一个全局的保守解集合；

C. while语句，对循环体c构造约束规则；

D. 赋值语句包含两条规则，首先[[e2]]⊆ref(1,T,0)表示将e2的指针信息集合保存到临时变量T(获取e2的指针信息到IN集合)中，[[e1]]⊆ref(1,1,T)从T中提取指针信息集合合并为e1的输出集合(将e2的指针信息合并到e1的输出集合OUT)。

E. 给定变量x，用ref(x,X,**X**)表示x指向的变量集合；

F. 给定取地址表达式ref x，同理，用ref(0,[[x]],**[[x]]**)表示!x集合指向的变量集合；

G. 给定deref表达式!x，将x指向的变量的指针信息集合保存到临时变量T中。

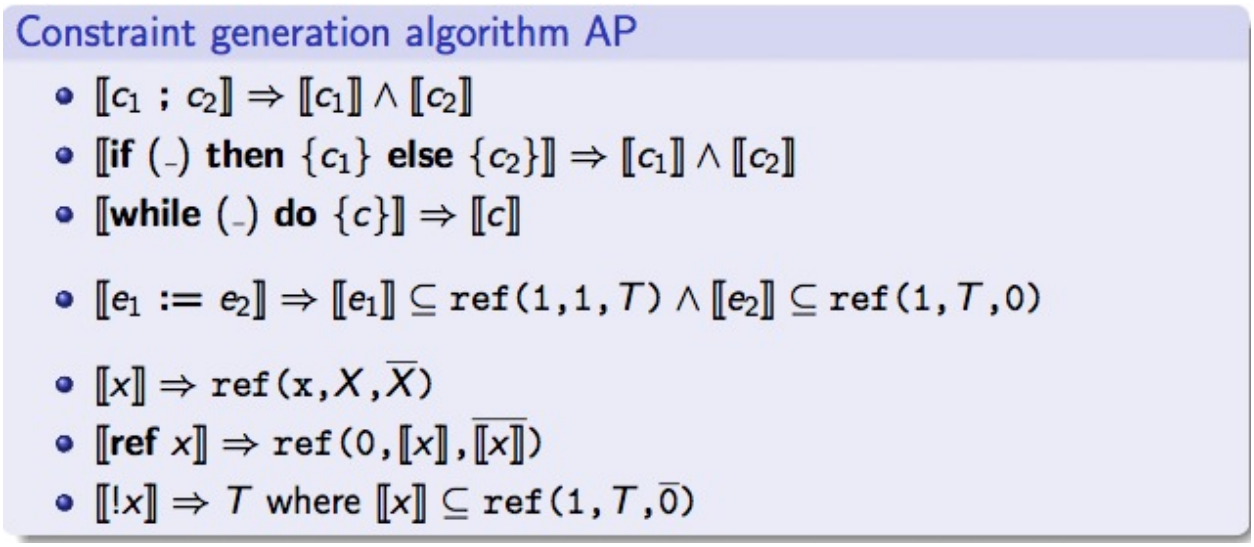

图7.6 Andersen-style指针分析的约束规则

指针分析的过程，首先扫描程序生成集合约束规则，然后不断应用规则化简直到得到所有的指针关系，下面举两个例子说明这一约束求解过程。

例子1，对如下程序段做指针分析：

x = ref y;
a = x;
x = ref b;
z = ref w;
!x = z;

第一步，对每条语句构造集合约束，内容如下：

A. x = ref y⇒ref(x,X,**X**)⊆ref(1,1,**T1**) ∧ ref(0,[[y]],**[[y]]**)⊆ref(1,T1,**0**)，由于[[y]] = ref(0,Y,**Y**)，因此约束规则为：ref(x,X,**X**)⊆ref(1,1,**T1**) ∧ ref(0,ref(y,Y,**Y**),**ref(y,Y,Y)**)⊆ref(1,T1,**0**)

B. a = x ⇒ ref(a,A,**A**)⊆ref(1,1,T2) ∧ ref(x,X,**X**)⊆ref(1,T2,0)

C. x = ref b ⇒ ref(x,X,**X**)⊆ref(1,1,**T3**)∧ref(0,ref(b,B,**B**),**ref(b,B,B)**)⊆ref(1,T3,**0**)

D. z = ref w ⇒ ref(z,Z,**Z**)⊆ref(1,1,**T4**)∧ref(0,ref(w,W,**W**),**ref(w,W,W)**) ⊆ref(1,T4,**0**)

E. !x = z ⇒ ref(x,X,**X**)⊆ref(1,T5,**0**) ∧ T5⊆ref(1,1,**T6**)∧ref(z,Z,**Z**)⊆ref(1,T6,**0**)，注意这里包含两条规则，deref和赋值，尤其注意deref(!x)的写法：T5⊆ref(1,1,**T6**)。

第二步规则化简：

A. 先化简ref(x,X,**X**)⊆ref(1,1,**T1**)，根据构造函数化简规则有x⊆1∧X⊆1∧T1⊆X⇒T1⊆X，同理 ref(0,ref(y,Y,**Y**),**ref(y,Y,Y)**)⊆ref(1,T1,**0**)⇒0⊆1∧ ref(y,Y,**Y**)⊆T1∧0⊆ ref(y,Y,**Y**)⇒ref(y,Y,**Y**)⊆T1，根据传递规则有ref(y,Y,**Y**)⊆T1∧T1⊆X，即ref(y,Y,**Y**)⊆X，这意味着x指向的变量集合中包括变量y；

B. 化简后有T2⊆A∧X⊆T2，应用传递规则得到X⊆A，意味着a指向所有x指向的变量；

C. 和A的情况相同，化简后有 ref(b,B,**B**)⊆T3∧T3⊆X，应用传递规则得ref(b,B,**B**)⊆X，意味着x指向的变量集合中包括b；

D. 同理，化简后有 ref(w,W,**W**)⊆T4∧T4⊆Z，应用传递规则ref(w,W,**W**)⊆Z，即将w添加到z指向的变量集合；

E. 根据赋值规则，有Z⊆T6∧X⊆T5∧T5⊆ref(1,1,T6)，因此Z⊆T6∧X⊆ref(1,1,T6)，因ref(w,W,W)⊆Z，即将w添加到T5集合中所有变量的指针集合中。由于 ref(y,Y,**Y**)⊆X⊆T5，ref(b,B,**B**)⊆X⊆T5，因此ref(z,Z,**Z**)⊆Y，ref(z,Z,**Z**)⊆B。

由于ref(y,Y,Y)⊆X, X⊆A，因此ref(y,Y,Y)⊆A，同理ref(b,B,B)⊆A。至此得到的所有的指针关系如下：x->{y,b}, a->{y,b}, z->{w}, y->{w}, b->{w}

下面再举一个例子，说明约束求解过程：

例子2：

y = ref a;
b = ref a;
a = ref b;

```
while(_) do {
  x = y;
  y = !x;
  !x = ref z;
}
```

