PA4 实验报告

计 76 沈诣博 2017011427

本阶段的工作

本人在原有框架上增加了求解 DU 链的方法和接口，改动文件如下：

1.dataflow/BasicBlock.java：

增加了函数 computeDUDefAndLiveUse(Map<Temp,Set<BasicBlock») 初始化修改定义之后的 LiveUse 集合，并且求出基本块内的 DU 链。

增加了函数 computeDEFED(Map<Temp,Set<BasicBlock»)，使用画家算法求出修改定义之后的 Def 集合。

增加了函数 analyzeDUChain() 添加块和块之间的 DU 链。

2.dataflow/FlowGraph.java：

增加了函数 analyzeDUChain() 使用修改定义之后的 LiveUse 和 Def 集合按照公式求出修改定义之后的 LiveOut，从而用它定位块和块之间的 DU 链。

BasicBlock.java 的修改如下：

在函数 computeDUDefAndLiveUse(Map<Temp,Set<BasicBlock») 中，传入的参数是一个空 Map 的指针，用来记录不同的变量被定值的基本块。我参考了函数 computeDefAndLiveUse() 的逻辑，对于每一个块，从头到尾遍历语句，遇见定值前引用的变量便保存进 LiveUse() 中，遇见定值后引用的变量则和最近一次的定值信息一起加入基本块的 DU 链；遇见定值点，则记录该点在块内定值或者更新最后的定值点，并且将块的指针保存进 Map 中。在对所有块遍历之后，我们分别求出了不同块的 LiveUse 信息，块内的 DU 链；并且求出了程序中所有寄存器的定值位置，从而可以在下一步建立 Def 集合。

在函数 computeDEFED(Map<Temp,Set<BasicBlock») 中，传入的参数和上一个函数相同，通过不同寄存器的定值位置反解求出不同的 Def 集合；在对所有块遍历之后，它们的 Def 集合也就求出来了。

函数 analyzeDUChain() 在计算得出等效的 LiveOut 之后调用，按照公式，计算出来的 LiveOut 包括 DU 链中的定值元素和它在块外的引用点，综合在第一个函数中求出来的最后一个定值点，便得到块于块之间的 DU 链。

不同的集合和公式说明如下:

修改之后的 LiveUse(B) 为对 (s,A) 的集合, 其中 s 是块 B 中某点,s 引用变量 A 的值, 且 B 中在 s 前面没有 A 的定值点。
修改之后的 Def(B) 为对 (s,A) 的集合, 其中 s 是不属于 B 的某点,s 引用变量 A 的值, 但 A 在 B 中被重新定值。
求值公式如下:

$$LiveIn[B] = LiveUse[B] \bigcup (LiveOut[B] - Def[B])$$

$$LiveOut[B] = \bigcup (LiveIn[b]), b \in S[B]$$

其中 S[B] 是块 B 的全部后继。
使用公式 1，2 对于不同的块的 LiveIn，LiveOut 进行循环赋值，直到得到一个 LiveIn 和 LiveOut 的闭包，此时的不同属性均符合要求。
FlowGraph.java 的修改如下：
在函数 analyzeDUChain() 中实现了上述算法，且在创建类的时候按顺序调用不同 BasicBlock 中的上述接口和 analyzeDUChain()，最终得到一条完整的 DU 链。

