

编译原理第四阶段实验报告

Use-Definition Chaining

计 24 2012011335 柯均洁

一、实验总述

在实验框架下完成了数据流分析中的引用-定值链（Use-Definition Chaining）的求解

首先求出了到达-定值数据流，然后在到达-定值数据流信息的基础上计算 UD 链的信息，其规则如下：

求变量 A 在位于基本块 B 中的引用点 u 的 UD 链

1. 如果在基本块中，变量 A 的引用点 u 之前有 A 的定值点，且距离 u 最近的定值点为点 d（即 A 在 d 的定值可以到达 u），那么 A 在点 u 的 UD 链就是{d}
2. 如果在基本块中，变量 A 的引用点 u 之前没有 A 的定值点，那么，in[B]中 A 的所有定值点均到达 u，它们就是 A 在 u 的 UD 链

二、UD 链实现细节

（1）到达-定值数据流计算

1. 在 Tac 类中增加一个 globalNum 域

原来的 Tac 中只保存了其所在的 BasicBlock 编号，而计算到达定值数据流需要知道每条语句在整个 Functy 中的编号，于是在 Tac 类中增加 globalNum 域来记录该语句的全局编号。

在 FlowGraph 的 markBasicBlocks 函数中增加对 TacList 中语句的全局编号的计算，从 0 开始编号，逐条语句加 1

2. 增加 DefRefPoint 类

为了记录定值和引用点的信息，在 BasicBlock.java 中增加了 DefRefPoint 类，类中保存了被定值或者引用的变量 Temp，已经该变量被定值或者引用的语句编号
重载了两种比较器，一个是先按被引用或定值的变量排序的 TEMP_COMPARATOR，另外一个先按语句编号进行排序的 INDEX_COMPARATOR

3. gen[B] kill[B] out[B] in[B]

由于在计算中不仅需要知道这些集合的语句编号，还需要知道这些语句编号所具体对应的变量。因此将这四个集合保存为 TreeSet<DefRefPoint>。按照 TEMP_COMPARATOR 排序。

4. gen[B]计算

在 BasicBlock 中的 computeDefAndLiveUse 函数中同时计算 gen[B]。出现变量的定值的时候，动态地刷新每个变量最新被定值的位置。当所有语句处理完，就会得到 B 中被定值的变量以及它所被定值的位置。

5. kill[B]计算

在 FlowGraph 中增加了 calKill 函数。在得到所有的 gen[B]之后，就可以计算出所有变量的有效定值点集合 genAll，每个基本块的 kill[B]是 gen[B]集合中变量所对应的有效定值点几何除去 gen[B]

6. prev[B]计算

在到达定值数据流计算中需要用到基本块的前驱。因此在 BasicBlock 中增加 Set<Integer> prev，保存该基本块的直接前驱。

在 FlowGraph 中增加了 calPrev 函数，利用 BasicBlock 原有的后继保存域 next[2]来计算 prev

7. out[B]和 in[B]计算

在 FlowGraph 中增加了 analyzeArriveDef 函数来计算到达-定值数据流，首先调用 calPrev()和 calKill()。然后按照算法流程计算 out[B]和 in[B]

```
(1)  for  $i := 1$  to  $n$  {                                //置初值
(2)       $in[B_i] := \emptyset$ ;
(3)       $out[B_i] := gen[B_i]$ ;
(4)  }
(5)  change := true;
(6)  while change {
(7)      change := false;
(8)      for  $i := 1$  to  $n$  {
(9)           $newin := \cup out[b] (b \in P[B_i])$ ;
(10)         if  $newin \neq in[B_i]$  {
(11)             change := true;
(12)              $in[B_i] := newin$ ;
(13)              $out[B_i] := gen[B_i] \cup (in[B_i] - kill[B_i])$ 
(14)         }
(15)     }
(16) }
```

(2) UD 链计算

在 BasicBlock 中增加了 calUseDefChain 函数

- 为了知道在每个引用点之前是否有块内的定值，需要动态记录每个变量的块内最新定值点保存为一名叫 newGenPointInBlock 的 hashtable。
- 将 UD 链保存为名叫 udChain 的 Map<DefRefPoint, List<Integer>>。也就是说每条 ud 链都是一个变量的引用点，对应一条全局语句标号的链表。

- 对每个引用点，按照算法计算其 UD 链。

三、UD 链输出

为了单独输出 filename.ud，在 BasicBlock 中增加了 printUDChainTo 函数，但是就要单独为输出 ud 文件单独在 Option 中增加一个 Level，看到实验要求中不能改变 Option，所以最终决定在原来活跃变量数据流输出信息的基础上增加，将 udChain 的信息输出到原来的.dout 文件中，稍微改变了原有的格式。

也就是说，在 BasicBlock 的 printLivenessTo 增加了 Use-Def Chain 的输出，并且将 Tac 语句进行了编号。因为没有改变原实验框架下 Driver 中的内容，若需要看到增加的输出，就需要将 Option 设为 Level3。输出样例见下页。

输出样例

```
1 FUNCTION main :
2 BASIC BLOCK 0 :
3   Def      = [ _T3 _T4 _T5 _T6 _T7 _T8 ]
4   liveUse = [ ]
5   liveIn  = [ ]
6   liveOut = [ _T3 _T5 ]
7   Use-Def Chain :
8     (_T6, 2): [1]
9     (_T7, 4): [3]
10    (_T8, 6): [5]
11    (_T5, 7): [2]
12
13    1: _T6 = 1      [ _T6 ]
14    2: _T5 = _T6    [ _T5 ]
15    3: _T7 = "wow!" [ _T5 _T7 ]
16    4: _T4 = _T7    [ _T5 ]
17    5: _T8 = 3      [ _T5 _T8 ]
18    6: _T3 = _T8    [ _T3 _T5 ]
19    7: END BY BEQZ, if _T5 = 0 : goto 2; 1 : goto 1
20 BASIC BLOCK 1 :
21   Def      = [ _T9 _T10 ]
22   liveUse = [ _T3 ]
23   liveIn  = [ _T3 _T5 ]
24   liveOut = [ _T3 _T5 ]
25   Use-Def Chain :
26     (_T3, 9): [6]
27     (_T9, 9): [8]
28     (_T10, 10): [9]
29
30    8: _T9 = 5      [ _T3 _T5 _T9 ]
31    9: _T10 = (_T3 * _T9) [ _T5 _T10 ]
32   10: _T3 = _T10   [ _T3 _T5 ]
33   11: END BY BRANCH, goto 2
```

- 在原有的 liveOut 后面加入了 Use-Def Chain 的输出，依次是每个引用点以及对应的 UD 链
- UD 链的信息含义为：
(变量，变量引用点的语句标号): [ud 链的语句标号]
- 在原有的 tac 语句输出中增加了语句标号