李嬰珩 2017050025

编译原理PA4 实验报告

2020年1月5日

编译原理Decaf实验

PA4

一、实验工作内容

本阶段主要是实现死代码的消除。活跃变量分析中，如果某个语句的Def不是空(说明存在赋值，即不存在副作用)，在不是call语句的情况下，如果它的赋值目标不在liveOut里的话，就可以直接消去。 这是因为如果赋值目标不在liveOut中，说明这个赋值始终没有被使用过，因此是可以被消去的。

二、实验流程中遇到的困难和解决办法

下面我将按照代码执行流程的顺序对我的工作进行阐述，并在过程中说明我遇到的困难和解决办法。

优化TAC代码的入口在Optimizer.java中，注意到框架中并没有进行任何的优化。

```
public TacProg transform(TacProg input) {
  var analyzer = new LivenessAnalyzer<>(); //建立新的LivenessAnalyzer
  var modifiedFuncs = new ArrayList<TacFunc>(); //经过优化的函数列表

  for(var func : input.funcs) {
    var builder = new CFGBuilder<>();
    var cfg = builder.buildFrom(new ArrayList<>(func.getInstrSeq())); //建立CFG
    analyzer.accept(cfg); //使用访问者模式对CFG进行访问
```

首先，我需要将输入的三地址码程序解析成CFG图并使用访问者模式通过LivenessAnalyzer 对 CFG 图进行访问和分析。

我在这里新建了一个函数列表 modifiedFuncs ，后续遍历 LivenessAnalyzer 分析过后的函数时，我将优化过后的函数添加到这个列表中。

下面转到 LivenessAnalyzer 中，

```
    private void analyzeLivenessForEachLocIn(BasicBlock<I> bb) {
        var liveOut = new TreeSet<>(bb.liveOut);
        var it = bb.backwardIterator();

        while (it.hasNext()) {
            boolean useful = false;
            var loc = it.next();
            loc.liveOut = new TreeSet<>(liveOut);

            if (loc.instr.dsts.length != 0) {
              for (var w : loc.instr.getWritten()) if (loc.liveOut.contains(w)) { useful = true; break; }
            } else { useful = true;}

            if (!useful) {
              loc.modified = true;
              if (loc.instr instanceof TacInstr.DirectCall || loc.instr instanceof TacInstr.IndirectCall) {
                loc.instr.dsts = new Temp[]{};
                loc.modified = false;
                useful = true;
              }
            }

            liveOut.removeAll(loc.instr.getWritten());
            if (useful) liveOut.addAll(loc.instr.getRead());
            loc.liveIn = new TreeSet<>(liveOut);
        }
        // assert liveIn == bb.liveIn
    }
```

这里的函数逆序遍历一个基本块的语句列表，我通过两方面特征来检查一条语句是否有用：

- 语句的 dst 的长度是否为 0
- getWritten 集合和该语句的 liveOut 集合交集是否为空

当检测到这个语句是无用语句时，我会在语句上进行标记（loc.modified = true），这个标记会通知 Optimizer 优化或删除这条语句。特别的，如果检测到这条语句是 call 语句时，我会直接在 LivenessAnalyzer 里面进行处理，将 call 语句的 dst 设置为空，并取消 modified 标记，Optimizer 不用在做处理了。

以上是对于语句的标记和处理，只进行这些工作是不够的，**接下来就是我遇到困难的地方：** 除了语句的删除，我需要对 liveOut 集合进行更新，否则会出现只优化一条语句的现象。在代码中，我通过语句是否有用（useful变量）来细化 liveOut 的更新（见最后三行），值得一题的是，由于上文中我在此直接修改了 call 语句的 dst，所以不用在特殊判别了。这样更新后的 liveOut 可以随着语句的逆序遍历不断上传，达到优化目标。

接下来回到 Optimizer，

```
    var modifiedFunc = new TacFunc(func.entry, func.numArgs);
    modifiedFunc.tempUsed = func.tempUsed;
    modifiedFunc.add(new TacInstr.Mark(func.entry));// 函数名

    for (var bb : cfg) {
      bb.label.ifPresent(b -> { modifiedFunc.add(new TacInstr.Mark(bb.label.get())); });
      for (var loc : bb) if (!loc.modified) modifiedFunc.add((TacInstr)loc.instr);
    }

    modifiedFuncs.add(modifiedFunc);
  }
  return new TacProg(input.vtables, modifiedFuncs);
  //return input;
}
```

