编译原理PA3

死代码消除

实现流程

经过观察发现，框架中已经完整地实现了基本块的构造与后向数据流分析，我只需要利用现成的结果就行了。

死代码消除的判定是所有被赋值的变量都不能出现在该语句的`liveOut`中。这一属性通过如下方式访问：`CFG`里有若干个`BasicBlock`节点，每个节点有一系列语句，它们被连同`liveIn`、`liveOut`等属性包装成为`Loc`，只需要对这些`Loc`进行遍历，从而将满足要求的语句消除。消除的方式是将其同时从`BasicBlock`与`TacProg`中删除，这一操作可行性的保障在于包装后的`Loc`中包含的指令与`TacProg`中的指令指向同一个对象。

一遍消除是不能干净地消除所有死代码的。例如：

```
_T2 = _T1
_T3 = _T2
return _T0
```

第一遍处理时，只有第二句是死代码，然而实际上第一句也是多余的，只有在消除第二句之后才会“暴露”出来。我采取了双重措施：第一是从后往前消除，一边消除一边对各语句`liveOut`进行更新，这样效率较高，但是不能处理跨基本块的情形；第二是多次消除直至没有新删除的语句，这对于跨基本块的死代码同样有效。

优化效果展示

考虑如下代码：

```
class Main {
    static int f() {
        int i = 0;
        int j = 1;
        int k = i+j;
        return j;
    }
    static void main() {
        Print(f());
    }
}
```

优化重点在于函数`f`。不加优化的函数`f`对应TAC代码如下：

```
FUNCTION<Main.f>:
    _T1 = 0
    _T0 = _T1
    _T3 = 1
    _T2 = _T3
    _T5 = (_T0 + _T2)
    _T4 = _T5
    return _T2
```

优化后为：

```
FUNCTION<Main.f>:
    _T3 = 1
    _T2 = _T3
    return _T2
```

除了没有做常量传播，已经达到了最优。

如果复杂一些，将死代码放在分支中，可能会出现如下的Decaf代码：

```
class Main {
    static int f(int x) {
        int i = 0;
        int j = 1;
        int k = i+j;
        if (x == 0) {
            k = 256 + k;
        } else {
            k = 128 + k;
        }
        return j;
    }
    static void main() {
        Print(f(233));
    }
}
```

未经优化的TAC如下：

```
FUNCTION<Main.f>:
    _T2 = 0
    _T1 = _T2
    _T4 = 1
    _T3 = _T4
    _T6 = (_T1 + _T3)
    _T5 = _T6
    _T7 = 0
    _T8 = (_T0 == _T7)
    if (_T8 == 0) branch _L1
    _T9 = 256
    _T10 = (_T9 + _T5)
    _T5 = _T10
    branch _L2
_L1:
    _T11 = 128
```

```
    _T12 = (_T11 + _T5)
    _T5 = _T12
_L2:
    return _T3
```

优化后如下：

```
FUNCTION<Main.f>:
    _T4 = 1
    _T3 = _T4
    _T7 = 0
    _T8 = (_T0 == _T7)
    if (_T8 == 0) branch _L1
    branch _L2
_L1:
_L2:
    return _T3
```

这一优化结果就比较耐人寻味了。首先分支结构得到了保留；其次优化起了效果，但是并不彻底。进一步注意到，没有优化的变量都和分支转移的条件有关，因此与流程无关的代码都已经被删掉了，在能力范围内的优化仍然是“彻底”的。

复写传播

实现流程

由于框架中没有前向数据流处理的代码，我需要手写一个，而且我希望这个处理框架具有一定通用性，可以用于多种情况。这个类的名字叫`DefReachAnalysis`，实现了对流图进行迭代求解的过程，而将其中根据具体应用而定的部分抽象了出来，包括由`In`求`Out`的函数、交汇方式、计算一条语句对应的`Gen`和`Kill`集合、初始化边界。为了配合前向流分析记录不同类型的数据，我在`BasicBlock`、`Loc`中也为前向分析对应地添加了类型为`AnalysisInfo`的集合。

之后，我们具体地实现复写传播。定义`DefReachAnalysis`的派生类`CopyAnalysis`，构造其对应的记录信息类`CopyOptInfo`，继承自`AnalysisInfo`。然后对`DefReachAnalysis`中留空的函数进行实现：

- `computeKill`：把被赋值的变量放到集合中。为了简洁起见，不处理复写语句的全集，而是只记录被kill的复写语句包含哪些变量。
- `computeGen`：对于复写语句，把左右打包成`Pair`放进集合中。
- `initialize`：处理边界情况。由于没有处理全集，我们用一个仅包含一个特殊元素的集合表示全集，之后在`update`和`merge`中特殊处理。
- `update`：首先特判`in`是全集的情况；其他情况下，把满足以下条件的`in`集合复写语句删除：
    - 被kill掉元素出现在本语句右边；
    - 被赋值者与`gen`中的某个复写语句左边相同。

  最后，把`gen`并进来，得到结果。
- `merge`：特判全集的情况，其他情况下把两个集合取交集。

最后，处理出来每条语句前的复写集合，我们对每条语句的源变量进行更新。由于框架的限制，我采用了构造新语句替换掉的办法，并且不得不对所有使用了源变量的语句进行分类讨论——它们在父类和子类中重复定义了这一成员。我们仍然需要对代码进行反复的迭代优化，以达到最优化效果。

优化效果展示

我们用一个变量交换的例子进行说明：

```
class Main {
    static void main() {
        int a = 1;
        int b = 2;
        int c = a;
        a = b;
        b = c;
        Print(a, b);
    }
}
```

这个例子在未优化的时候TAC代码如下：

```
main:
    _T1 = 1
    _T0 = _T1
    _T3 = 2
    _T2 = _T3
    _T4 = _T0
    _T0 = _T2
    _T2 = _T4
    parm _T0
    call _PrintInt
    parm _T2
    call _PrintInt
    return
```

出现了多处复写语句，但是有的可以起优化效果，有的却不能（比如`_T4 = _T0`）。经过正确复写传递优化后，效果如下：

```
main:
    _T1 = 1
    _T0 = _T1
    _T3 = 2
    _T2 = _T3
    _T4 = _T1
    _T0 = _T3
    _T2 = _T1
    parm _T3
    call _PrintInt
    parm _T1
    call _PrintInt
    return
```

结合着之前的死代码优化，就可以优化到极致：

```
main:
    _T1 = 1
    _T3 = 2
    parm _T3
    call _PrintInt
    parm _T1
    call _PrintInt
    return
```

性能测试

测试点	未优化	死代码优化	复写+死代码
`mandelbrot.decaf`	5565242	5565242	5487502
`rbtree.decaf`	2793358	2715602	2632582
`sort.decaf`	618314	618314	616824

可见死代码优化需要结合其他优化方法一起使用才能达到较优效果。