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：

Definition (Control-Flow Graph)

Formally, a control-flow graph $G = (N, E)$ is:

- A set of nodes N, one for each program statement
- A set of edges $E \subseteq N \times N$, such that $x \to y$ iff statement y may execute directly after statement x
- There is a unique *entry node* and a unique *exit node*

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：

The *dominator relation* DOM $\in N \times N$:

- x DOM y $\Leftrightarrow$ every path from *entry* to y must pass through x
- reflexive, transitive, anti-symmetric

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

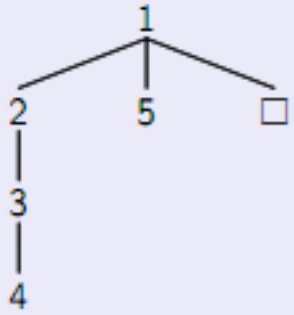

图2.3 dominator树举例

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：

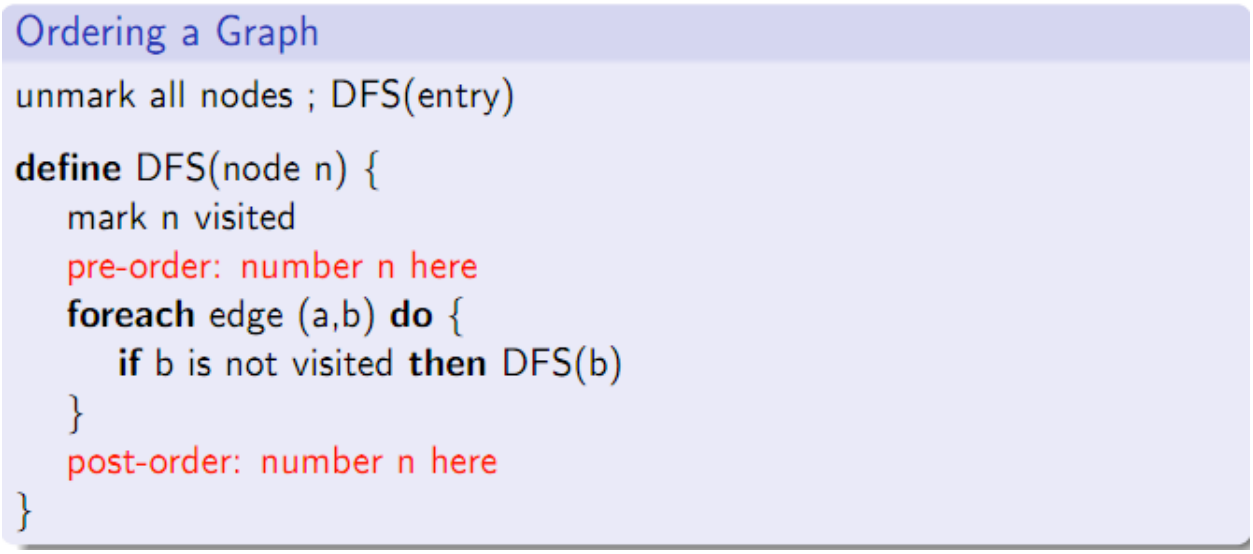

图2.5 DFS过程中的先序和后序编号

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 $\Leftrightarrow$

- $\exists$ 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。

Program

```
1input z;
2x := 2;
3y := x+3;
4if (z < y) then {
   5while (z < w) do {
       6x := y;
       7y := x+3;
       8w := y-x
   }
} else {
    9y := x;
    10w := x-y
}
```

Is node 6 control-dependent on node 4?

NO (see the formal definition). Node 4 doesn't strictly control the execution of node 6, node 5 does. But we can say that node 6 is *transitively* control-dependent on node 4, since 6 depends on 5 which depends on 4.

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。

Program

```
1 x := 6;
2 y := x/2;
3 while (y > 0) do {
    4 if (y < x) then {
        5 x := x/y;
        6 y := y-1
    } else {
        7 skip
    };
    8 x := x-1
  }
```

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。

参考文献

[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.