在 LivenessAnalyzer 分析过后，我再此遍历所有语句块中的所有语句，对于有标记的语句，我全部都忽略掉，因为当前只是处理死代码消除工作，简单删除就可以。对于 call 语句，LivenessAnalyzer 已经做了处理，这里不用在此处理了。把优化后的函数添加到函数列表中，与原先输入的 vtable 一起构造新的 TacProg 返回，消除死代码的任务便完成了。

三、性能测试结果

无用赋值语句测试：

basic-basic.decaf

```
class Main {
    static void main() {
        int gpa1 = 4;
        int gpa2 = 3;
        int gpa3 = 3;
        int id = 2017050025;
        int x = 2;
        int y = 3;
        bool a = true;
        string s = "Hello DECAF 20170050025";
        Print(x,"\n");
        Print(y,"\n");
        Print(x+y,"\n");
        Print(x*y,"\n");
        Print(a,"\n");
        Print(s,"\n");
    }
}
```

basic-basic.output

```
VTABLE<Main>:
    NULL
    "Main"

FUNCTION<Main.new>:
    _T0 = 4
    parm _T0
    _T1 = call _Alloc
    _T2 = VTABLE<Main>
    *(_T1 + 0) = _T2
    return _T1

main:
    _T9 = 2
    _T8 = _T9
    _T11 = 3
    _T10 = _T11
    _T13 = 1
    _T12 = _T13
    _T15 = "Hello DECAF 20170050025"
    _T14 = _T15
    parm _T8
    call _PrintInt
    _T16 = "\n"
    parm _T16
    call _PrintString
    parm _T10
    call _PrintInt
    _T17 = "\n"
    parm _T17
    call _PrintString
    _T18 = (_T8 + _T10)
    parm _T18
```

可以看到，我新添加的 4 个 int 变量在后续都没有用到，在生成的 TAC 代码中没有出现相关的语句。

basic-basic.decaf

```
class Main {
    static void main() {
        int gpa1 = 4;
        int gpa2 = 3;
        int gpa3 = 3;
        int id = 2017050025;
        int x = gpa1;
        int y = 3;
        bool a = true;
        string s = "Hello DECAF 20170050025";
        Print(x,"\n");
        Print(y,"\n");
        Print(x+y,"\n");
        Print(x*y,"\n");
        Print(a,"\n");
        Print(s,"\n");
    }
}
```

basic-basic.output

```
VTABLE<Main>:
    NULL
    "Main"

FUNCTION<Main.new>:
    _T0 = 4
    parm _T0
    _T1 = call _Alloc
    _T2 = VTABLE<Main>
    *(_T1 + 0) = _T2
    return _T1

main:
    _T1 = 4
    _T0 = _T1
    _T8 = _T0
    _T10 = 3
    _T9 = _T10
    _T12 = 1
    _T11 = _T12
    _T14 = "Hello DECAF 20170050025"
    _T13 = _T14
    parm _T8
    call _PrintInt
    _T15 = "\n"
    parm _T15
    call _PrintString
    parm _T9
    call _PrintInt
    _T16 = "\n"
    parm _T16
```

接下来我更改了 decaf 代码中的第 7 行（int x = 2; => int x = gpa1;），生成的 TAC 代码也做出了相应的改变，结果是正确的。

Call 语句测试：

我在 basic-math.decaf 中做了一些改动，使得可以更清晰地了解关于 call 的优化情况。

basic-math.decaf

```
class Main {
    static void main() {
        int id = 2017050025;
        int date = 20200105;
        int result = Maths.max(id, date);
        int another = 74;
    }
}

class Maths {
    static int abs(int a) {
        if (a >= 0) {
            return a;
```

左边为没有做优化前生成的 TAC 右边为优化过后的 TAC：

```
main:
    _T1 = 2017050025
    _T0 = _T1
    _T3 = 20200105
    _T2 = _T3
    parm _T0
    parm _T2
    _T5 = call FUNCTION<Maths.max>
    _T4 = _T5
    _T7 = 74
    _T6 = _T7
    return
```

```
main:
    _T1 = 2017050025
    _T0 = _T1
    _T3 = 20200105
    _T2 = _T3
    parm _T0
    parm _T2
    _T5 = call FUNCTION<Maths.max>
    return
```

因为 simulator 的原因，关于 call 的代码部分只优化掉了一条，_T5 = call FUNCTION<Maths.max> 无法被优化成 call FUNCTION<Maths.max> 。

但其他部分已经优化完毕了。