# Decaf PA4 实验报告*

邓博文

2019 年 2 月 23 日

## 1 本阶段工作

### 1.1 框架解读

这里先简单说明一下自己对于已有框架的理解。首先,正如在 README 文档里提到的，现有框架进行数据流分析、汇编代码生成的基本单位是函数 (对应代码中的`Functy`)。进行数据流分析时，代表数据流图的`FlowGraph`和`Functy`是一一对应的关系，一个数据流图也正是根据同一个`Functy`内的各个基本块 (`BasicBlock`) 建立的。

### 1.2 数据流图建立

建立数据流图的步骤如下:

- `deletMemo`: 删除对应标注 (Memo) 的 Tac 语句，这些标注只会在类的成员函数中，作用是指示函数的参数，类的非静态成员函数都会隐含 this 关键字作为参数。

*使用的 JDK 为 JDK11

- `markBasicBlocks`：标注每一个 Tac 语句属于哪一个基本块并且删除那些不是跳转目标的行号（Mark 语句）。

- `gatherBasicBlocks`：建立各个基本块。为各个基本块分配对应的 Tac 语句序列，指明各基本块如何结束（return, branch, bnez, beqz），并确定每一个基本块的后续基本块。实际上这一步就已经基本确定了整个数据流图中各个基本块的关系。

- `simplify`：确定各个基本块的入度（有多少基本块可以到达此基本块），移除那些不可达的基本块或者 Tac 语句序列为空的基本块，修改对应的流图间转移的关系（如以 bnez 或者 beqz 结束且其中某一分支被移除了，那么需要合并两个分支），并为每一个剩下的基本块分配新的标号。至此，整个数据流图已经确定。

- `allocateTacIds`：为各个基本块中的 Tac 语句分配唯一的 id（使用的是`IDAllocator`类的静态方法，所以此 id 不仅仅在一个基本块内部唯一，而且全局唯一）。

- `analyzeLiveness`：以基本块为单位进行活跃变量分析，实现的就是课上提到的算法。

- `BasicBlock.analyzeLiveness`：以 Tac 语句为单位进行活跃变量分析。之所以要进行这一步是为了后续输出时可以输出运行到某一 Tac 语句时仍然活跃的变量有哪些。因为当前 Tac 语句的 LiveOut 就是下一个语句的 LiveIn，所以对每个 Tac 语句，只需要保存其 LiveOut 即可。具体计算时从当前基本块最后一个 Tac 语句开始，它的 LiveOut 就是整个基本块的 LiveOut，之后计算其 LiveIn（也就是上一个 Tac 语句的 LiveOut），这里只需要先让 LiveIn 等于 LiveOut，之后再根

据 Tac 语句的种类，从其中移除 Def 再加上 LiveUse 即可。

## 1.3 求解 DU 链

### 1.3.1 概述

自己是按照第 12 讲讲义 2.4.2 节中描述的算法来求解 DU 链。
对于 DU 链的求解，本质上还是求解数据流方程1

$$\begin{cases} LiveOut(B) & = \cup LiveIn(B_i) \\ LiveIn(B) & = LiveOut(B) \cup (LiveUse(B) - Def(B)) \end{cases} \tag{1}$$

这里涉及到的四个集合都需要携带位置信息。现在对各个集合的含义进行说明:

- $Def(B) : (p, A)$ p 点是基本块 B 之外的一点，A 在 p 点被使用，且在基本块 B 之内被定义。
- $LiveUse(B) : (p, A)$ p 点基本块是 B 内一点，A 在 p 点被使用，且在基本块 B 中 p 点之前，A 未被定义。
- $LiveIn(B) : (p, A)$ p 点是基本块 B 内一点，A 在 p 点被使用，并且基本块 B 未定义过 A。
- $LiveOut(B) : (p, A)$ p 点是基本块 B 之外的一点，A 在 p 点被使用。

观察可以发现，这里的 p 点都是变量被使用的位置。这点很重要，自己一开始没有想清楚，导致一直没办法理解方程中集合的差和并操作的意义。

### 1.3.2 实现

分别定义包含位置信息的四个集合 liveUseDU, liveInDU, liveOutDU, defDU，以及不包含位置新的 redef（重定义）集合，其作用会在后面说明。

在`BasicBlock`中添加`visited`变量，作用在后面会说明。

在`Tac`中添加`isGenerator`函数，作用是判断当前 Tac 语句是不是会定义变量的语句，用于后续判断哪些语句在输出时需要加上 DU 链的信息。

修改`computeDefAndLiveUse`，加入 liveUseDU 的计算步骤。具体计算方法和计算 liveUs 时类似，都是逐条 Tac 语句遍历，根据其种类，将适当的操作数加入 liveUseDU。不同之处在于根据 liveUseDU 的定义，不能使用 def，因为 def 实现时是当前块在此语句之前没有访问（使用或者定义）变量时，才会将其添加到 def 集合内，在某些情况下可能会漏掉一些变量。所以使用 redef 集合记录各语句定义的变量，因为是顺序访问基本块中的语句，所以如果访问当前语句时 redef 不包括某一变量，则此语句之前一定没有定义过该变量。在函数结束前，将 LiveInDU 集合初始化为 LiveUseDU 集合。