第一步给每条语句构造约束规则，由于是流不敏感分析，故无需考虑控制流：

A. y = ref a ⇒ ref(a,A,**A**)⊆Y

B. b = ref a ⇒ ref(a,A,**A**)⊆B

C. a = ref b ⇒ ref(b,B,**B**)⊆A

D. x = y ⇒ Y ⊆ X

E. y = !x ⇒ T1 ⊆ Y∧X⊆T2∧T2⊆ref(1,T1,0), T1表示x指向的变量集合

F. !x = ref z ⇒ref(x,X,**X**)⊆ref(1,T3,**0**) ∧ T3⊆ref(1,1,**T4**) ∧ ref(0,ref(z,Z,Z),**ref(z,Z,Z)**)⊆ref(1,T4,**0**),应用化简规则后得X⊆T3 ∧T3⊆ref(1,1,T4) ∧ref(z,Z,Z)⊆T4

上述规则已经是进行初步化简后的结果，但还需要应用传递规则得到最终结果。传递过程如下：

第一轮：

ref(a,A,A)⊆Y,
ref(a,A,A)⊆B,
ref(b,B,B)⊆A,
ref(a,A,A)⊆Y∧Y⊆X⇒ref(a,A,A)⊆X,

ref(b,B,B) ⊆ A ∧ ref(a,A,A)⊆X ∧ T1 ⊆ Y ⇒ ref(b,B,B) ⊆ Y,

ref(b,B,B) ⊆ A ∧ ref(a,A,A) ⊆X ∧ ref(z,Z,Z) ⊆T2 ⇒ ref(z,Z,Z) ⊆A

第二轮：

ref(a,A,A) ⊆Y, ref(b,B,B) ⊆ Y
ref(a,A,A) ⊆ B
ref(b,B,B) ⊆A, ref(z,Z,Z) ⊆A
{ref(a,A,A), ref(b,B,B) } ⊆Y ∧ Y ⊆ X ⇒ {ref(a,A,A), ref(b,B,B)} ⊆X

{ref(b,B,B), ref(z,Z,Z)} ⊆ A ∧ ref(a,A,A) ⊆B ∧ {ref(a,A,A), ref(b,B,B)} ⊆X ∧ T1 ⊆ Y ⇒ {ref(a,A,A), ref(b,B,B), ref(z,Z,Z)} ⊆ Y
{ref(a,A,A), ref(b,B,B)} ⊆X ∧ ref(z,Z,Z)⊆T2 ⇒ ref(z,Z,Z) ⊆A, ref(z,Z,Z) ⊆B

第三轮：

{ref(a,A,A), ref(b,B,B), ref(z,Z,Z)} ⊆ Y
{ref(a,A,A), ref(z,Z,Z) } ⊆ B
{ref(b,B,B), ref(z,Z,Z) } ⊆ A
{ref(a,A,A), ref(b,B,B), ref(z,Z,Z)} ⊆ X
ref(z,Z,Z) ⊆ X ∧ ref(z,Z,Z) ⊆ T2 ⇒ ref(z,Z,Z) ⊆ Z

因此得到的最后结果：

y -> {a,b,z}
b -> {a,z}
a -> {b,z}
x -> {a,b,z}
z -> {z}

7.3.2 实现算法

前面的求解过程虽然可操作，但实际实现时过于繁琐，首先将约束生成过程简化，如图7.7所示。其实在前一节例子2中已经很大程度上采用了这种简化规则，如将赋值语句直接生成一条包含规则，将[[ref x]]处理为{x}，将[[x]]处理为x指向的变量集合X，注意[[!x]]的处理，[X]表示间接指针关系，即X中变量指向的变量集合。通过这种简化，程序语句对应的约束规则只有三种基本形式：

A. x = ref y ⇒ {y} ⊆ X

B. x = y ⇒ Y ⊆ X

C. x = !y ⇒ [Y] ⊆ X或!x = y ⇒ Y ⊆ [X]

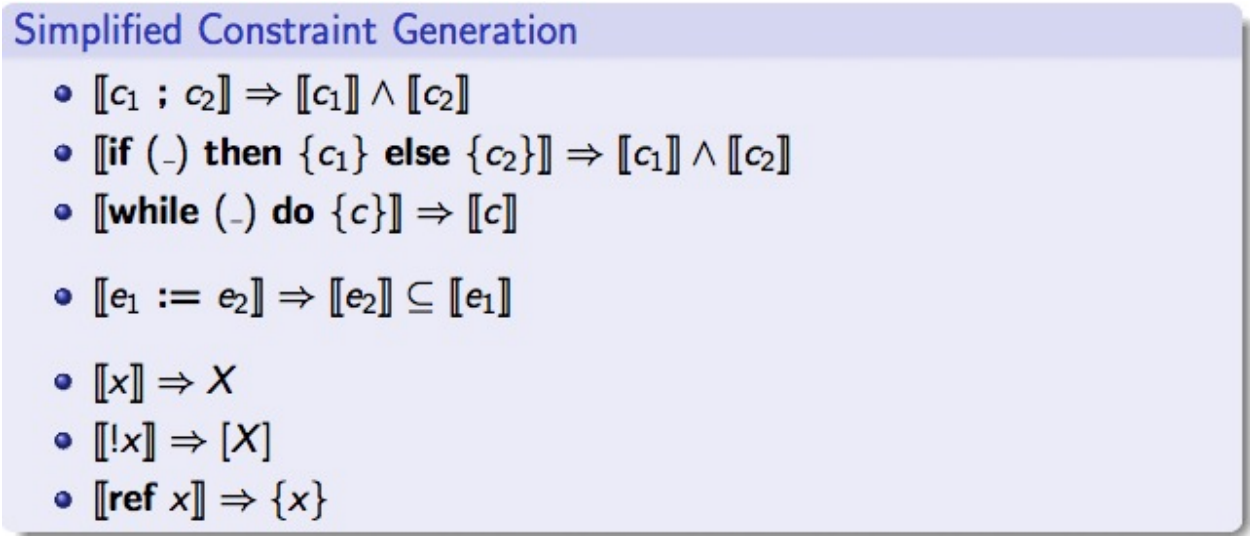

图7.7 简化的约束生成

图7.7的化简规则可以进一步抽象为约束图G=(N,E)，其中N表示变量集合，图中每个节点(变量)关联一个该变量指向的变量集合，E表示边集合，节点X和Y之间存在边X->Y当且仅当X⊆Y。约束图中的边表示变量之间的指针约束关系，图的初始状态包含初始约束，即x = ref y和x = y两种情况。

将约束规则抽象为图之后，就可以在节点间传递指针信息来完成整个指针分析过程。首先扫描程序，为每个变量构造一个节点，然后初始化基本指针信息，对于x = ref y，则将y

