Decaf_PA3 实验报告

计 31 班 刘智峰 2013011427

【实验内容】

在 PA2 中，我们已经完成了对输入程序的语义分析工作，此时的输入程序必定是有明确的语义而且不具有不符合语言规范的语义错误的，在接下来的 PA3 中，我们将对该输入程序进行翻译，把使用带属性修饰的抽象语法树（decorated AST）来表示的输入程序翻译成适合后期处理的另一种中间表示方式。

在 Decaf 编译器中，为简单起见，我们在 AST 以外只涉及一种中间表示，这种中间表示叫做三地址码（Three Address Code, TAC），是一种比较接近汇编语言的表示方式。PA3 中我们需要把 AST 表示的程序翻译为跟它在语义上等价的 TAC 中间表示，并在合适的地方加入诸如检查数组访问越界、数组大小非法等运行时错误的内容。在 TAC 表示的基础上，经过临时变量的活性分析（PA4 内容）、生成汇编代码（在 PA4 中直接提供）以后，整个编译过程便告完成。

【实验理解】

刚开始做的时候，没有充分理解本次实验新增的实验框架，想像之前两次实验一样通过查看 ouput 和 result 文件夹内的样例以及仿照已有的代码，来进行实验。通过这种方法，我只仿照数组大小检查的函数实现了除 0 检查，其他的就懵了……

于是静下心，好好理解了实验框架。在理解的过程中，我询问了计 33 班的郭志芃同学，他也十分耐心地给我讲解了一遍大概的框架，在此对他表示感谢！

下面以实验框架中自带的 visitIf(Tree.if ifStmt)来阐述我的理解。

```
public void visitIf(Tree.If ifStmt) {
    ifStmt.condition.accept(this);
    if (ifStmt.falseBranch != null) {  // 有else
        Label falseLabel = Label.createLabel();
        tr.genBeqz(ifStmt.condition.val, falseLabel);
        ifStmt.trueBranch.accept(this);
        Label exit = Label.createLabel();
        tr.genBranch(exit);
        tr.genMark(falseLabel);
        ifStmt.falseBranch.accept(this);
        tr.genMark(exit);
    } else if (ifStmt.trueBranch != null) { // 没有else 且if内有代码
        Label exit = Label.createLabel();
        tr.genBeqz(ifStmt.condition.val, exit);
        if (ifStmt.trueBranch != null) {
            ifStmt.trueBranch.accept(this);
        }
        tr.genMark(exit);
    }
}
```

首先，第三次实验为了维护 TAC 结构，引入了 Temp、Label 等数据类型。下面先观察 visitIf 中的如果没有 else 的情况，即 else if(ifStmt.trueBranch !=null) 以下。假设此时的情况为：

If(A){

doSomething;

}

Others；

首先，这里新打了一个表 exit，然后用 genBeqz 函数判断 if 的条件的值 ifStmt.condition.val 是否为 0，即 A 是否为 0。如果为 0，则跳转到刚才打的表 exit 处，否则就继续往下执行 doSomething 的内容。那么跳转到 exit 表，到底跳转到哪呢？可以发现，在 if 语句结束后，visitIf 中有一句 genMark(exit)，我理解为，在 if 结束后的地方给 exit 打了一个标记，这样如果跳转到 exit，即跳转到 if 执行完后的地方，即此处的 Others。

再看 visitIf 中有 else 的情况，假设此时的情况为：

```
If(A){
    doSomething;
}else{
    doSomething2;
}
Others;
```

同样的，这里打了一个表 falseLabel，然后用 genBeqz 函数判断 if 的条件的值 ifStmt.condition.val 是否为 0，即 A 是否为 0。如果为 0，则跳到 falseLabel。那么 falseLabel 到底在哪呢？可以发现，在 else 的语句块被 accept 之前，即 ifStmt.falseBranch.accept(this)之前有 genMark(falseLabel) ,说明 falseLabel 的标记在 else 语句块之前，跳转到 falseLabel 即跳转去执行 else 语句。如果不跳转，则顺序执行 if 内的语句，即 doSomething。在这，又打了一个表 exit，同时执行完 if 后，用 genBranch 跳到了这个表处。同样可以发现，这个表在 if-else 语句块结束的地方被 Mark 了，即在 Others 处被 Mark，所以如果 if 的判断条件正确，则执行完相应的语句后就退出了 if-else 语句块，继续执行后面的语句。

