PA3 report

计61 潘庆霖 2016011388

1、Scopy特性

实现要点：

- 在Parse.y中的语法形式：

```
OCStmt  : SCOPY '(' IDENTIFIER ',' Expr ')'
          {
            $$.stmt = new Tree.ScopyClass($3.ident,$5.expr,$1.loc,$3.loc);
          }
        ;
```

- 在TypeCheck.java中对应的类型检查，重写visitScopyClass
- 在TransPass2.java中，重写visitScopyClass,代码如下：

```
@Override
public void visitScopyClass(ScopyClass that) {
        that.ident.accept(this);
        that.expr.accept(this);
        Class c = ((ClassType)that.expr.type).getSymbol();  //获取用于赋值的变量的类型
        Temp t = tr.genDirectCall(c.getNewFuncLabel(),  //为接受浅复制的标识符分配temp
                BaseType.INT);
        t.sym = that.ident.symbol;
        that.ident.symbol.setTemp(t);
        that.expr.val.size = c.getSize();   //设置对象的大小，用于genScopy中判断对象的规
模
        tr.genScopy(that.expr.val,that.ident.symbol.getTemp());
}
```

- 在Translater.java中，实现对应的浅复制函数:

```
//for scopy
public void genScopy(Temp src, Temp dst){
    int time = src.size / OffsetCounter.WORD_SIZE - 1;  //计算得到类变量的个数
    if (time != 0) {    //如果该类中没有类变量，则浅复制不需要做任何事情
        for (int i = 0; i < time; i++) {    //如果有类变量，则按字节逐字节复制
            genStore(genLoad(src, OffsetCounter.WORD_SIZE * (i + 1)),
                dst, OffsetCounter.WORD_SIZE * (i + 1));
        }
    }
}
```

实现思路：

要实现对象的浅复制，要求对scopy的两个参数都分配了对应的Temp标号，两个标号分别指代两片内存。并且，在PA1、PA2中的语法树构建和类型检查，保证了两个Temp指代的是同样类型的对象。

在本次框架中，对象的表示是通过一个Temp对象来在tac中表示，Temp中存储一个32位的地址，该地址指向的内存中，第一个单元存储虚函数表，接下去的单元依次存储继承的类变量和这一个类中定义的类变量。

依据上面两点，可以通过下面两步实现：

1、保证scopy（a,b)中a,b都可以拿到对应的Temp，对应的Class符号表项（用来获取对象在内存中的大小）。

2、对于对应的TempA,TempB;生成对应的浅复制tac码，从对应内存区域的第二个单元开始，逐个单元进行Load、Store。

2、Sealed特性的实现

实现要点：

- 在Parse.y中对应的语法形式：

```
ClassDef   :    CLASS IDENTIFIER ExtendsClause '{' FieldList '}'
            {
                $$.cdef = new Tree.ClassDef($2.ident, $3.ident, $5.flist, $1.loc);
            }
           |    SEALED CLASS IDENTIFIER ExtendsClause '{' FieldList '}'
            {
                $$.cdef = new Tree.ClassDef($3.ident, $4.ident, $6.flist, $1.loc);
                $$.cdef.isSealed = true;
            }
           ;
```

- 在BuildSym.java中进行相应的检查，在第一次遍历类定义的时候，记录带有Sealed标记的类的类名。第二遍检查类的继承关系的时候，如果发现当前检查的类的超类带有Sealed标记，就报错。
- 在Translate过程中，不需要对这个特性再进行对应实现。

实现思路

Sealed特性主要和类继承有关，这个特性的实现可以通过直接在符号表的构建阶段进行检查来解决。

3、Guarded条件卫士的实现

实现要点：

- 在Parse.y中对应的语法形式：

```
GuardedStmt     :   IF '{' IfBranches '}'
```

```
                    {
                        $$.stmt = new Tree.GuardedStmt($3.elist,$1.loc);
                    }
                ;
IfBranches      :   IfBranches GUARD IfSubStmt
                    {
                        $$.elist = $1.elist;
                        $$.elist.add($3.expr);
                    }
                |   IfSubStmt
                    {
                        $$.elist = new ArrayList<Expr>();
                        $$.elist.add($1.expr);
                    }
                |   /* empty */
                    {
                        $$ = new SemValue();
                    }
                ;

IfSubStmt       :   Expr ':' Stmt
                    {
                        $$.expr = new Tree.IfSubStmt($1.expr,$3.stmt,$1.loc);
                    }
                ;
```