添加到节点x的指针集合中，对于x = y，则添加一条从y到x的边。接下来对图中所有约束没有满足的边进行约束求解，过程如上面的例子，知道所有节点的集合约束都满足为止。在之前讨论的数据流分析中，控制流图是静态的，程序分析过程中只会在节点间传播解集合信息，图本身没有变化，而这里的约束图会在分析过程中动态变化。考虑[Y]⊆X，每次Y的内容发生变化都将动态生成一些新的约束，因此分析过程中会不断产生新的边，整个分析过程需要不断迭代知道不再有边产生为止。可见，这种基于约束的指针分析有丰富的动态程序行为(dynamic irregular)，静态分析很难确知约束图的演化过程。基于约束图的指针分析算法如图7.8所示。

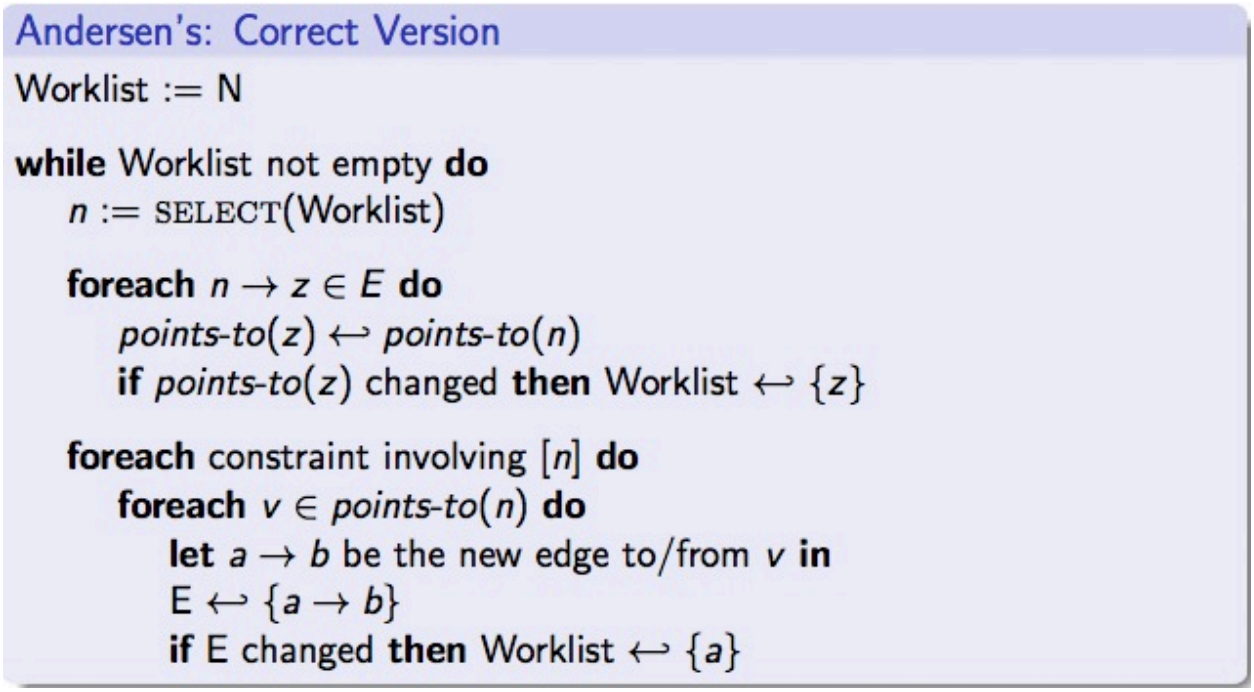

图7.8 Andersen-style指针分析算法

算法仍采用基于工作队列的框架，不过和数据流分析不同，这里没有IN/OUT集合。首先遍历所有边n->z，然后将n的指针信息传播给节点z，如果z的指针关系发生的改变，则需要将z添加到工作队列，以便将z新得到的指针关系传播给后继节点。

算法的关键是对[n]相关约束的处理。对每个涉及指针dereference的约束(x = !y或!x = y)，遍历n指向的遍历集合，对其中任何遍历v，考察以v为起点(终点)的边a->b，将其添加到约束图中，如果图中原来没有这条边(即在运行过程中动态生成的边)，则将该边的起始节点{a}添加到工作队列中。算法重复迭代知道工作队列为空为止，即所有节点的约束都得到满足。

下面举例说明此算法的工作过程：

例子3：

```
e = ref c;
if () {
  g = ref a;
  !g = e;
}else {
  c = ref f;
  d = !e;
  a = d;
  b = a;
  !e = b;
}
```

初始化：约束图包括a,b,c,d,e,f,g 7个节点，其中e->{c}，c->{f}，g->{a}，添加两条边d->a, a->b。下面开始迭代过程：

第一轮：

由!g = e, g->{a}，添加边e->a

由d = !e, e->{c}，添加边c->d

由!e = b, e->{c}，添加边b->c

传播指针信息，得到a->{f,c}, d->{f,c}, c->{f,c}, b->{f,c}。节点a,d,c,b构成了环。

第二轮，依次考察!g = e, d = !e, !e = b，发现没有边可添加，迭代完成。

这个算法的复杂度为O(V^3)，回顾前面流敏感指针分析算法的复杂度为O(N*V^4)，可见流不敏感指针分析的复杂度要低得多。

7.3.3 优化策略

Andersen-Style指针分析不如流敏感分析精确，但算法复杂度低得多，因此在很多程序分析场合有应用空间。尽管如此，最坏情况下O(V^3)的复杂度使得此算法仍存在效率问题，因此有一些工作研究如何进一步优化此算法的实现，一般从两个维度着手：(1) 设法减少约束图中的节点数目；(2) 设法降低节点指针关系集合(point-to set)的大小。

在指针分析研究中，人们很早就发现程序中总存在一些变量，它们的points-to sets相同，也就是说这些指针变量指向完全相同的一组变量。给定指针变量x和y，如果x和y的points-to set内容相同，那么称x和y pointer equivalent。如果能将具有相同指针关系的变量合并，则能减少约束关系图中的节点和边的数目，提高算法效率。但问题在于指针分析的目的就是要找出所有指针变量的point-to set，如何在算法运行过程中找到point to set相同的变量集合？

在例子3中，节点a,d,c,b构成一个环，从结果可以看出四个节点有完全相同的指针信息。对于流不敏感的指针分析来说，任何环上的节点一定有相同的point to set，因为只有一个

节点的指针关系集合发生改变，必定通过环传播到环上的其他节点。在一个约束图上找环可以用O(N+E)的Tarjan算法实现。

问题在于，约束图是动态图，所有的环并不是在初始化时已知，而是在指针分析过程中动态形成，这就需要在运行过程中动态检测环并合并环上的节点。但动态检测不那么容易操作，检测过于频繁能检测出更多的环，从而最大程度合并point equivalent节点，但频繁检测带来的开销可能抵消其好处；另一方面，检测过于迟缓则可能漏掉很多合并机会。为了找到合适的环检测时机，可以有很多启发策略。这里介绍一种：Lazy cycle detection, (LCD).