有了上述理解，对实验的大体框架就有了大概的认识，对于新增语法的支持也就不那么难做了。

【实验一】

1) 实验描述：检测“除零”非法这种运行时错误。

2) 实现思路：仿照判断数组长度是否合法的 genCheckArraySize 函数来写即

可。具体思路就是检测除法和取模操作的第二个操作数是否为 0，如果为 0 则打 result 中除零错误检测文件中的语句 ,并跳转到 exit 表处。如果不是 0，则直接跳转到 exit 表处。即，此函数的关键就是 genEqu(src,gemLoadImm(0))。

3) 实现说明：

有了思路，再参考 genCheckArraySize 函数，实现起来就很简单了。

```
/*
 * divisionbyzero1 2
 */
public void genCheckDivisionZero(Temp src){
    Label exit = Label.createLabel();
    Temp cond = genEqu(src, genLoadImm4(0));
    genBeqz(cond, exit);
    Temp msg = genLoadStrConst(RuntimeError.DIVISION_BY_ZERO);
    genParm(msg);
    genIntrinsicCall(Intrinsic.PRINT_STRING);
    genIntrinsicCall(Intrinsic.HALT);
    genMark(exit);
}
```

实现了此函数后，再在原来的 genDiv 和 genMod 函数中引用，对第二个操作数进行判断即可。

【实验二】

1） 实验描述：实现自增、自减单操作算子的 TAC 代码生成。自增、自减单操作算子形如 i++,++i,i--,--i，其语义解释与 C 语言中一致。

2）实现思路：++、--运算符，说到底就是一个等式，a++、++a，即 a=a+1，唯一的区别就是之后加还是立即加。同样的，a--、--a，即 a=a-1，唯一的区别就是之后减还是立即减。因此，只需要仿照原有的处理等式的函数 visitAssign 就能轻松实现++、--功能。由于在实验 1、实验 2 中我将++、--看做是 unary 一元运算符，所以我直接在 visitUnary 中实现了++、--操

作。

3） 实现说明：

a++：

```
case Tree.POSTINC: //后++
    expr.val = tr.genAdd(expr.expr.val, tr.genLoadImm4(0));
    Temp realAns = tr.genAdd(expr.expr.val, tr.genLoadImm4(1));
    expr.expr.accept(this);
    switch (((Tree.LValue)expr.expr).lvKind) {
        case ARRAY_ELEMENT:
            Tree.Indexed arrayRef = (Tree.Indexed) expr.expr;
            Temp esz = tr.genLoadImm4(OffsetCounter.WORD_SIZE);
            Temp t = tr.genMul(arrayRef.index.val, esz);
            Temp base = tr.genAdd(arrayRef.array.val, t);
            tr.genStore(realAns, base, 0);
            break;
        case MEMBER_VAR:
            Tree.Ident varRef = (Tree.Ident) expr.expr;
            tr.genStore(realAns, varRef.owner.val, varRef.symbol
                    .getOffset());
            break;
        case PARAM_VAR:
        case LOCAL_VAR:
            tr.genAssign(((Tree.Ident) expr.expr).symbol.getTemp(),
                    realAns);
            break;
    }
```

++a：

```
case Tree.PREINC: // 前++
    expr.val = tr.genAdd(expr.expr.val, tr.genLoadImm4(1));
    Temp realAns2 = tr.genAdd(expr.expr.val, tr.genLoadImm4(1));
    expr.expr.accept(this);
    switch (((Tree.LValue)expr.expr).lvKind) {
        case ARRAY_ELEMENT:
            Tree.Indexed arrayRef = (Tree.Indexed) expr.expr;
            Temp esz = tr.genLoadImm4(OffsetCounter.WORD_SIZE);
            Temp t = tr.genMul(arrayRef.index.val, esz);
            Temp base = tr.genAdd(arrayRef.array.val, t);
            tr.genStore(realAns2, base, 0);
            break;
        case MEMBER_VAR:
            Tree.Ident varRef = (Tree.Ident) expr.expr;
            tr.genStore(realAns2, varRef.owner.val, varRef.symbol
                    .getOffset());
            break;
        case PARAM_VAR:
        case LOCAL_VAR:
            tr.genAssign(((Tree.Ident) expr.expr).symbol.getTemp(),
                    realAns2);
            break;
    }
    break;
```

