Decaf PA 2

王世因 2016011246

1. 类的浅复制scopy

- 修改文件TypeCheck.java：增加visit函数，考察scopy的参数的类型

这个的问题比较简单，只需要增加`visitScopy`函数的定义即可。通过便利各种可能的错误，得到判定条件，并写代码中。重点就是考察函数的两个参数是不是`class`和`class type`的。

```
@Override
public void visitScopy(Tree.Scopy scopy) {
    Symbol v = table.lookupBeforeLocation(scopy.indentifier,
scopy.getLocation());
    if (v == null) {
        issueError(new UndeclVarError(scopy.getLocation(),
scopy.indentifier));
    }
    else{
        if(!v.getType().isClassType()) {
            issueError(new BadScopyArgError(scopy.getLocation(), "dst",
v.getType().toString()));
            scopy.expr.accept(this);
            if (!scopy.expr.type.isClassType()) {
                issueError(new BadScopyArgError(scopy.expr.getLocation(),
"src", scopy.expr.type.toString()));
            }
            scopy.type = scopy.expr.type;
        }
        else{
            scopy.expr.accept(this);
            if (!scopy.expr.type.isClassType()) {
                issueError(new BadScopySrcError(scopy.getLocation(),
v.getType().toString(), scopy.expr.type.toString()));
            }
            scopy.type = scopy.expr.type;
        }
    }

}
```

2. sealed

- 修改文件symbol/Class.java：增加了类的`sealed`布尔值属性和`isSealed()`判断函数，体现

是否可以被继承

- 修改文件BuildSym.java：在定义新的类的时候，检查它的父类能不能被继承

这个需要在每次的定义新的`class`的时候访问它的父类，看看它是不是一个`sealed`的类，具体实现上直接在给定的`visitClassDef`的基础上添加如下的判断即可：

```
@Override
public void visitClassDef(Tree.ClassDef classDef) {
    table.open(classDef.symbol.getAssociatedScope());
    Class parent = table.lookupClass(classDef.parent);

    if(parent!=null && parent.isSealed()){
        issueError(new BadSealedInherError(classDef.getLocation()));
    }
    for (Tree f : classDef.fields) {
        f.accept(this);
    }
    table.close();
}
```

3. 串行条件卫士

- 修改文件TypeCheck.java：增加两个visit函数定义
- 修改文件BuildSym.java：增加两个visit函数定义

仿照`visitIf`，我利用了事先写好的`checkTestExpr(IFG.condition);`完成了检查：

```
@Override
public void visitGuard(Tree.Guard guard){
    for (Tree s : guard.block) {
        s.accept(this);
    }
}

@Override
public void visitIfG(Tree.IfG IFG){
    checkTestExpr(IFG.condition);
    if (IFG.trueBranch != null) {
        IFG.trueBranch.accept(this);
    }
}
```

4. 简单的类型推导

因为前面的三个问题只需要根据现有代码照猫画虎就可以，不涉及到建符号表和变量推倒，我做的比较快。因为我直接上手就开始写，没有事先摸清楚两遍遍历的细节，所以在这个问题上我卡了好久。后来我仔细研读了`BuildSym.java`，终于通过了所有的测试点。

- 修改文件type/BaseType.java：增加`VAR`类型

```
public static final BaseType VAR = new BaseType("unknown");
```

- 修改文件parser.y：更改原来在PA-1的写法，把类型推导的代码合并成一类处理

这个问题分为多步，首先是根据`var x = Expr.`定义新的变量，和`VarDef`的功能类似，只不过是需要看`Expr`的类型。因为我在`PA-1`中是把`VAR IDENTIFIER`放到`LValue`下的，而且LValue又衍生出很多的子类，在定义LValue的时候不能访问到`Expr`的类型信息，写起来比较复杂。于是我就重新写了`parser.y`的结构，把变量推倒的赋值单独写成一个语句，顺利解决了问题。

```
SimpleStmt      :   LValuel '=' Expr
                    {
                        $$.stmt = new Tree.Assign($1.lvalue, $3.expr,
$2.loc);
                    }
                |   VAR IDENTIFIER '=' Expr
                    {
                        $$.stmt = new Tree.VarAssign($4.expr, $2.loc,
$2.ident);
                    }
                |   Call
                    {
                        $$.stmt = new Tree.Exec($1.expr, $1.loc);
                    }
                |   /* empty */
                    {
                        $$ = new SemValue();
                    }
                ;
```

- 修改文件BuildSym.java：

```
public void visitVarAssign(Tree.VarAssign var) {
        var.expr.accept(this);
        var.type = var.expr.type;
        Variable v = new Variable(var.name, var.expr.type,
var.getLocation());
        Symbol sym = table.lookup(var.name, true);
        if (sym != null) {
            if (table.getCurrentScope().equals(sym.getScope())) {
                issueError(new DeclConflictError(v.getLocation(),
v.getName(),
                        sym.getLocation()));
            } else if ((sym.getScope().isFormalScope() &&
table.getCurrentScope().isLocalScope() &&
((LocalScope)table.getCurrentScope()).isCombinedtoFormal() )) {
                issueError(new DeclConflictError(v.getLocation(),
v.getName(),
                        sym.getLocation()));
```

```
            } else {
                table.declare(v);
            }
        } else {
            table.declare(v);
        }
        var.symbol = v;
    }
```