前面提到，约束图中有环意味着所有环上节点有相同的point to set，那么反过来理解，如果运行过程中发现多个节点有相同的point to set，那么意味着可能存在环。LCD的启发策略是：当沿边x->y传播x的指针信息给y时，如果发现二者的point to set相同，那么启动Tarjan算法检测环。下面举例说明LCD的应用：

例子4：

```
e = ref c;
if (_) {
  g = ref a;
  !g = e;
} else {
  c = ref f;
  d = !e;
  a = d;
  b = a;
  !e = b;
}
```

初始化约束图，节点集合: {a, b, c, d, e, f, g}，初始指针集合：c->{f}, g->{a}, e->{c}，添加边a->b, d->a，开始迭代循环。

第一轮：

!g = e ⇒添加边e->a

d = !e ⇒ 添加边c->d

!e = b ⇒ 添加边b->c

传播指针关系集合，a->{f,c}, b->{f,c}, c->{f,c}, d->{f,c}，abcd有相同的集合，启动Tarjan算法检测环，然后将a,b,c,d合并为一个节点ABCD，注意有一条e->ABCD的边

第二轮：

!g = e，因g->a，而e->ABCD已经存在，故不需添加边

d = !e，因e->{c}，c已经和d合并，不需添加边

!e = b，因e->{c}，而b->{c,f}，因此需要添加边ABCD->e

传播指针关系集合，a,b,c,d,e的point to set相同：{c,f}，g->{a}，因为e和ABCD有相同的指针关系，启动Tarjan算法检测环，合并ABCD和e为一个节点。
第三轮
依次检查!g = e, d = !e和!e = b，无边需要添加，迭代结束。
Andersen指针分析有很多应用，除了LCD意外，还有其他一些重要优化(Pointer/Location equivalence, Offline optimization)，更多详细细节可以参考Hardekopf et al “The And and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code”和Hardekopf et al “Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis”.

8 类型约束和Steensgaard指针分析

这部分从类型系统出发，最后以基于类型约束的Steensgaard指针分析结束。

8.1 类型系统

所有编程语言都有数据类型。所谓类型就是对一类数据的抽象，基本类型比如32位整数，32位/64位的浮点、布尔、字符/字符串等等。可以用基本类型通过类型构造函数得到符合类型，常见的如数组、函数、用户自定义结构体/联合等等。
类型系统涉及很多术语，使用上很混乱。下面提到的术语基于Luca Cardelli的文章”Type Systems”。
程序执行中的错误可以分为两类：(1) trapped errors，导致程序立刻终止执行，如除零、Java中的数组越界访问；(2) Untrapped errors，程序在出错后继续执行，但可能出现任意行为，如C里的缓冲区溢出，Jump到错误的地址。如果一个程序执行时不可能出现untrapped错误，则认为该程序是安全的(safe program)；程序设计语言是安全的当且仅当所有该语言的程序是安全的(safe language)。
设计语言时可以定制一组禁止的行为(forbidden behaviors)，forbidden errors必须包括所有untrapped错误，但可能包含trapped errors。如果程序执行不可能出现禁止行为，则认为该程序是well behaved。如果一种语言的所有程序都满足well behaved条件则认为该语言是strongly typed，否则就认为是weakly typed。strongly/weakly typed是用的最广但却使用不规范的术语。
如果语言使用静态类型系统在编译时拒绝ill behaved程序，则该语言是静态类型的语言(statically typed)；否则如果在运行时动态拒绝ill behaviors，则是动态类型语言(dynamically typed)。静态类型的语言还可进一步分为显式类型(explicitly typed)和隐式类

型(implicitly typed)两种。如果类型是语言语法的一部分，那么是显式类型语言；否则类型必须通过编译时推导，是隐式类型的语言。表8.1总结了常见语言的分类。

表8.1 常见语言的类型系统

类型系统特征	语言举例
无类型	汇编语言
弱类型、静态类型	C/C++
弱类型、动态类型检查	Perl/PHP
强类型、静态类型检查	Java/C#
强类型、动态类型检查	Python, Scheme
静态显式类型	Java/C
静态隐式类型	O'Caml, Haskell
强类型、静态隐式类型	Lingo

显然，语法正确的程序未必well typed，well typed程序未必是符合程序员意图的正确程序。检查一个程序是否well typed由类型系统完成。类型系统认为程序well typed当且仅当可以证明该程序所有的执行路径都不可能出现未定义的行为(undefined behavior)。由于这个原因，和程序分析一样，类型检查是对程序的"保守"分析，即可能拒绝事实上正确的程序。例如下面这个例子：

Example

```
def f = fun(n) { ... }
def g = fun(m) { ... }
x := f(10); y := g(42); z := 1 + f
```

图8.1 Ill-typed, well behaved程序举例

这个程序是ill-typed，因为将一个函数类型和整数1相加不符合类型运算规则，另一方面，如果这个程序实际执行时不可能执行到z = 1 + f，那么就是一个合法的程序。可见程序语言的类型系统有两个极端，一类语言几乎不在类型系统上引入任何约束，如C/C++，给程序员最大创造空间，但可能出现未定义的行为；一类语言(如C#/Java)试图在设计之初就给语言所有可能的行为规定边界，界定了哪些行为是禁止的，通过编译时的类型检查拒绝ill behaved的程序，但表达能力肯定不如弱类型语言。设计一种语言既能在编译时拒绝大部分ill behaved的程序，同时又保留弱类型语言的表达能力并非易事。

8.2 Lingo类型系统

Lingo的类型系统如下。基本类型包括int, bool, unit和ref。引入unit是出于一种学究的目的，即保证任何函数任何表达式任何语句都有一个合理的类型，但有时候程序员并不关心，比如表达式a = b返回的类型是unit。注意和C的表达式类型的区别。另外这里把基本数据类型和函数的类型的定义显式分开了，函数的返回值只可能是基本数据类型而不能是函数类型，即在Lingo中不允许高阶函数(允许一个函数返回一个函数)。 变量的定义、函数的定义和声明以及程序的语法如图所示。

Types

$$\tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid \texttt{ref}\ \rho$$
$$\lambda ::= \tau \to \tau \qquad \rho ::= \tau \mid \lambda$$

Explicitly-Typed Lingo

$$vd \in VDec ::= \textbf{decl}\ x = \tau \mid vd\ \textbf{,}\ vd$$
$$fd \in FDec ::= \textbf{decl}\ x = \lambda \mid fd\ \textbf{,}\ fd$$
$$f \in FDef ::= \textbf{def}\ x_1 = \textbf{fun}\ (x_2 : \tau)\ \{vd\ \textbf{,}\ c\ \textbf{;}\ \textbf{return}\ x_3\}$$
$$p \in Prog ::= [fd\ \textbf{,}\ f+]\ vd\ \textbf{,}\ c$$

图8.2 Lingo类型系统

语言的类型系统通常体现为一组推导规则，例如如果表达式e1和e2都是int，则e1+e2一定是int类型，e1:int ^ e2:int => e1+e2:int，用记号表示如下。

Example (Inference Rule)

$$\frac{e_1 : \texttt{int} \qquad e_2 : \texttt{int}}{e_1 + e_2 : \texttt{int}} \ (\textsc{add})$$