在`FlowGraph`类中加入`computedefDU`函数。它的作用是计算各基本块的 defDU 集合。这里是整个算法的难点。困难之处在于，按照 defDU 的定义，对于基本块中某一变量 A 的定值点 p，需要找到某一对其重新定义点之前所有的引用点。因为数据流图中可能可能存在环，并且每一个基本块至多有两个后续基本块，所以就需要遍历所有的可能性并且不重复。这里需要注意到 defDU 中的变量必定是由 redef 中的变量组成的，只是引用点位置不同而已，并且随着某一条路径不断变长，每次某一基本块定义了 redef 中的变量，经过这个基本块的所有路径之后都不可能使得 defDU 增加了。基于这一性质，在遍历过程中的某一个时刻，下一个基本块一定不能是已经在当前路径上的某一基本块。所以每一次扩充路径，都将新加入的基本块的 visited 加 1（初始值为 0），之后扩充路径时如果某一候补基本块 vistied

不为 0，那么就跳过。在实现这个函数的过程中，发现了原有代码中的一个 bug。原始代码中，在`gatherBasicBlocks`函数中对于不以 4 中跳转语句结束的基本块，会根据其是否有下一基本块将其归为以 return 或者 branch 结束。对于以 return 结束的情况，当前基本块的两个 next 应该都为-1，但是原始代码里没有对这种情况进行处理，导致其值为 0，这使得我实现的算法在有些情况下会误认为最后一个基本块可以回到第一个基本块（编号为 0），进而导致死循环。另外需要注意的一点是，因为 Java 中的集合变量都是引用，对其修改会影响到初始的变量，所以对于每一个基本块，为了保证访问其可能存在的两个后续块时 redef 是一致的，需要对 redef 进行拷贝，并将拷贝后的集合传入新一轮的迭代。剩下的实现就比较简单了，只需要判断新加入的基本块的 liveUseDU 是否包含 redef 中的变量即可，如果包含，则扩充 defDU 集合。

在`FlowGraph`类中加入`resolveDUChain`函数。此函数遍历所有基本块，调用`computedefDU`计算 defDU，之后按照求解活跃变量时完全一样的方法求解数据流方程1即可。

在`BasicBlock`类中加入`resolveDUChain`函数。作用是根据 Tac 语句的种类，判断其是否定义了变量，如果定义了，则需要输出 DU 链信息。对当前基本块中其后的语句进行遍历。如果有对定义的变量的重定义，那么 DU 链就是重定义点之前所有引用其值的语句位置的集合。如果没有，那么就是其后所有引用其值的语句位置的集合与 LiveOutDU 集合的并集。

# 2 分析 TestCases/S4/t0.decaf

to.decaf 进行数据流分析后共有 3 个基本块。逐个分析如下：

```
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
```

此基本块对应自动生成的 Main 类的 new 函数。其 DU 链十分简单。无需过多解释。

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

此基本块对应 main 函数，没有变量定义，所以自然也没有 DU 链输出。

```
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
```

这部分包含了第 12 讲讲义中图 4 的流图。其中对应关系为（不完全对应，代码中包括其他变量以及语句）

- $BASIC\ BLOCK\ 0 \rightarrow B_1$
- $BASIC\ BLOCK\ 2 \rightarrow B_2$
- $BASIC\ BLOCK\ 4 \rightarrow B_3$
- $BASIC\ BLOCK\ 5 \rightarrow B_4$
- $BASIC\ BLOCK\ 6 \rightarrow B_5$

可以看出 _T5 是 a，_T6 是 b，_T3 是 i，_T4 是 j。
在 2.2 节图 4 中，$d_1$ 对应 15-16，$d_2$ 对应 17-19，$d_3$ 对应 22-23，$d_4$ 对应 27-29，$d_5$ 对应 31-33，$d_6$ 对应 35，$d_7$ 对应 36。

- $i$ 在定值点 $d_1$ 的 DU 链为 $\{d_2\}$: 16 [18],18 [19($d_2$)]
- $j$ 在定值点 $d_2$ 的 DU 链为 $\{d_4\}$: 19 [28], 28 [29 ($d_4$)]

- $i$ 在定值点 $d_3$ 的 DU 链为 $\{d_6\}$：23 [35 ($d_6$)]
- $j$ 在定值点 $d_4$ 的 DU 链为 $\{d_4, d_5, d_7\}$：29 [28, 32, 36 ($d_7$)], 28 [29 ($d_4$)], 32 [33 ($d_5$)]
- $j$ 在定值点 $d_5$ 的 DU 链为 $\{d_4, d_7\}$：33 [28, 36 ($d_7$)], 28 [29 ($d_4$)]
- $d_6, d_7$ 的定值点没有被引用，因此其 DU 链为空。