- 修改文件TypeCheck.java：

虽然我们在第一遍遍历的时候就处理过类型推导语句的类型，但是因为有下面这种情况的存在，我们在第二遍的时候还要再推导一遍。

```
var k = foo();
```

```
@Override
public void visitVarAssign(Tree.VarAssign var) {
    var.expr.accept(this);
    var.type = var.expr.type;
    Symbol v = table.lookup(var.name, true);
    if(v.isVariable()){
        v.type = var.expr.type;
    }
    var.symbol = (Variable) v;
}
```

这是我主要卡住的点，因为没有在第二遍的时候进行类型推导，`type`会设定为`null`，然后会在奇怪地地方报空指针错误，再加上报错的时候显示的类是此类的父类，我以为是类的继承上出了问题，我耗费了好多时间。

之后还有一些报错的小代码，比如不能让`unknown`出现在等式的右边这种，就很好写了，根据前三问的经验照猫画虎即可。

```
private Type checkBinaryOp(Tree.Expr left, Tree.Expr right, int op,
Location location) {
    ......
    if(left.type.equal(BaseType.VAR) || right.type.equal(BaseType.VAR)){
        compatible = false;
    }
    ......
}

@Override
public void visitAssign(Tree.Assign assign) {
    assign.left.accept(this);
    assign.expr.accept(this);
```

```
    if (!assign.left.type.equal(BaseType.ERROR) &&
(assign.left.type.isFuncType() ||
        assign.expr.type.equal(BaseType.VAR)||

!assign.expr.type.compatible(assign.left.type))) {
        issueError(new IncompatBinOpError(assign.getLocation(),
                   assign.left.type.toString(), "=",
assign.expr.type.toString()));
    }
}
```

5. 数组

5.1 数组初始化常量表达式%%

- 修改文件TypeCheck.java：

这个比较简单，只需要写一个visitor就好。

```
@Override
public void visitArrayInit(Tree.ArrayInit arr){
    arr.e1.accept(this);
    arr.e2.accept(this);
    if(arr.e1.type.equal(BaseType.VAR)){
        issueError(new BadArrElementError(arr.e1.getLocation()));
        arr.type = BaseType.ERROR;
        return;
    }
    else if(!arr.e2.type.equal(BaseType.INT)){
        issueError(new BadArrTimesError(arr.getLocation()));
        arr.type = BaseType.ERROR;
        return;
    }
    arr.elementType = arr.e1.type;
    arr.type = new ArrayType(arr.elementType);
}
```

5.2 数组下标动态访问表达式default

- 修改文件parser.java

我注意到实验原有代码中定义了一个`visitIndexed`，以为是可以应用于此句型的，于是就按照这个写了，但是结果发现这个的报错和数组访问的报错可能一样，但是对应的输出不一样，所以就重新写了一个`visitSlice`，结构和`visitIndexed`一样，只不过把报错的类型改了。对应在`parser.java`中也进行了修改。

```
Expr                | ArraySlice DEFAULT Expr
                    {
                        $$.expr = new Tree.ArrayDefault($1.lvalue, $3.expr,
$1.loc);
                    }
```

- 修改文件TypeCheck.java：增加了两个对应的`visitSlice`和`visitArrayDefault`。

```
@Override
public void visitSlice(Tree.Slice indexed) {
   indexed.lvKind = Tree.LValue.Kind.ARRAY_ELEMENT;
   indexed.array.accept(this);
   if (!indexed.array.type.isArrayType()) {
      issueError(new BadArrOperArgError(indexed.array.getLocation()));
      indexed.type = BaseType.ERROR;
      return;
   } else {
      indexed.type = ((ArrayType) indexed.array.type).getElementType();
   }
   indexed.index.accept(this);
   if (!indexed.index.type.equal(BaseType.INT)) {
      issueError(new BadArrIndexError(indexed.index.getLocation()));
   }
}

@Override
public void visitArrayDefault(Tree.ArrayDefault arr){
    arr.e.accept(this);
    arr.index.accept(this);
    if(arr.index.type.equal(BaseType.ERROR)){
        arr.type = BaseType.ERROR;
        return;
    }
    if(arr.e.type.equal(arr.index.type)) {
        arr.type = arr.e.type;
        return;
    }
    else{
        issueError(new BadDefError(arr.getLocation(),
arr.index.type.toString(), arr.e.type.toString()));
    }

    if(arr.index.type.equal(BaseType.INT) ||
arr.index.type.equal(BaseType.STRING) ||
       arr.index.type.equal(BaseType.BOOL)||arr.index.type.isArrayType()){
        arr.type = arr.index.type;
        return;
    }
```

```
    if(arr.e.type.equal(BaseType.INT) || arr.e.type.equal(BaseType.STRING)
||
       arr.e.type.equal(BaseType.BOOL)||arr.e.type.isArrayType()){
        arr.type = arr.e.type;
        return;
    }
    arr.type = BaseType.ERROR;
}
```