图8.3 类型推导规则举例

这只是一种记号，横线上方写类型推导条件，横线下方写类型推导结果。为了完整描述Lingo的类型系统，只要对每种语法元素(表达式、语句、函数、程序)定义好类型推导规则。首先看表达式常量，类型推导规则如下(考虑int和bool两种情况)：

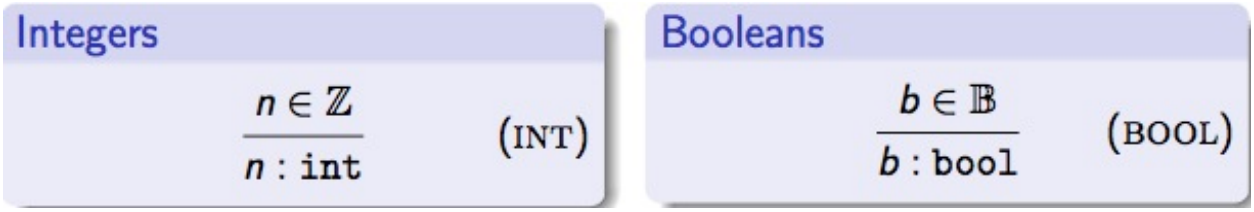

图 8.4 常量类型推导规则

为了描述变量的类型，需要引入类型环境Γ: x->τ，Γ是一个类型映射函数，给定变量x得到其对应的类型τ。引入类型环境后，变量表达式的类型推导规则如下，读作可以证明变量x的类型是τ如果类型环境Γ将x映射为τ。

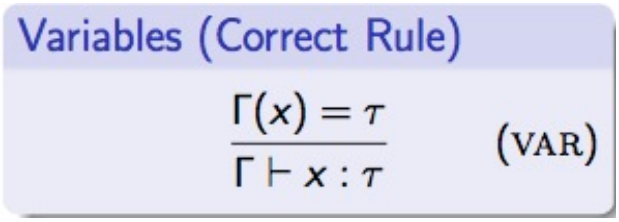

图 8.5 变量表达式类型推导规则

指针表达式的类型推导规则如下。如果变量x的类型是τ，则ref x的类型是ref τ；如果变量x的类型是ref τ，则!x的类型是τ。

图8.6 指针表达式的推导规则

布尔、算术及关系表达式的推导规则如下，原则同前面加法表达式。

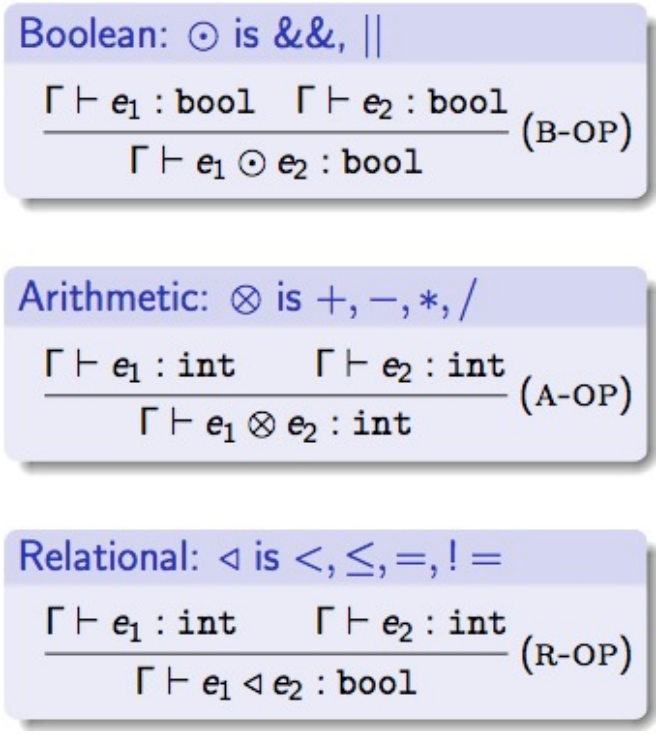

图8.7 布尔表达式推导规则

基本语句的类型推导规则如下， skip语句相当于nop操作， 返回类型unit；顺序语句c1; c2返回类型unit当且仅当c1和c2都有类型unit。赋值语句有类型unit当且仅当左值表达式和右值表达式类型相同。

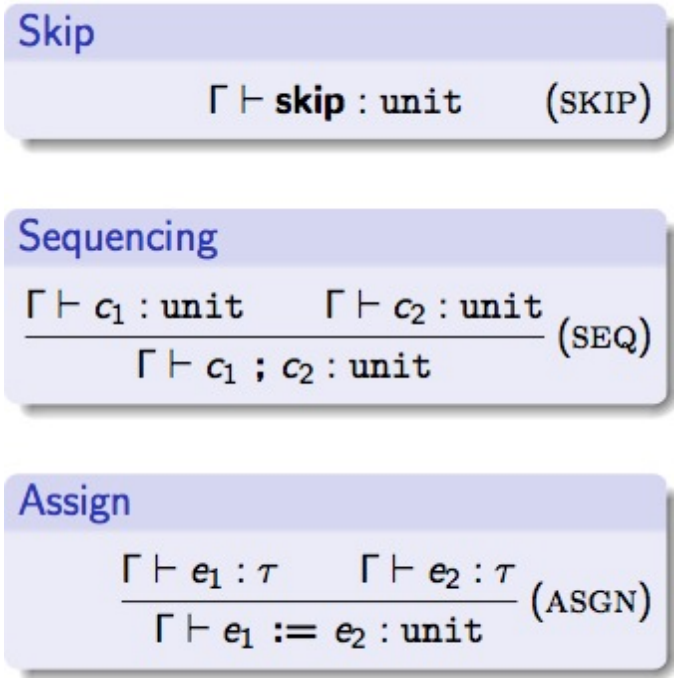

图8.8 基本语句的类型规则

if/else语句的类型检查规则如下。if/else语句有unit类型，当且仅当条件表达式e有bool类型，且语句c1和c2都有unit类型。可见，类型系统引入unit主要是类型检查方便。同理有while语句的类型推导规则。

If

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash c_1 : \texttt{unit} \qquad \Gamma \vdash c_2 : \texttt{unit}}{\Gamma \vdash \textbf{if}\ (e)\ \textbf{then}\ \{c_1\}\ \textbf{else}\ \{c_2\} : \texttt{unit}} \quad (\text{IF})$$

While

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash c : \texttt{unit}}{\Gamma \vdash \textbf{while}\ (e)\ \textbf{do}\ \{c\} : \texttt{unit}} \quad (\text{WHILE})$$

图8.9 if/else语句的类型规则

函数定义和函数调用的类型检查规则如下。函数定义：x1 : τ1->τ2，其中x1是表示函数的变量名，τ1和τ2分别表示参数和返回值的类型。由于函数参数和局部变量需要打开新的作用域，因此创建一个新的类型环境包括函数内可见的变量定义。函数定义的类型正确当且仅当调用参数和返回值的类型正确且函数体内所有语句的类型正确。函数调用的类型推导规则只需要检查调用参数和返回值。