- 在BuildSym.java中进行相应的检查，如果IfSubStmt中的trueBranch是一个代码块，则初始化它对应的LocalScope，具体代码如下：

```
@Override
public void visitIfSubStmt(Tree.IfSubStmt that) {
    //如果对应的执行语句是代码块，需要初始化对应的LocalScope
    if(that.trueBranch.tag == Tree.BLOCK){
        Tree.Block block = (Tree.Block)(that.trueBranch);
        block.associatedScope = new LocalScope(block);
        table.open(block.associatedScope);
        for(Tree s : block.block){
            s.accept(this);
        }
        table.close();
    }
}

@Override
public void visitIfBranches(Tree.GuardedStmt that) {
    for(Tree.Expr e: that.branches){
        e.accept(this);
    }
}
```

- 在TypeCheck.java中，检查condition的表达式类型是否为bool类型

```
//TypeCheck.java
@Override
public void visitIfBranches(Tree.GuardedStmt that) {
    for(Tree.Expr expr : that.branches){
        expr.accept(this);
    }
}

@Override
public void visitIfSubStmt(Tree.IfSubStmt that) {
    that.condition.accept(this);
    if(!(that.condition.type == BaseType.BOOL)){    //检查迭代判断的表达式是否布尔类型
        issueError(new BadTestExpr(that.condition.loc));
    }
    that.trueBranch.accept(this);
}
```

- 在Translate.java阶段，主要生成根据condition表达式的类型，来判断是否跳转到trueBranch的对应tac代码块，具体代码如下：

```
//Translate
@Override
public void visitIfBranches(GuardedStmt that) {
    for(Expr e: that.branches){
        e.accept(this);
    }
}

@Override
public void visitIfSubStmt(IfSubStmt that) {
    that.condition.accept(this);    //首先分配codition的对应temp，生成对应的tac码
    Label exit = Label.createLabel();   //设置一个Label用于辅助逻辑判断
    tr.genBeqz(that.condition.val, exit);   //condition条件如果为假则不执行，直接结束
    if (that.trueBranch != null) {  //生成trueBranch中对应需要执行的代码的tac码
        that.trueBranch.accept(this);
    }
    tr.genMark(exit);
}
```

4、var自动类型推导的实现

实现要点：

- 在Parse.y中对应的语法形式：

```
LValue          :   VAR IDENTIFIER
                    {
                        $$.lvalue = new Tree.VarIdent($2.ident, $2.loc);
                    }
                ;
```

- 类似var a = b；的语句在a未被定义过的情况下应该当成一个变量声明来看待，所以应该在BuildSym.java中有对应的符号表处理，具体代码如下：

```
//BuildSym.java
@Override
public void visitVarIdent(Tree.VarIdent that) {
    Variable v = new Variable(that.name, BaseType.VAR,
            that.getLocation());
    Symbol sym = table.lookup(that.name, true);
    //解决变量重复定义的问题
    if (sym != null) {
        if (table.getCurrentScope().equals(sym.getScope())) {
            issueError(new DeclConflictError(v.getLocation(), v.getName(),
                    sym.getLocation()));
            that.type = BaseType.ERROR;
        } else if (sym.getScope().isFormalScope() &&
table.getCurrentScope().isLocalScope()) {
            issueError(new DeclConflictError(v.getLocation(), v.getName(),
                    sym.getLocation()));
            that.type = BaseType.ERROR;
        } else {
            table.declare(v);
        }
    } else {
        table.declare(v);
    }
    that.type = BaseType.VAR;
    that.symbol = v;
}
```

