Decaf PA 4

王世因 2016011246

（1）实验基本情况简介

Tac

这指的是一条语句，对应的重要属性有：

```
public enum Kind {
    ADD, SUB, MUL, DIV, MOD, NEG, LAND, LOR, LNOT, GTR, GEQ, EQU, NEQ, LEQ,
    LES, ASSIGN, LOAD_VTBL, INDIRECT_CALL, DIRECT_CALL, RETURN, BRANCH,
    BEQZ, BNEZ, LOAD, STORE, LOAD_IMM4, LOAD_STR_CONST, MEMO, MARK, PARM
}
public Kind opc;
public Tac prev; // 前一条语句
public Tac next; // 后一条语句
public Temp op0; // 操作符的参数0
public Temp op1; // 操作符的参数1，可能为空
public Temp op2; // 操作符的参数2，可能为空
public Set<Temp> liveOut; // 在执行后仍保持活跃的变量，总程序运行中需要多次访问进行更新
```

因为我们在计算各种集合的时候需要访问各种变量的定义和使用情况，因此我增加了三个属性，记录在这条语句中使用的变量和定义的变量，默认值是`null`

```
public Temp def;
public Temp used1, used2;
```

在`Tac`初始化的时候，枚举各种语句类型给上面的三个属性赋值。

```
private void defDU(){
    switch (opc) {
        case ADD:
        case SUB:
        case MUL:
        case DIV:
        case MOD:
        case LAND:
        case LOR:
        case GTR:
        case GEQ:
        case EQU:
        case NEQ:
```

```
            case LEQ:
            case LES:
                used1 = op1;
                used2 = op2;
                def = op0;
                break;
            case NEG:
            case LNOT:
            case ASSIGN:
            case INDIRECT_CALL:
            case LOAD:
                used1 = op1;
                def = op0;
                break;
            case LOAD_VTBL:
            case DIRECT_CALL:
            case RETURN:
            case LOAD_STR_CONST:
            case LOAD_IMM4:
                def = op0;
                break;
            case STORE:
                used1 = op0;
                used2 = op1;
                break;
            case PARM:
                used1 = op0;
                break;
            default:
                break;
        }
    }
```

BasicBlock

在代码生成中，我们把代码切分成若干个`BasicBlock`，所有的`BasicBlock`之间存在有向连接关系，程序的执行是在这些`BasicBlock`之间跳转，其中涉及的函数和属性为：

```
public Tac tacList; //BasicBlock中的语句的列表，需要经常性地遍历
for (Tac tac = tacList; tac != null; tac = tac.next) {}//通过这种方式遍历
```

涉及的各种集合：

```
public Set<Temp> def; // 在块中定义的变量
public Set<Temp> liveUse; // 在块中用到的变量
public Set<Temp> liveIn; // 在进入块的时候保持活跃的变量
public Set<Temp> liveOut; // 在离开块的时候保持活跃的变量
public Set<Temp> saves;
```

用于维护上面的集合的函数；

```
public void analyzeLiveness();
public void insertBefore(Tac insert, Tac base);
public void insertAfter(Tac insert, Tac base);
public void computeDefAndLiveUse();
public Map<Pair, Set<Integer>> DUChain;
```

因为这里原有代码都实现的很好了，所以并不需要我进行太多修改，只是增加了一个属性，便于搜索的时候标记是否访问过：

```
public boolean visited;
```

FlowGraph

这里我们需要在访问每个FlowGraph的过程中，遇到每一个定义后，都要计算DU链，然后通过事先写好的`BasicBlock.DUChain`函数自动完成之后的计算。换句话说，本次PA的任务就是给每一个`BasicBlock.DUChain`计算对应的取值。

这样问题就很简单了，每次看到一个定义，我通过树状搜索的方式找到这个定义的所有引用，递归进行，把返回值传入DU链的封装接口。

```
for (BasicBlock bb : bbs) {
    Tac taclist = bb.tacList;
    for (Tac t = taclist; t != null; t = t.next){
        // search the DU chain for each definition
        if (t.def != null){
            Pair pair = new Pair(t.id, t.def);
            Set<Integer> locations = new TreeSet<Integer>();
            for (BasicBlock blk : bbs){
                blk.visited = false;
            }
            search(t.def, locations, t.next, bb, true);
            bb.DUChain.put(pair, locations);
        }
    }
}
```