分析样例

TestCases/S4/t0.decaf 对应的.s 文件输出和 DU 链的分析如下：

```
FUNCTION _Main_New :
BASIC BLOCK 0 :
1 _T0 = 4 [ 2 ] //_T0在第二句中被传参，之后并未出现
2 parm _T0
3 _T1 =  call _Alloc [ 5 6 ]  //_T1在5,6句中被引用，然后并未出现
4 _T2 = VTBL <_Main> [ 5 ]  //_T2在5,6句中被引用，然后并未出现
5 *(_T1 + 0) = _T2
6 END BY RETURN, result = _T1

FUNCTION main :
BASIC BLOCK 0 :
7 call _Main.f
8 END BY RETURN, void result

FUNCTION _Main.f :
BASIC BLOCK 0 :
9 _T7 = 0 [ 10 ]  //_T7在10句中被引用，然后并未出现
10  _T5 = _T7 [ ] //_T5在之后并未被引用
11  _T8 = 1 [ 12 ]  //_T8在12句中被引用，然后并未出现
12  _T6 = _T8 [ ] //_T6在之后并未被引用
13  _T10 = 0 [ 14 ] //_T10在14句中被引用，然后并未出现
```

```
14  _T9 = _T10 [ 21 24 30 ] //_T9在21,24,30句中被引用，然后并未出现
15  _T11 = 2 [ 16 ] //_T11在16句中被引用，然后并未出现
16  _T3 = _T11 [ 18 ] //_T3在18句中被引用，然后在23句被杀死
17  _T12 = 1 [ 18 ] //_T12在18句中被引用，然后并未出现
18  _T13 = (_T3 + _T12) [ 19 ]  //_T13在19句中被引用，然后并未出现
19  _T4 = _T13 [ 28 ] //_T4在28句中被引用，然后在29句被杀死（重新赋值）
20  END BY BRANCH, goto 1
BASIC BLOCK 1 :
21  END BY BEQZ, if _T9 =
      0 : goto 7; 1 : goto 2
BASIC BLOCK 2 :
22  _T14 = 1 [ 23 ] //_T14在2句中被引用，然后并未出现
23  _T3 = _T14 [ 35 ]//_T3在35句被引用，然后并未出现
24  END BY BEQZ, if _T9 =
      0 : goto 4; 1 : goto 3
BASIC BLOCK 3 :
25  call _Main.f
26  END BY BRANCH, goto 4
BASIC BLOCK 4 :
27  _T15 = 1 [ 28 ]//_T15在28句被引用，然后并未出现
28  _T16 = (_T4 + _T15) [ 29 ]//_T16在39句被引用，然后并未出现
29  _T4 = _T16 [ 28 32 36 ]//_T4按照4–6的顺序在36句被引用，4–6–4的顺序在28句
    被引用，4–5的顺序在32句被引用，然后在33句被杀死
30  END BY BEQZ, if _T9 =
      0 : goto 6; 1 : goto 5
BASIC BLOCK 5 :
31  _T17 = 4 [ 32 ]//_T17在32句被引用，然后并未出现
32  _T18 = (_T4 − _T17) [ 33 ]//_T18在33句被引用，然后并未出现
33  _T4 = _T18 [ 28 36 ]//_T4沿着5–6的顺序在36句被引用，沿着5–6–1的顺序在28
    句被引用，然后在29句被杀死
34  END BY BRANCH, goto 6
BASIC BLOCK 6 :
35  _T5 = _T3 [ ]//_T5在之后并未出现
36  _T6 = _T4 [ ]//_T6在之后并未出现
37  END BY BRANCH, goto 1
BASIC BLOCK 7 :
38  END BY RETURN, void result
```

和 2.2 图进行比较，发现：块 1 中，前 14 句是和块之间跳转条件相关的准备；而 15，16 句的 tac 对应图中的 d1(i:=2)；它们 DU 链的并集减去自己的编号为 18；

17,18,19 句 tac 对应图中的 d2(j:=i+1)，而此处它们 DU 链的并集为 28；

27，28，29 句 tac 对应的是块 3 中的 d4(j:=j+1)，此时它们 DU 链的并集为 28,32,36；

31,32,33 句 tac 对应的是块 4 中的 d5(j:=j-4)，此时它们 DU 链的并集为 28,36；
35 句对应块 6 中的 d6，36 句对应块 6 中的 d7，它们的 DU 链都为空；
22,23 句的 tac 对应的是块 2 中的 d3(i := 1)，它们的 DU 链是 35，而前面已经提到了,35 句对应点 d6。
综上所诉，可以由这一份 t0.du 得出：
DU[d1]={},DU[d2]={d4},DU[d3]={d6},DU[d4]={d4,d5,d7},DU[d5]={d4,d7},DU[d6]={},DU[d7]={}
易得，其和讲义 2.4.2 节给出的 DU 链是一致的。