- 在类型检查阶段，可以确定var类型的变量的实际类型，这时应当修改对应的符号表项，并且要注意，修改符号表项的类型，与重新插入一个符号表项来覆盖之前同名的符号表项，有本质差别。（后者会导致，同一个变量名，一些语法节点绑定的符号表项和其他语法节点绑定的符号表项不一致）具体代码如下：

```
//TypeCheck.java
@Override
public void visitAssign(Tree.Assign assign) {
    assign.left.accept(this);
    assign.expr.accept(this);
    if(assign.left.type.equal(BaseType.VAR)){
        Variable v = (Variable)table.lookup(assign.left.name,true);
        v.setType(assign.expr.type);    //这里的setType是用来修改符号表项的Type的，自行添
加的接口。
        assign.left.type = assign.expr.type;
        assign.left.lvKind = Tree.LValue.Kind.LOCAL_VAR;    //设置为局部变量
```

```
    }

    if (!assign.left.type.equal(BaseType.ERROR)
            && (assign.left.type.isFuncType() || !assign.expr.type
                    .compatible(assign.left.type))) {
        issueError(new IncompatBinOpError(assign.getLocation(),
                assign.left.type.toString(), "=", assign.expr.type
                        .toString()));
    }
}
```

- 在Translate阶段，var类型定义的变量可能是类变量，也可能是方法中的局部变量。因此，首先应该对变量定义进行充分的检查。在TransPass1.java中，实现对var类型的类变量的对应检测，以便让类有一个正确的内存大小。在TranPass2.java中，要对var类型的局部变量分配Temp标号，并复制等号右边表达式对应的Temp，达到赋值的效果。具体代码如下：

```
//TransPass1.java
@Override
public void visitVarIdent(VarIdent that) {
    vars.add(that.symbol);
    objectSize += OffsetCounter.WORD_SIZE;
}
@Override
public void visitAssign(Assign that) {
    that.left.accept(this);
}
```

```
//TransPass2.java
@Override
public void visitAssign(Tree.Assign assign) {
    assign.left.accept(this);
    assign.expr.accept(this);
    switch (assign.left.lvKind) {
    case ARRAY_ELEMENT: //数组类型
        Tree.Indexed arrayRef = (Tree.Indexed) assign.left;
        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(assign.expr.val, base, 0);
        break;
    case MEMBER_VAR:    //成员变量
        Tree.Ident varRef = (Tree.Ident) assign.left;
        tr.genStore(assign.expr.val, varRef.owner.val, varRef.symbol
                .getOffset());
        break;
    case PARAM_VAR:     //参量
    case LOCAL_VAR:     //局部变量
        if(!assign.left.isVarIdentifier()){//这里增加了Tree.LValue中增加了判断接口
            tr.genAssign(((Tree.Ident) assign.left).symbol.getTemp(),
                assign.expr.val);
        }else{
            tr.genAssign(((Tree.VarIdent) assign.left).symbol.getTemp(),
```

```
                    assign.expr.val);
            }
            break;
        }
    }

    @Override
    public void visitVarIdent(VarIdent that) {
        Temp t = Temp.createTempI4();
        t.sym = that.symbol;
        that.symbol.setTemp(t);
    }
```

5-1 数组初始化常量表达式 E%%n

实现要点：

- 在Parse.y中对应的语法形式：

```
Expr            :   Expr MOMO Expr
                    {
                        $$.expr = new Tree.ArrayRepeat($1.expr,$3.expr,$1.loc);
                    }
                ;
```

