report-PA4

计71 钟闰鑫 2017010306

实现流程与遇到的困难

死代码消除的耗时主要花费在了理解框架内的其他三个优化的代码实现上以及一些细节debug上

这里选择了采用框架内的`Flow`来求解数据流方程，剩下的事情主要就是如何构造出def/live_use，如何使用`Flow`和如何优化

def/live_use的构造

由于框架已经处理好关于基本块的所有事情，def/live_use的构造只需要利用其结构提供的接口构造就好

采用从后往前的构造方式，每个tac语句判断其相关的寄存器编号，考虑是否放入def/live_use的`HashSet`中，按照顺序，在def里插入定值的编号，在live_use里删除定值的编号，然后在live_use里插入被使用的编号，其中live_use的插入删除顺序必须注意，因为会有诸如`%1 = %1 + 1`之类的指令等

以下几点是在实现过程中遇到的坑

- 框架中将基本块中的跳转之类的指令拿出来了，放到了名为`next`的enum上，而这些跳转，比如`Jif`，是有可能使用到寄存器的，因此也需要放到live_use里
- 由于是从后往前的构造，跳转指令之类的对于live_use的更新应当放到最开始(因为跳转指令是在基本块最后)

`Flow`的使用

框架中的`Flow`的采用迭代算法，使用bitset进行了加速

由于死代码消除是后向数据流，在实际实现中，将live_use对应到gen，def对应到kill，live_in对应到out，live_out对应到in

除此之外，实现中还要注意的有

- `Flow`所使用的`Meet`应该用`Or`而不是`And`，因为在通过后继块的live_in来求前面的live_out的时候是要用**并**

```
let mut alive_flow = Flow::<Or>::new(f.bb.len() + 1, reg_len);
```

- out(live_in)在迭代开始应该默认为空集，假定所有编号在基本块开始都不活跃，然后一步一步增加活跃变量

```
for x in out.iter_mut() { *x = 0; }
```

(虽然后面发现这些在实验指导书上已经写了，但果然还是得做一遍才知道这些初值的含义orz)

实际优化

通过`Flow`解出in(live_out)和out(live_in)后，就可以用in(live_out)来进行优化了

实际流程也是从后往前逐条tac语句进行判断，边删除边更新此时的活跃变量集合

这一步操作与def/live_use的构造类似，也有一些坑点

- 同样，优化删除过程中也需要考虑不在基本块内的跳转指令的使用，在开始循环之前进行判断，并更新活跃变量集合
- 对于活跃变量的删除、添加操作需要注意，实现的流程如下

  1. 若定值未在活跃变量中

     将该条tac语句删除(对于`Call`则更改)，不对活跃变量集合进行操作

  2. 若定值在活跃变量中

     不删除该条语句，将定值从活跃变量集合中删除，再将使用的编号添加进活跃变量集合中

  虽然这样的流程在一次死代码删除中仍有一些问题，比如一些后面的活跃变量在tac被删除后就不活跃了，但由于可以进行多次死代码删除，因此实际上是没有问题的

性能测试

下面对优化性能(tac数量)进行测试，优化次数都为10次，其中最后一个测例不在给出测例中(见testcase/S4/useless_assign.decaf)，是一个简单的含有无用赋值的decaf

	没有优化	**只有死代码消除**	**公共表达式，常量传播，复写传播**	**四种优化都有**
basic-basic	38	38	38	32
basic-fibonacci	2987	2987	3253	2720
basic-math	110	110	110	109
basic-queue	2739	2676	2934	2483
basic-stack	616	616	638	585
mandelbrot	3664076	3664076	4605304	3088148
rbtree	2299376	2257973	2583042	2030896
sort	489113	489113	608105	413865

	没有优化	只有死代码消除	公共表达式，常量传播，复写传播	四种优化都有
useless_assign	52	36	52	32

从上表可以看出，单纯的死代码消除仅在本身就含有大量无用赋值时才有较显著的效果，对于一般的程序来说基本没有什么优化效果，而公共表达式等的使用则可以提供更多的死代码，从而能在死代码消除后得到非常显著的效果