实验一：基于 JoeQ 的数据流分析

李晨昊 2017011466

2019-10-1

目录

1 MySolver

首先按照提示收集所有所有会使函数退出的 quads，以及计算出函数的入口 quad。

其实这些值对数据流的计算并没有帮助，之所以需要得到它们，只是为了在数据流的计算结束之后，更新函数的入口或者出口处的数据流值。

之后根据数据流的方向进行迭代求解。每条 quad 的初始值直接全部赋成 top 即可，不必考虑 entry 或者 exit 的值，因为它们并不对应于任何一条 quad。在迭代计算的时候，如果一个 quad 的前驱是 entry，或者后继是 exit，则需要用到 entry 或者 exit 的值，对应的值在迭代的过程中不可能变化：例如，如果需要用到 entry，则一定是向前的数据流，entry 的值不可能改变，exit 的值会改变，但是不会参与计算，所以只需要在迭代结束之后更新 exit 的值即可。

考虑了上面这一点之后，迭代的实现基本上就和标准的算法完全一样了。

2 ReachingDefs

部分参考了已有的 `ConstantProp` 和 `Liveness` 的实现，不过感觉那些代码写的都太丑了，因此我在此基础上做了一些修改：

1. `copy`，`setToTop` 和 `setToButtom` 都没有实现，直接 throw exception 了，因为这三个函数都没有用到。
2. 已有的代码里到处都是保护性拷贝，这充分体现了 java 缺少生命周期，可变性等概念的缺点。我实现的比较精细，自己注意一下那些地方需要避免指针 alias，即可避免绝大多数的拷贝。

顺便说一句，现在的确感觉 rust 写多了之后，写这种古老的语言的代码有点不太舒服了。暑假的时候我用 rust 重写了编译原理的实验框架decaf-rs，在里面也实现了基于数据流分析的优化，代码量比这个要少得多。

核心逻辑 `processQuad` 的实现思路基本上也和标准算法是一样的，对于这条 quad 的所有 def，在集合中删除它们所有的定值点 (按照定义，*kill* 集合应该不包含自身，不过其实是否包含自身对结果是没有影响的)。然后如果这条 quad 含有 def，就把这条 quad 加入集合。在预处理的过程中，为了后面能够便于找到一个 register 的所有定值点，需要遍历一遍 quads 收集相关的信息。

3 Faintness

注意到 faint variable 的定义是：非活跃 (dead) 的或只用于计算 faint variable 的变量，也就是说它和活跃变量其实是相反的，那么不难想象它的 meet operator 应该是集合交。不过为了充分利用 `Liveness` 中的已有代码，我依然按照活跃变量来实现了 (即 “非 faint 变量”)，为了得到 faint 变量，只需要在最后输出的时候取一个补集即可。

核心逻辑 `processQuad` 的实现思路是，如果这条语句是 `Binary` 或者 `Move`，就先查看它的赋值目标是否活跃，如果不活跃的话就不把它的操作数加入集合。否则 (如果赋值目标活跃，或者这条语句不是 `Binary` 或者 `Move`)，都直接按照正常的活跃变量分析进行。

4 TestFaintness

经测试 readme 中给出的例子，即：

```
int foo()
{
  int x = 1;
  int y = x + 2;
  int z = x + y;
  return x;
}
```

并不能实现 readme 中描述的效果，查看 quads 可以发现，最后一条 `return` 直接就是 `return`

`1`，而不是 `return x` 了 (也许是 javac 或者 joeq 进行了常量传播的优化)，所以 `x` 也会被当成 faint variable。经尝试，只需要把 `int x = 1;` 修改成用变量来初始化 `x`，就可以达到预期的效果。这就是函数 `test2`。

此外还实现了一个函数 `test3`，主要是为了检查一些特殊的使用下是否能正确检测非 faint variable，例如用于函数传参等。