- 语法要求E是可以推导类型的表达式，所以这一特性在符号表构造阶段无需对应实现。
- 符号检查阶段主要检查：E的类型是否是基本类型或者类类型、n是否是整数类型。
- 在Translate阶段。没有变量定义和方法声明的相关，所以只需要在TransPass2.java中有对应的实现。具体思路：
    - 在内存中分配n+1个单元大小的区域用于存储数组
    - 为每个单元进行赋值（如果是类类型，则实例化一个新的对象，浅复制之后再将对应Temp赋值给该单元）
    - 在内存中对应位置（第一个单元）保存数组的长度。
    - 返回对应的Temp，给当前的语法节点的val赋值。

```
//TransPass2.java
@Override
public void visitArrayRepeat(ArrayRepeat that) {
    that.expr.accept(this);
    that.value.accept(this);
    Class c = null;
    if(that.expr.type.isBaseType()){
        that.expr.val.size = 4;
    }else{
        c = ((ClassType)that.expr.type).getSymbol();
        that.expr.val.size = c.getSize();
    }
    that.val = tr.genNewArrayWithMOMO(that.expr.val, that.value.val,c);
```

```
}
```

```
//Translater.java
public void genCheckMOMOArraySize(Temp size) {
    Label exit = Label.createLabel();
    Temp cond = genLes(size, genLoadImm4(0));
    genBeqz(cond, exit);
    Temp msg = genLoadStrConst(RuntimeError.MOMO_NEGATIVEA_ARR_SIZE);
    genParm(msg);
    genIntrinsicCall(Intrinsic.PRINT_STRING);
    genIntrinsicCall(Intrinsic.HALT);
    genMark(exit);
}

//for momo
public Temp genNewArrayWithMOMO(Temp e, Temp n, Class c) {
    genCheckMOMOArraySize(n);

    Temp unit = genLoadImm4(OffsetCounter.WORD_SIZE);
    Temp size = genAdd(unit, genMul(unit, n));

    genParm(size);
    Temp obj = genIntrinsicCall(Intrinsic.ALLOCATE);
    genStore(n, obj, 0);
    Label loop = Label.createLabel();
    Label exit = Label.createLabel();
    Temp zero = genLoadImm4(0);
    append(Tac.genAdd(obj, obj, size));
    genMark(loop);
    append(Tac.genSub(size, size, unit));
    Temp cond = genLes(zero, size);
    genBeqz(cond, exit);
    append(Tac.genSub(obj, obj, unit));
    if(e.size < 5){
        genStore(e, obj, 0);
    }else{
        Temp newObj = genDirectCall(c.getNewFuncLabel(),
                BaseType.INT);
        genStore(newObj, obj, 0);
        genScopy(e, newObj);
    }
    genBranch(loop);
    genMark(exit);
    return obj;
}
```

5-2 数组下标动态访问表达式 E[E1] default E`

实现要点：

- Parse.y中对应的语法形式：

```
Expr     :  Expr '[' Expr ']' DEFAULT Expr
            {
               $$.expr = new Tree.ArrayDefault($1.expr,$3.expr,$6.expr,$3.loc);
            }
          ;
```

- 没有涉及变量的定义，所以BuildSym.java中无需进行对应实现。
- TypeCheck.java中，进行对应的类型检查：
  - E的类型是否是数组类型
  - E1表达式的类型是否是整数类型
  - 判断E`的类型是否和E中的元素类型一致
- 在TransPass2.java中，生成对应的运行时检查的tac码。即能够先判断下标E1是否符合下标的条件，如果不符合直接返回E`对应的Temp，否则获取E数组中对应的单元的Temp，并返回之。具体代码如下：

```
//TransPass2.java
@Override
public void visitArrayDefault(ArrayDefault that) {
    that.arr.accept(this);
    that.idx.accept(this);
    that.defaultIdx.accept(this);

    Label idx_e = Label.createLabel();
    Label exit = Label.createLabel();

    Temp length = tr.genLoad(that.arr.val, -OffsetCounter.WORD_SIZE);   //获取数组长
度
    Temp cond = tr.genLes(that.idx.val, length);    // idx < length
    Temp res = Temp.createTempI4(); //用来保存返回的结果
    tr.genBeqz(cond, idx_e);
    Temp cond1 = tr.genLes(that.idx.val, tr.genLoadImm4(0));    //idx < 0
    tr.genBnez(cond1, idx_e);

    Temp esz = tr.genLoadImm4(OffsetCounter.WORD_SIZE);
    Temp t = tr.genMul(that.idx.val, esz);
    Temp base = tr.genAdd(that.arr.val, t);
    tr.genAssign(res, tr.genLoad(base, 0));
    tr.genBranch(exit);

    tr.genMark(idx_e);
    tr.genAssign(res, that.defaultIdx.val);

    tr.genMark(exit);
    that.val = res;
}
```

5-3 数组迭代语句 foreach 特性

实现要点：

- 在Parse.y中的语法形式：

```
ForeachStmt : FOREACH '(' BoundVariable IN Expr WHILE Expr ')' StmtBlock
              {
               $$.stmt = new Tree.ForeachStmt($3.vdef, $5.expr, $7.expr, $9.slist,
  $9.stmt.loc, $1.loc);
              }
            |   FOREACH '(' BoundVariable IN Expr ')' StmtBlock
              {
                $$.stmt = new Tree.ForeachStmt($3.vdef, $5.expr, null, $7.slist,
$7.stmt.loc, $1.loc);
              }
            ;

BoundVariable   :   Type IDENTIFIER
                {
                    $$.vdef = new Tree.BoundVariable($2.ident, $1.type, $1.loc);
                }
                |   VAR IDENTIFIER
                {
                    $$.vdef = new Tree.BoundVariable($2.ident, new
Tree.TypeVar($1.loc), $1.loc);
                }
                ;
```