Function Definition

$$\frac{\Gamma \vdash x_1 : \tau_1 \to \tau_2 \qquad \Gamma' = \Gamma[\text{add } vd, x_2 \mapsto \tau_1] \qquad \Gamma' \vdash x_3 : \tau_2 \qquad \Gamma' \vdash c : \texttt{unit}}{\Gamma \vdash \textbf{def}\ x_1 \ = \ \textbf{fun}\ (x_2 : \tau_1)\ \{vd\ ,\ c\ ;\ \textbf{return}\ x_3\} : \texttt{unit}} \quad (\text{FDEF})$$

Function Call

$$\frac{\Gamma \vdash e : \tau_1 \to \tau_2 \qquad \Gamma \vdash x : \tau_1}{\Gamma \vdash e(x) : \tau_2} \quad (\text{CALL})$$

图8.10 函数类型推导规则

在表达式、语句、函数及函数调用类型推导规则的基础上，整个程序的类型规则如下。条件部分操作类型环境，即将所有全局可见的函数和变量类型添加到类型环境中，推导程序中所有函数定义和变量生命的类型，如果通过则表示类型检查没有问题。

Program

$$\frac{\Gamma = [\text{add } fd] \qquad \Gamma \vdash f_i : \texttt{unit} \qquad \Gamma[\text{add } vd] \vdash c : \texttt{unit}}{\vdash fd \ , \ f_i \cdot vd \ , \ c : \texttt{unit}} \ (\text{PROG})$$

图8.11 程序的类型推导规则

根据上述类型推导规则，很容易实现一个针对Lingo的类型检查工具。类型检查的基本思想都是递归地检查每一个表达式、语句和函数是否有正确的类型绑定。任何语言的编译器都做类型检查，实际实现要复杂得多，因为：(1) 实际程序的类型系统往往复杂得多，需要考虑的边角情况非常多；(2) 在类型检查的过程中输出合理的出错信息，合理的错误信息能提示程序员的错误地点，对辅助调试有意义。

8.3 类型推导和类型约束

类型检查的一项重要工作是类型推导。对于静态显式类型的语言，如C/C++或Java程序员需要给每个变量显式指定类型，这样虽然能准确表达程序员意图，但有时候显得繁琐。像python一类的脚本语言没有这些麻烦，整个程序写下来可以完全不指定类型，原因在于对于程序员的一些日常任务，编译器可以实现自动类型推导。

图8.12是一个例子。由w = 10得知w：int，从而知道AbsVal函数定义的参数x：int，因此知道其返回值y：int，因此得知z：int。

Example (Concrete Program)

```
def AbsVal = fun (x) {
   if (x < 0) then { y := x } else { y := 0-x } ; return y
}
w := 10 ; z := AbsVal(w)
```

图8.12 类型推导举例

在算法上实现类型自动推导和类型检查需要用到类型约束。前一章用到集合约束来做Andersen指针分析，这里用到的是Term约束，其定义如下。集合约束用集合包含关系表达约束条件，Term约束使用关系表达式(E1=E2)表达约束。注意这里的变量是指类型，构造函数是类型构造函数。

Definition (Term Constraint Language)

$$x \in Variable \quad c \in Constructor$$

$$E \in Expr ::= x \mid c(E_1, \ldots, E_{a(c)})$$
$$S \in Stmt ::= E_1 = E_2 \mid S_1 \wedge S_2$$

图8.13 Term约束语言

相应的，Term约束的化简规则如下：

Rewrite Rules

$$S \wedge \boxed{E = E} \rightarrow S$$

$$S \wedge \boxed{x = E} \rightarrow S[x \mapsto E] \qquad S \wedge \boxed{E = x} \rightarrow S[x \mapsto E]$$

$$S \wedge \boxed{c(E_1, \ldots, E_k) = c(E'_1, \ldots, E'_k)} \rightarrow S \wedge \left(\bigwedge_{i \in 1..k} E_i = E'_i \right)$$

图8.14 Term约束的化简规则

由于E=E总是成立，因此可以直接消去：S ^ E=E -> S。另外x=E和E=x的处理过程完全一样，即将即将绑定x->E添加到约束集合S中，=只表示等价关系，不区分左值和右值。由于这个原因，构造函数分解规则不用考虑covariant和contravariant条件。和前文基于集合约束的指针分析一样，基于约束的类型推导第一步是定义Universe of Discourse，内容如下：