a--：

```
case Tree.POSTDEC: //后--
    expr.val = tr.genSub(expr.expr.val, tr.genLoadImm4(0));
    Temp realAns3 = tr.genSub(expr.expr.val, tr.genLoadImm4(1));
    expr.expr.accept(this);
    switch (((Tree.LValue)expr.expr).lvKind) {
        case ARRAY_ELEMENT:
            Tree.Indexed arrayRef = (Tree.Indexed) expr.expr;
            Temp esz = tr.genLoadImm4(OffsetCounter.WORD_SIZE);
            Temp t = tr.genMul(arrayRef.index.val, esz);
            Temp base = tr.genAdd(arrayRef.array.val, t);
            tr.genStore(realAns3, base, 0);
            break;
        case MEMBER_VAR:
            Tree.Ident varRef = (Tree.Ident) expr.expr;
            tr.genStore(realAns3, varRef.owner.val, varRef.symbol
                    .getOffset());
            break;
        case PARAM_VAR:
        case LOCAL_VAR:
            tr.genAssign(((Tree.Ident) expr.expr).symbol.getTemp(),
                    realAns3);
            break;
    }
```

--a：

```
case Tree.PREDEC: // 前--
    expr.val = tr.genSub(expr.expr.val, tr.genLoadImm4(1));
    Temp realAns4 = tr.genSub(expr.expr.val, tr.genLoadImm4(1));
    expr.expr.accept(this);
    switch (((Tree.LValue)expr.expr).lvKind) {
        case ARRAY_ELEMENT:
            Tree.Indexed arrayRef = (Tree.Indexed) expr.expr;
            Temp esz = tr.genLoadImm4(OffsetCounter.WORD_SIZE);
            Temp t = tr.genMul(arrayRef.index.val, esz);
            Temp base = tr.genAdd(arrayRef.array.val, t);
            tr.genStore(realAns4, base, 0);
            break;
        case MEMBER_VAR:
            Tree.Ident varRef = (Tree.Ident) expr.expr;
            tr.genStore(realAns4, varRef.owner.val, varRef.symbol
                    .getOffset());
            break;
        case PARAM_VAR:
        case LOCAL_VAR:
            tr.genAssign(((Tree.Ident) expr.expr).symbol.getTemp(),
                    realAns4);
            break;
    }
    break;
```

【实验三】

1) 问题描述：实现三操作数算子？：的 TAC 代码生成。三操作数算子形如 A？B：C。

2) 实现思路： “A ? B : C” 对应规则为 Expr1 ? Expr2 : Expr3。此处只需要根据 Expr1 这个 bool 式的结果进行跳转即可。新声明一个表 falseLabel 并在 Expr3 之前 Mark，然后判断 Expr1 的结果。如果结果为 false，则跳到 falseLabel 执行 Expr3 ,否则继续执行 Expr2 ,并声明一个表 exit ,在 Expr2 执行完后用 genBranch 跳转，并在函数的最后 Mark 即可。

3) 实现说明：

```
/*
 * ?_: 仿照visitif的写法
 *
 */
@Override
public void visitQuestionAndColon(Tree.QuestionAndColon questionAndColon){
    questionAndColon.left.accept(this);
    questionAndColon.val = tr.genLoadImm4(0); // 必须要给表达式初值 否则会出错
    Label falseLabel = Label.createLabel();
    tr.genBeqz(questionAndColon.left.val, falseLabel); // choose

    questionAndColon.middle.accept(this);
    tr.genAssign(questionAndColon.val, questionAndColon.middle.val); // 表达式赋值
    Label exit = Label.createLabel();
    tr.genBranch(exit);
    tr.genMark(falseLabel);

    questionAndColon.right.accept(this);
    tr.genAssign(questionAndColon.val, questionAndColon.right.val); // 表达式赋值
    tr.genMark(exit);
}
```

【实验四】

1）问题描述：实现反射运算 numinstances 的 TAC 代码生成。反射运算 numinstances 形如 numinstances（A），其语义解释为：计算结果返回类 A 当前实例对象的个数。

2）实现思路：numinstances 的实现参考了网络学堂的讨论中杨俊晔同学的思路：为 Class 类增加一个 Temp 类型的变量 numInstances，并且在第一趟 visitClassDef 的时候为其初始化为 0。接着在第二趟的 visitNewClass 每次

调用都把 numInstances+1。最后输出 numInstances 的值即可。


回复文章如下（共4）

回复人	杨俊晔	回复时间	2015-12-04 22:40
正文	我是这么做的：为Class类增加一个Temp类型的变量numInstances，并且在第一趟visitClassDef的时候为其初始化为0。接着在第二趟的visitNewClass写一个每次调用都把numInstances+1的代码。 这样写感觉还是挺常规的。		
附件	无附件		