需要注意的是，针对当前的块儿，访问应该从定义开始，因为此块的前半部分没有针对这个定义访问过，所以不能标注访问；针对其他的块儿，访问从块儿的开头开始，而且要标记已经访问。一路上加入所有的引用此定义变量的语句到DU链，如果碰到重新定义此变量就返回。

```
private void search(Temp variable, Set<Integer> locations, Tac start,
BasicBlock bb, boolean begin) {
    if (bb.visited) {
        return;
    }
```

```
    if (!begin){
        start = bb.tacList;
        bb.visited = true;
    }

    for (Tac t = start; t != null; t = t.next) {
        if (t.used1==variable || t.used2==variable) {
            locations.add(t.id);
        }
        if (t.def==variable) { // if the variable is redefined, then end
search
            return;
        }
    }

    // search following BasicBlocks
    switch (bb.endKind) {
        case BY_BEQZ:
        case BY_BNEZ:
            if (bb.var == variable) {
                locations.add(bb.endId);
            }
            search(variable, locations, null, getBlock(bb.next[1]), false);
        case BY_BRANCH:
            search(variable, locations, null, getBlock(bb.next[0]), false);
            break;
        case BY_RETURN:
            if (bb.var == variable) {
                locations.add(bb.endId);
            }
            break;
    }
}
```

（2）以 TestCases/S4/t0.decaf(对应于讲义第 12 讲 2.2 节图 4)为例，分析输出的 TAC 序列与 DU 链信息，并验证它与讲义中 2.4.2 节给出的结果是一致的。

t0.decaf 代码长度比较短，通过观察，可以分为以下的几个块儿，**这里我把相同的块儿放到了一行**

```
class Main {

    static void main() {f();}

    static void f() {
        int i;int j;int a;int b;a = 0;b = 1;bool flag;flag = false;i = 2;j
= i + 1;
```

```
        while (flag) {

            i = 1;if (flag){

                f();}

            j = j + 1;if (flag){

                j = j - 4;}

            a = i;b = j;}}}
```

根据下面的输出，手动推导所有的DU链为：

D	U
a=0;	[]
b=1;	[]
flag=false;	[while(flag), if(flag), if(flag)]
i=2;	[j=i+1]
j=i+1;	[j=j+1]
i=1;	[a=i]
j=j+1;	[b=j, j=j-4, j=j+1]
j=j-4;	[j=j+1]
a=i;	[]
b=j;	[]

经过比较，手动推导的和程序生成的DU链一致。

程序因为处理存储的原因，增加了很多DU链，比如分配内存、记录函数表、定义和赋值分开写，这里我们可以忽略。

t0.du

```
FUNCTION _Main_New :
BASIC BLOCK 0 :
1   _T0 = 4 [ 2 ]
2   parm _T0
3   _T1 =  call _Alloc [ 5 6 ]
4   _T2 = VTBL <_Main> [ 5 ]
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 ]
10  _T5 = _T7 [ ]
11  _T8 = 1 [ 12 ]
12  _T6 = _T8 [ ]
13  _T10 = 0 [ 14 ]
14  _T9 = _T10 [ 21 24 30 ]
15  _T11 = 2 [ 16 ]
16  _T3 = _T11 [ 18 ]
17  _T12 = 1 [ 18 ]
18  _T13 = (_T3 + _T12) [ 19 ]
19  _T4 = _T13 [ 28 ]
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 ]
23  _T3 = _T14 [ 35 ]
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 ]
28  _T16 = (_T4 + _T15) [ 29 ]
29  _T4 = _T16 [ 28 32 36 ]
30  END BY BEQZ, if _T9 =
        0 : goto 6; 1 : goto 5
BASIC BLOCK 5 :
31  _T17 = 4 [ 32 ]
32  _T18 = (_T4 - _T17) [ 33 ]
33  _T4 = _T18 [ 28 36 ]
34  END BY BRANCH, goto 6
BASIC BLOCK 6 :
35  _T5 = _T3 [ ]
36  _T6 = _T4 [ ]
37  END BY BRANCH, goto 1
BASIC BLOCK 7 :
38  END BY RETURN, void result
```

遇到的问题和实验体会

通过第五次书面作业，我熟悉了DU链的基本原理，理解了BasicBlock和FlowGraph的数据结构，对写代码起到了很重要的辅助作用。可见，在写代码前，需要提前熟悉实验的原理，否则事倍功半。

因为实验文档过长的原因，一开始我不太理解本次PA到底要做什么，后来上网查了一些资料并和同学讨论后才恍然大悟。

在寻找所有的使用和定义的过程中，我一开始忘记考虑跳转语句中用到的变量，思维还不够严谨。后来根据`BasicBlock.computeDefAndLiveUse`函数中的代码，照猫画虎，结构化地考虑完了所有的情况。

我调试时间最长的错误是，不能把入口的块儿也标成`visited = true;`因为此块儿在定义前的部分还可能没有访问过。而且，再次访问的时候，会被认为是本变量的再次定义，然后直接返回，不会出现死循环。