A. 每个函数定义f对应一个类型变量*f*；

B. 函数f中每个变量*x*对应类型变量*xf*

C. 基本类型对应无参数类型构造函数，int, bool, unit

D. 指针对应一元构造函数ref

E. 定义二元函数构造函数->。这里只考虑函数只有一个参数的情况，多个参数可以在此基础上扩展。

有了Universe of Discourse，第二部是约束生成(constraint generation)：

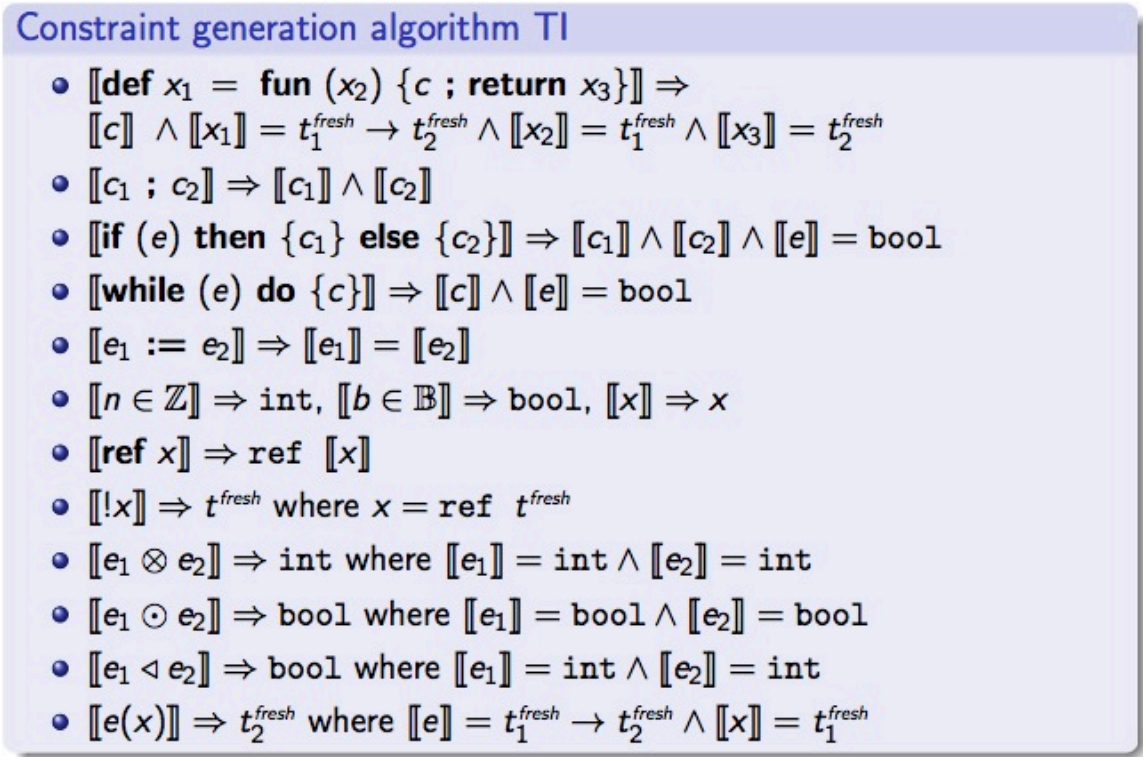

图8.15 类型推导约束生成

下面逐条来理解上述每条规则的意义：

A. 递归处理函数体的类型[[c]]，如果函数x1：t1->t2，那么参数和返回值分别有t1和t2类型。这里仅仅是前文类型推导规则的约束表达形式；

B. 对于顺序语句[[c1;c2]]，要求[[c1]]和[[c2]]同时满足；

C. 对于if/else语句，要求if条件为bool类型且[[c1]]和[[c2]]同时满足。后续每条约束对应前文的类型推导规则，略。

类型推导举例1：

```
d = 42;
a = ref d;
b = ref e;
c = ref f;
a = b;
!a = c;
```

首先由d = 42知d：int；由a = ref d知a：ref int；由a=b，依据上面等价表达式处理规则a和b有相同类型，故b：ref int；由于a: ref int，故!a：int，根据!a = c，二者有相同类型，因此c为int，但c = ref f，int和ref类型不相容，因此类型检查失败，此程序是ill-typed。

因为约束求解过程中只涉及union运算，实际实现中有一种高效的数据结构可用于求解term约束：union-find集合数据结构，定义两个基本操作：

A. Find(x)：返回x所在的集合；

B. Unite(x,y)：对x和y所在的集合求并。这里使用unite而不是union的原因是实际实现时，union和C/C++里面的关键字冲突。

此算法的复杂度为O(n* α(n))，其中 α(n)是inverse Ackermann函数，对于大多数实际问题规模都接近常数。算法定义：

A. 初始每个元素所在的集合只有自己：对任意元素n，有n.parent = n；

B. find(x)的定义：

A. if(x.parent == x) return x;
B. else return find(x.parent);

C. unite(x)的定义：

A. x = find(x); y = find(y); x.parent = y;

find操作的复杂度由集合元素构成的树的深度决定，因此基本算法的find过程是O(n)的复杂度。一般有两种优化方法：(1) union-by-rank，每次执行unite(x,y)操作时，考虑x和y代表的集合元素构成的树的深度，总是将深度较小的集合合并到深度大的集合，即避免树垂直增长；(2) 路径压缩，即在每次find的时候更新节点的parent域，即让find经过的节点直接指向parent，修改后的find定义如下：

```
find(x) {
  if(x.parent == x) return x;
  else {
    x.parent = find(x.parent);
    return x.parent;
  }
}
```

union-find是一种基本的数据结构，网上有更多资料描述其用法。

8.4 Steensgaard指针分析

类型约束的另一个应用是指针分析，由Steensgaard首次提出。和Andersen指针分析一样，这种分析也是基于约束的流不敏感指针分析。由于这种分析只考虑变量的类型约束，因此比Andersen指针分析更不精确，后面会比较几种指针分析的相对精度。

下面以Lingo的子集(去掉函数)为例说明指针分析的过程。

第一步定义Universe of Discourse：

A. 每个程序变量x对应类型类型变量τx；

B. 对一元类型构造函数ref(τ)表示对类型τ的引用；

第二步约束生成：

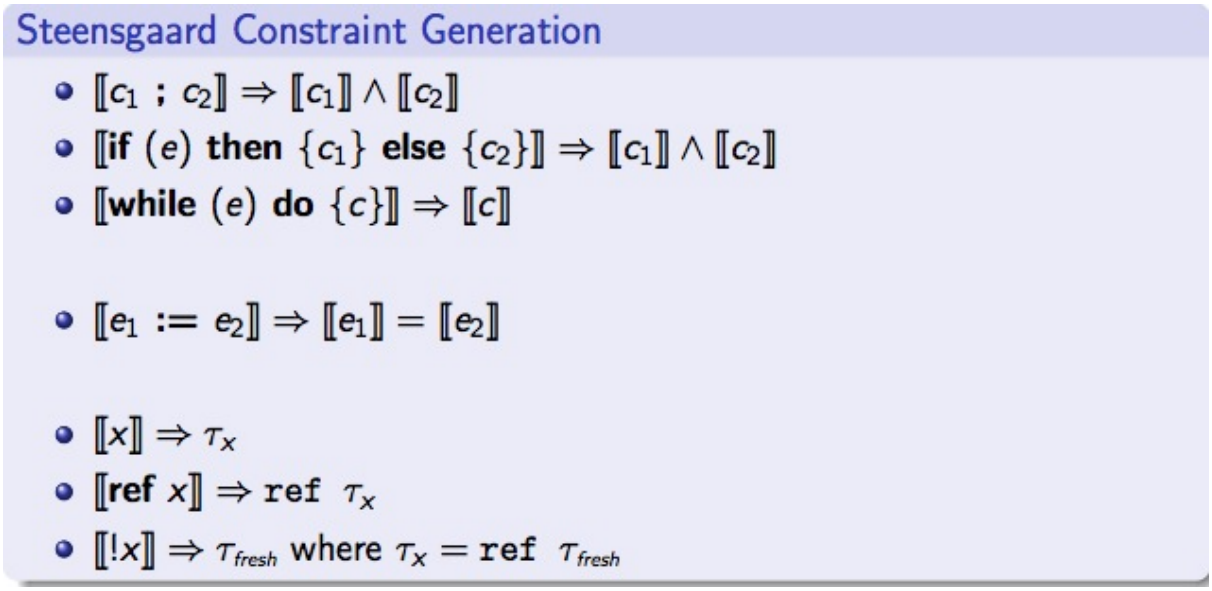

图8.16 Steensgaard指针分析约束生成

分析过程举例2：

程序：

```
d = 42;
a = ref d;
b = ref e;
c = ref f;
a = b;
!a = c;
```

这是个ill-typed程序，前面对这个程序例子进行了类型推导。Steensgaard指针分析不关心程序的类型检查是否通过，只分析指针关系。因此即使对于ill-typed程序也能得到分析结果。

```
a = ref d; => a-> {d}
b = ref e; => b-> {e}
c = ref f;  => c-> {f}
a = b; => a->{d,e} ^ b->{e,d}
!a = c; => d->{f} ^ e->{f}
```

分析过程举例3：

```
z = ref x;
w = ref a;
a = 42;
if (a<b) {
  !z = ref a;
  y = ref b;
} else {
  x = ref b;
```

```
  y = w;
}
```