- 由于foreach语句有自己的作用域，并且在该作用域上可能定义新的变量，所以在BuildSym.java中实现对应LocalScope的初始化，并且在新定义的Scope上进行变量的定义。具体如下：

```
@Override
public void visitForeachStmt(Tree.ForeachStmt that) {
    that.associatedScope = new LocalScope(that.block);
    table.open(that.associatedScope);
    that.boundVariable.accept(this);
    that.arr.accept(this);
    if(that.judge != null){
        that.judge.accept(this);
    }
    that.block.accept(this);
    table.close();

}

@Override
public void visitBoundVariable(Tree.BoundVariable that) {
    that.type.accept(this);
    if (that.type.type.equal(BaseType.VOID)) {
        issueError(new BadVarTypeError(that.getLocation(), that.name));
        // for argList
        that.symbol = new Variable(".error", BaseType.ERROR,that
                .getLocation());
        return;
```

```
    }
    Variable v = new Variable(that.name, that.type.type,
            that.getLocation());
    Symbol sym = table.lookup(that.name, true);
    //变量重复定义的问题
    if (sym != null) {
        if (table.getCurrentScope().equals(sym.getScope())) {   //当前作用域内已经定义
了这个变量
            issueError(new DeclConflictError(v.getLocation(), v.getName(),
                    sym.getLocation()));
        } else if (sym.getScope().isFormalScope() &&
 table.getCurrentScope().isLocalScope()) {
            issueError(new DeclConflictError(v.getLocation(), v.getName(),
                    sym.getLocation()));
        } else {
            table.declare(v);
        }
    } else {
        table.declare(v);
    }
    that.symbol = v;
}
```

- 类型检查过程中：
  - 主要对语法中的变量类型是否符合要求进行检查，对var类型的变量进行符号表项的对应类型修改。
  - 实现对break语句的检查。在分析代码块之前将当前语法节点push入栈Breaks中，分析代码块结束之后再pop出来，防止错误的break语句报错情况。
- 在TransPass2.java中，主要实现：
  - 运行时对break语句的支持，
  - 为var类型的迭代变量赋予对应的Temp，
  - 生成对应迭代逻辑过程的tac码块。
  - 需要注意的点是：
    - 区分各个Temp赋值是否是在运行时进行（注意tr.genAssign与=的差别），生成对应的tac代码。
  - 具体代码如下：

```
//TransPass2.java
@Override
public void visitForeachStmt(ForeachStmt that) {
    Temp t = Temp.createTempI4();
    t.sym = that.boundVariable.symbol;
    that.boundVariable.symbol.setTemp(t);
    that.arr.accept(this);

    Label loop = Label.createLabel();   //循环入口
    Label exit = Label.createLabel();
    Temp esz = tr.genLoadImm4(OffsetCounter.WORD_SIZE); //一个数组单位的长度，4
    Temp len = tr.genLoadImm4(0);   //数组字节长度
    Temp base = tr.genLoadImm4(0);  //基址
    Temp j = tr.genLoadImm4(0); //基址
    Temp test = tr.genLoadImm4(0);
```

```
    Temp length = tr.genLoad(that.arr.val, -OffsetCounter.WORD_SIZE);   //获取数组长
度
    Temp idx = tr.genLoadImm4(0);

    tr.genMark(loop);

    tr.genAssign(len, tr.genMul(idx, esz));
    tr.genAssign(base, tr.genAdd(that.arr.val, len));
    tr.genAssign(t, tr.genLoad(base, 0));

    if(that.judge != null){
        that.judge.accept(this);
        tr.genAssign(test, that.judge.val);
    }else{
        tr.genAssign(test, tr.genLoadImm4(1));
    }
    tr.genBeqz(test, exit);
    loopExits.push(exit);
    that.block.accept(this);

    tr.genAssign(idx, tr.genAdd(idx, tr.genLoadImm4(1)));
    tr.genAssign(j, tr.genLes(idx, length));
    tr.genBnez(j, loop);
    tr.genMark(exit);

    loopExits.pop();

}
```