5.3 数组迭代语句

- 修改文件 Tree.java

我利用事先写好的`VarDef`和`IdentVar`，定义循环变量，具体实现如下。因为要兼顾PA1的功能，写起来有点冗余。另外，根据实验文档的提示，我新建了一个虚拟的`Block`用于定义局部变量的作用范围，这个`Block`中只有`ArrayFor`一条语句，满足要求。

```
public static class ArrayFor extends Tree{
    public LValue iter;
    public Tree.Block block;
    public Expr array, j;
    public LocalScope associatedScope;
    public Tree.Block virtualScope;
    public VarDef vardef;
    public IdentVar identvar;

    public ArrayFor(LValue iter, Expr array, Tree e2, Expr j, Location loc,
VarDef vdef, String varname, Location varloc) {
        super(ARRAYFOR, loc);
        this.iter = iter;
        this.block = (Tree.Block) e2;
        this.array = array;
        this.j = j;

        List <Tree> list = Arrays.asList(this);
        this.virtualScope = new Expr.Block(list, loc);

        assert (vdef==null && varname!=null) || (vdef!=null &&
varname==null);
        if(varname!=null){
            this.identvar = new IdentVar(varname, varloc);
        }
        if(vdef!= null){
            this.vardef = vdef;
        }

    }
    @Override
    public void accept(Visitor v) {v.visitArrayFor(this); }
```

```
    @Override
    public void printTo(IndentPrintWriter pw) {
        pw.println("foreach");
        pw.incIndent();
        iter.printTo(pw);
        array.printTo(pw);
        if(j!=null){
            j.printTo(pw);
        }
        else{
            pw.println("boolconst true");
        }
        if(block!=null){
            block.printTo(pw);
        }
        pw.decIndent();
    }
}
```

- 修改文件BuildSym.java：

和之前一样，在局部中，定义循环变量。

```
@Override
public void visitArrayFor(Tree.ArrayFor arr){
    arr.associatedScope = new LocalScope(arr.virtualScope);
    table.open(arr.associatedScope);
    arr.array.accept(this);

    if(arr.vardef!=null){
        arr.vardef.accept(this);
    }

    if(arr.identvar!=null){
        arr.array.accept(this);
        Variable v = new Variable(arr.identvar.name, BaseType.VAR,
arr.identvar.getLocation());
        Symbol sym = table.lookup(arr.identvar.name, true);
        if (sym != null) {
            if (table.getCurrentScope().equals(sym.getScope())) {
                issueError(new DeclConflictError(v.getLocation(),
v.getName(),
                                                 sym.getLocation()));
            } else if ((sym.getScope().isFormalScope() &&
table.getCurrentScope().isLocalScope() &&
((LocalScope)table.getCurrentScope()).isCombinedtoFormal() )) {
                issueError(new DeclConflictError(v.getLocation(),
v.getName(),
```

```
                                                    sym.getLocation()));
            } else {
                table.declare(v);
            }
        } else {
            table.declare(v);
        }
        arr.identvar.symbol = v;
        arr.identvar.type = BaseType.VAR;
    }

    for (Tree s : arr.block.block) {
        s.accept(this);
    }

    table.close();
}
```

- 修改文件TypeCheck.java：完成循环变量的类型推导，并进行一些简单的查错。

```
@Override
public void visitArrayFor(Tree.ArrayFor arrfor){
    arrfor.array.accept(this);
    Tree.Ident array = (Tree.Ident) arrfor.array;
    table.open(arrfor.associatedScope);
    Variable arr = (Variable) table.lookup(array.name, true);
    if(arrfor.j!=null){
        checkTestExpr(arrfor.j);
        if(arrfor.j.type.equal(BaseType.ERROR)){
            return;
        }
    }
    if(arrfor.array.type.equal(BaseType.ERROR)){
        return;
    }
    else if(arr==null){
        issueError(new BadArrOperArgError(arrfor.array.getLocation()));
        arrfor.array.type = BaseType.ERROR;
        return;
    }
    else if(!arr.type.isArrayType()){
        issueError(new BadArrOperArgError(arrfor.array.getLocation()));
        arrfor.array.type = BaseType.ERROR;
        return;
    }
    else{
        arrfor.array.type = arr.getType();
        if(arrfor.identvar!=null &&
arrfor.identvar.type.equal(BaseType.VAR)) {
```

```
            Variable ident = (Variable) table.lookup(arrfor.identvar.name,
true);
            ident.type = ((ArrayType) arrfor.array.type).getElementType();
        }
    }
    for (Tree s : arrfor.block.block) {
        s.accept(this);
    }
    table.close();

}
```

实验总结和体会

照猫画虎固然可行，但是提前摸清楚算法的原理，可以让我们少走许多弯路，对于实验也有一个高屋建瓴的把控。visitor模式太巧妙了，我体会到了数据结构设计的魅力，这个思想我会好好记住，没准日后会再用上。

我注意到在Intellij IDE中，编译的时候会进行冲突检查，在继承关系复杂的时候，因为不知道具体的输入是子类还是父类，使用只有子类中有的属性的时候会报错。所以有的时候需要强制类型转化一下。