分析过程不关心控制条件，过程如下：

z = ref x; => z-> {x}
w = ref a; => w-> {a}
!z=ref a; => x-> {a}
y = ref b; => y-> {b}
x = ref b; => x->{a, b}
y = w; => y->{a,b} ^ w->{a,b}

基于union-find数据结构的Steensgaard指针分析如下：

(1) 初始化

对任意变量x，初始化其points-to-set为空，即x.ptsto = {}

定义辅助函数merge(a,b)，合并变量集合a和b的points-to集合。

```
merge(a,b)
{
  a = find(a); b = find(b);
  ap = a.ptsto; bp = b.ptsto;

  new_rep = unite(a,b);
  if( new_rep == a) {
    repp = ap; oldp = bp;
  } else {
    repp = bp; oldp = ap;
  }

  if(repp != oldp) {
    if(repp == {})
      new_rep.ptsto = oldp;
    else if(oldp == {})
      skip;
    else merge(repp,oldp);
  }
}
```

接下来对每种语句的处理过程：

(2) x = ref y

```
x = find(x);
merge(x.ptsto,y);//将y加入x的points-to集合
```

(3) x = y

```
x = find(x); y = find(y);
merge(x.ptsto,y.ptsto);
```

(4) x = !y

```
x = find(x);
y = find(y); yp = find(yp.ptsto);
if(yp.ptrto == {}) {
  yp.ptsto = x.ptsto;
}else{
  merge(x.ptsto,yp.ptsto);
}
```

(5) !x = ref y

```
x = find(x); xp = find(x.ptsto);
if(xp.ptsto=={}) {
  xp.ptsto = {y};
}else {
  merge(xp.ptsto,y);
}
```

(6) !x = y

```
x = find(x); xp = find(x.ptsto);
y = find(y);
if(xp.ptsto == {}) {
  xp.ptsto = y.ptsto;
}else {
  merge(xp.ptsto,y.ptsto);
}
```

(7) !x = !y

```
x = find(x); xp = find(x.ptsto);
y = find(y); yp = find(y.ptsto);
if(xp.ptsto == {}) {
  xp.ptsto = yp.ptsto;
}else if(yp.ptsto=={}){
  yp.ptsto = xp.ptsto;
}else merge(xp.ptsto,yp.ptsto);
```

最后一个问题：Steensgaard指针分析、Andersen指针分析、Flow-Sensitive指针分析三者之间的关系是什么？

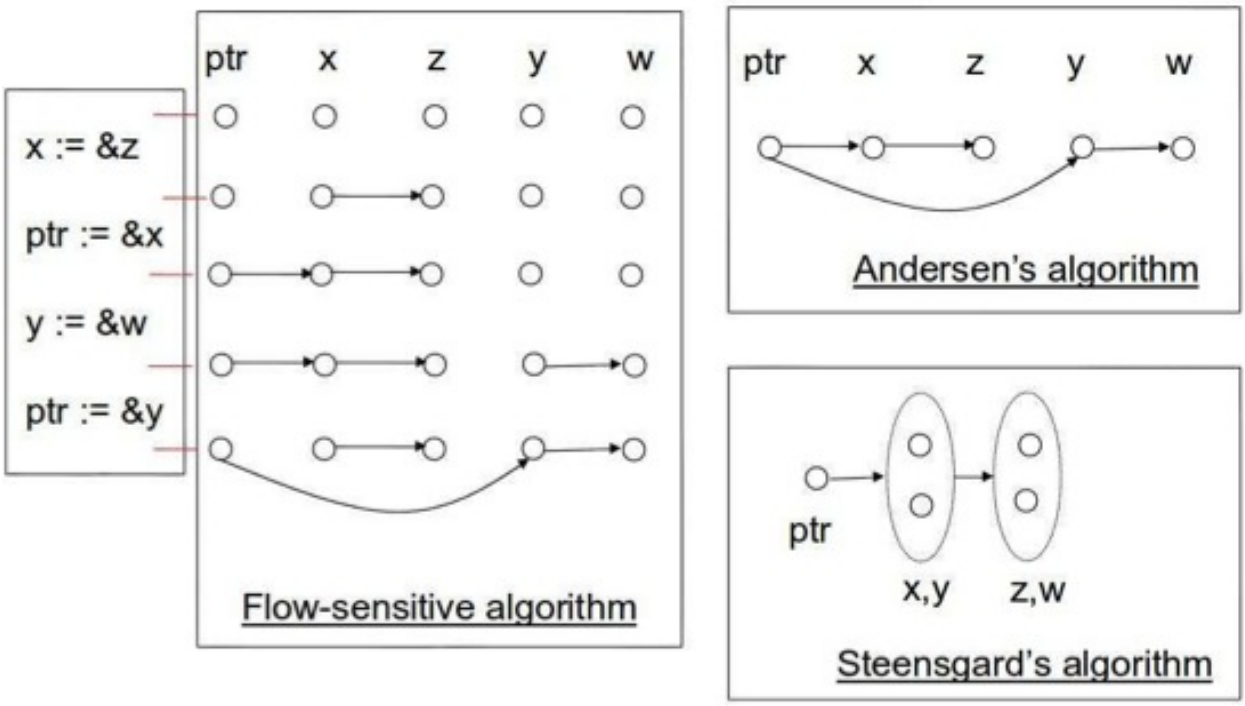

图8.16 Steensgaard-Andersen-Flow-Sensitive三者关系

由图中可以清晰看出三者之间的相对精度：Flow sensitive >= Andersen >= Steensgaard.

参考文献

控制流分析

[1] Keith D. Cooper, Timothy J. Harvey and Ken Kennedy. “A Simple, Fast Dominance Algorithm”. Software: Practice and Experience, 2001; 4:1-10.
[2] Esko Nuutila, Eljas Soisalon-soininen. “On Finding the Strongly Connected Components in a Directed Graph”. Information Processing Letters, 1994.

数据流分析

Kildall “A unified approach to global program optimization”
Kam et al, “Monotone data flow analysis frameworks”
Cooper et al “An empirical study of iteraive data-flow analysis”
Marlowe et al “Properties of data flow frameworks

Abstract Interpretation

Blanchet “Introduction to Abstract Interpretation”
Salcianu “Notes on Abstract Interpretation”
Schimdt “Trace based abstract interpretation of Operational Semantics”

稀疏分析

Cytron et al “Efficiently Computing Static Single Assignment Form and the Control Dependance Graph”
Wegman et al “Constant Propagation with Conditional Branches”

指针分析

Hind, “Pointer Analysis: Haven’t We Solved this Problem Yet?”
Hardekopf et al, “Semi-Sparse Flow-Sensitive Pointer Analysis”
Hardekopf et al, “Staged Flow Sensitive Pointer Analysis”
Chow et al, “Effective Representation of Aliases and Indirect Memory Operations in SSA Form”

过程间分析

Sharir et al. “Two Approaches to Interprocedural Data Flow Analysis”

约束分析

Hardekopf et al “The And and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code”
Steensgaard, “Points to analysis in almost linear time”