3）实现说明：

首先在 Class.java 文件中的 Class 中新增加了 Temp 类型的 numinstances，并初始化为 0。初始化需要调用 creatTempI4 函数。

```
public Temp numinstances;  // numinstances 记录被实例化了几次

 public Class(String name, String parentName, Location location) {
     this.name = name;
     this.parentName = parentName;
     this.location = location;
     this.order = -1;
     this.check = false;
     this.numNonStaticFunc = -1;
     this.numVar = -1;
     this.associatedScope = new ClassScope(this);
     this.numinstances = Temp.createTempI4();  // 初始化
     }
```

然后在第二次扫描的过程中，每一次 visitNewClass，都需要将其中的 numinstances 加 1。由于框架自带的 genAdd 函数返回的是一个新的 Temp，而在统计一个类被 new 了多少次时需要返回当前这种类而不是新的 Temp，所以我新定义了一种加法，返回的 Temp 为第一个参数。

```
//numinstances
public void genAddToOri(Temp src1 , Temp src2){
    append(Tac.genAdd(src1, src1, src2));
}
```

然后在 visitNewClass 中调用此函数，对 numinstances 加 1：

```
@Override
public void visitNewClass(Tree.NewClass newClass) {
    newClass.val = tr.genDirectCall(newClass.symbol.getNewFuncLabel(),
            BaseType.INT);

    tr.genAddToOri(newClass.symbol.numinstances, tr.genLoadImm4(1)); // numinstances
}
```

最后解析 numinstances(C)时，直接输出 C 的 numinstances 即可。

```
/*
 * numinstances
 */
@Override
public void visitNumTest(Tree.NumTest numTest){
    numTest.val = numTest.symbol.numinstances;
}
```

【实验五、六】

1) 问题描述：实现串行条件卫士语句和串行循环卫士语句的 TAC 码生成。语意见实验说明。

2）实现思路：在理解了visitIf函数后，串行条件卫士语句和串行循环卫士语句就很好写了。对于串行条件卫士语句 if E1:S1 ||| E2:S2 ||| … ||| En:Sn fi，即：

if (E1 == 0) then 跳转到Label1

S1 跳转到Exit

Label1:

If (E2 == 0) then 跳转到Label2

S2 跳转到Exit

Label2

……

if (En == 0) then 跳转到Labeln

Sn 跳转到Exit

Exit：

串行条件卫士之后的语句

按照这个思路，只需要对visitIf进行稍微修改即可。串行循环卫士语句也类似，只有两点区别：一是需要支持break语句，这点参照visitWhile即可。二是需

要引入loop表，同样参照visitWhile即可。

3）实现说明：

串行条件卫士语句：

```
/*
 * GuardedIfStmt
 */
@Override
public void visitGuardedIFStmt(Tree.GuardedIFStmt guardedIFStmt){
    Label exit = Label.createLabel();
    for(Tree.GuardedES i : guardedIFStmt.guardedES){
        i.expr.accept(this);

        Label falseLabel = Label.createLabel();
        tr.genBeqz(i.expr.val, falseLabel);
        i.tree.accept(this);
        tr.genBranch(exit);
        tr.genMark(falseLabel);
        tr.genMark(exit);
    }
}
```

串行循环卫士语句:

```
/*
 * GuardedDOStmt
 */
@Override
public void visitGuardedDOStmt(Tree.GuardedDOStmt guardedDOStmt){
    Label loop = Label.createLabel();
    tr.genMark(loop);
    Label exit = Label.createLabel();
    loopExits.push(exit);
    for(Tree.GuardedES i : guardedDOStmt.guardedES){
        Label exit1 = Label.createLabel();
        i.expr.accept(this);
        tr.genBeqz(i.expr.val, exit1);
        if (i.tree != null) {
            i.tree.accept(this);
        }
        tr.genBranch(loop);
        tr.genMark(exit1);
    }
    loopExits.pop();
    tr.genMark(exit);
}
```

【实验总结】

相比第一、第二阶段的实验，本次实验难度较大，主要是新增加的 TAC 框

架不好理解，不理解框架的话照着框架代码写也写不对。所以理解框架真的很重要，会起到事半功倍的作用。在此再次感谢计 33 班的郭志芃同学给我讲解实验框架，讲完真是豁然开朗，做起来就很快了！

第一、第二阶段实验都只花了半天时间，而此次实验零零碎碎花了三天。通过本次实验，我对 TAC 中间代码的生成有了更深层次的理解，对 decaf 实验也有了更多的认识。感谢老师、助教和郭志芃同学给我的帮助！