PA4实验报告

计73 林俊峰

2017011303

实现思路

消除死代码的步骤为

1. 构造基本块并建立 `CFG`
2. 通过活跃变量分析计算出每个块的 `LiveIn` 和 `LiveOut`
3. 由3的结果计算块中每条 `tac` 语句的 `LiveIn` 和 `LiveOut`
4. 对所有基本块进行后向遍历，若某语句的定值变量不在该语句的 `LiveOut` 中，则该语句为死代码，若无副作用则直接删除它。
5. 对于有副作用的死代码，这里只考虑函数调用的情况。将 `a = call b` 改为 `call b`
6. 由于每次删除语句会导致 `CFG` 活跃变量数据流产生变化，因此需要重复步骤2-5直到不再有语句发生变化。

在 `Java` 的 `decaf` 框架中，步骤1-3均已在 `dataflow` 模块中实现，直接调用对应方法即可

实现代码

```
public TacProg transform(TacProg input) {
    var analyzer = new LivenessAnalyzer<TacInstr>();
    CFG<TacInstr> cfg = null;
    for (var func : input.funcs){
        var seqs = func.getInstrSeq();
        boolean change = false;
        do {
            var builder = new CFGBuilder<TacInstr>();
            cfg = builder.buildFrom(seqs);  //生成CFG
            analyzer.accept(cfg);            //活跃变量分析
            change = false;
            for (var node : cfg.nodes) {
                var it = node.backwardIterator();
                while (it.hasNext()) { //后向遍历基本块
                    var loc = it.next();
                    Temp dst = null;
                    if (loc.instr.dsts.length > 0 &&
!loc.liveOut.contains(loc.instr.dsts[0])) {
                        //若有函数调用，则进行替换
                        if (loc.instr instanceof TacInstr.DirectCall) {
                            var entry =
((TacInstr.DirectCall)loc.instr).entry;
                            var instr = new TacInstr.DirectCall(entry);
                            func.replace(loc.instr, instr);
                        } else if (loc.instr instanceof
TacInstr.IndirectCall) {
                            var entry =
((TacInstr.IndirectCall)loc.instr).entry;
                            var instr = new
TacInstr.IndirectCall(entry);
                            func.replace(loc.instr, instr);
                        } else { //若无函数调用（无副作用），直接删除该语句
                            func.remove(loc.instr);
                            change = true;
                            break;
                        }
                    }
                }
                if (change) break;
            }
        }while(change); //若删除了语句，重新进行活跃变量分析
    }

    return input;
}
```

实现完之后发现跑测例偶尔超时，原因是每次有语句被删除后都会重新进行数据流分析以及遍历。考虑到每个基本块中的语句为顺序执行，故每次只需要把被删除语句的 `LiveOut` 赋值给下一条要遍历的语句即可，无需重新计算。代码如下

```
public TacProg transform(TacProg input) {
    var analyzer = new LivenessAnalyzer<TacInstr>();
    CFG<TacInstr> cfg = null;
    for (var func : input.funcs){
        var seqs = func.getInstrSeq();
        boolean change = false;
        do {
            var builder = new CFGBuilder<TacInstr>();
            cfg = builder.buildFrom(seqs);
            analyzer.accept(cfg);
            change = false;
            for (var node : cfg.nodes) {
                var it = node.backwardIterator();
                Loc<TacInstr> changedLoc = null;                //记录被
删除语句
                while (it.hasNext()) {
                    var loc = it.next();
                    Temp dst = null;
                    if (changedLoc != null) {                   //上一条
语句被删除，则传递LiveOut
                        loc.liveOut = changedLoc.liveOut;
                        changedLoc = null;
                    }
                    if (loc.instr.dsts.length > 0 &&
!loc.liveOut.contains(loc.instr.dsts[0])) {
                        if (loc.instr instanceof TacInstr.DirectCall) {
                            var entry =
((TacInstr.DirectCall)loc.instr).entry;
                            var instr = new TacInstr.DirectCall(entry);
                            func.replace(loc.instr, instr);
                        } else if (loc.instr instanceof
TacInstr.IndirectCall) {
                            var entry =
((TacInstr.IndirectCall)loc.instr).entry;
                            var instr = new
TacInstr.IndirectCall(entry);
                            func.replace(loc.instr, instr);
                        } else {
                            func.remove(loc.instr);
                            change = true;
                            changedLoc = loc;           //保存被删除的语
句
                        }
                    }
                }
                if (change) break;
            }
        }while(change);
    }
    return input;
}
```

实现结果

decaf文件

```
class Fibonacci {
    static int get(int index) {
        if (index < 2) {
            int x = 1;      //死代码
            int y = x+2;    //死代码
            int z = x*y;    //死代码
            return 1;
        }
        return get(index - 1) + get(index - 2);
    }
}
class Main {
    static void main() {
        int b = 1;                  //死代码
        int c = ReadInteger();    //带副作用的死代码
        int x = Fibonacci.get(3); //带副作用的死代码
    }
}
```

消除死代码之前的 `tac`

```
...省略一系列虚表和构造函数

#以下代码共47行
FUNCTION<Fibonacci.get>:
    _T1 = 2
    _T2 = 0
    _T3 = (_T1 == _T2)
    _T4 = (_T0 < _T1)
    if (_T4 == 0) branch _L1
    _T6 = 1
    _T5 = _T6
    _T8 = 2
    _T9 = 0
    _T10 = (_T8 == _T9)
    _T11 = (_T5 + _T8)
    _T7 = _T11
    _T13 = 0
    _T14 = (_T7 == _T13)
    _T15 = (_T5 * _T7)
    _T12 = _T15
    _T16 = 1
    return _T16
_L1:
    _T17 = 1
    _T18 = 0
    _T19 = (_T17 == _T18)
    _T20 = (_T0 - _T17)
    parm _T20
    _T21 = call FUNCTION<Fibonacci.get>
    _T22 = 2
    _T23 = 0
    _T24 = (_T22 == _T23)
    _T25 = (_T0 - _T22)
    parm _T25
    _T26 = call FUNCTION<Fibonacci.get>
    _T27 = 0
    _T28 = (_T26 == _T27)
    _T29 = (_T21 + _T26)
    return _T29

main:
    _T1 = 1
    _T0 = _T1
    _T3 = 3
    parm _T3
    _T4 = call FUNCTION<Fibonacci.get>
    _T2 = _T4
    parm _T2
    call _PrintInt
    return
```

消除死代码之后的`tac`

```
......省略一系列虚表和构造函数

#以下代码共26行
FUNCTION<Fibonacci.get>:
    _T1 = 2
    _T4 = (_T0 < _T1)
    if (_T4 == 0) branch _L1
    _T16 = 1
    return _T16
_L1:
    _T17 = 1
    _T20 = (_T0 - _T17)
    parm _T20
    _T21 = call FUNCTION<Fibonacci.get>
    _T22 = 2
    _T25 = (_T0 - _T22)
    parm _T25
    _T26 = call FUNCTION<Fibonacci.get>
    _T29 = (_T21 + _T26)
    return _T29

main:
    _T3 = 3
    parm _T3
    _T4 = call FUNCTION<Fibonacci.get>
    _T2 = _T4
    parm _T2
    call _PrintInt
    return
```

可以看到，`tac`中的死代码均已消除，`tac`代码行数从47行减少为26行