PA2 实验报告

计73 王焱 2017050024

一、实验综述

本阶段目标是对PA1中生成的抽象语法树进行语义分析。

二、具体实现

针对三个新添加的语法特性 `abstract`, `var`, `lambda` ,在PA1的词法、语法分析的基础上，进行语义分析，具体就是建立符号表并进行类型检查和推导，大致是先后在Namer和Typer中先后进行。

1.抽象类

由于在词法分析时，由abstract修饰的成员函数的函数体赋成了null，所以在Namer和Typer先后两次遍历AST时，对一个函数体的访问需要加上非空的判断，例如：

```
if(method.body != null)
    method.body.accept(this, ctx);
```

将 `ABSTRACT` 正确打印出来，抽象方法的打印由modifier管理，抽象类需要加特殊判断，在ClassSymbol.str()中实现：

```
 @Override
    protected String str() {
        String ABSTRACT = isAbstract ? "ABSTRACT " : "";
        return  ABSTRACT + "class " + name + parentSymbol.map(classSymbol -> " :
" + classSymbol.name).orElse("");
    }
```

最后一步处理相关的报错：

1. Main类不能为抽象类；

   在Namer的visitTopLevel中寻找主类时，添加是否是abstract的判断。

2. 不允许抽象方法重写基类中非抽象方法；

   在Namer的visitMethodDef中添加对于重载方法的判断条件。

3. 需要却没有声明抽象类；

   在ClassSymbol中记录并维护一个类的未重载的抽象方法列表 `public List<String> notOverride;`

4. 抽象类不能使用`new`进行实例化。

   在Typer的visitNewClass中添加是否是抽象类的判断。

2.局部类型推断

在Namer的visitLocalVarDef中 添加判断，如果过typeLit为空，则建一个type为空的VarSymbol。在Typer的visitLocalVarDef中当左边的类型为空时，即需要进行局部变量类型推导，将右边初始化表达式的类型赋给当前局部变量。

```
        //var类型推导
        if(lt == null){
            if(rt.isVoidType()){
                issue(new BadVarTypeError(stmt.id.pos,stmt.id.name));
                stmt.symbol.type=BuiltInType.ERROR;
                return;
            }
            else
                stmt.symbol.type = rt;
        }
```

3.First-class Functions

为了实现这个新特性，需要实现以下几个方面：

- 添加函数类型

  框架中已给出FuncType类型，只需要在TypeLitVisited中对其进行解析，实现visitTLambda函数。

```
@Override
    default void visitTLambda(Tree.TLambda tLambda, ScopeStack ctx){
        tLambda.returnType.accept(this, ctx);
        if(tLambda.returnType.type.eq(BuiltInType.ERROR))
            tLambda.type = BuiltInType.ERROR;
        var hasError = false;
        var argTypes =new ArrayList<Type>();//type
        for(var param: tLambda.params){
            param.accept(this, ctx);
            if(param.type.eq(BuiltInType.ERROR)) {
                tLambda.type = BuiltInType.ERROR;
                hasError = true;
            }
            else if(param.type.eq(BuiltInType.VOID)){
                issue(new VoidArgsError(param.pos));
                tLambda.type = BuiltInType.ERROR;
                hasError = true;
            }else{
                argTypes.add(param.type);
            }
        }
        if(!hasError){
            tLambda.type = new FunType(tLambda.returnType.type, argTypes);
        }

    }
```

- Lambda 表达式返回类型推导

  对于有Expr的lambda表达式，lambda表达式的类型就是Expr的类型，对于有Block的lambda表达式，则需要对Block中返回的类型进行推导。

  首先在Namer中添加visitLambda对lambda表达式进行解析，在Typer中添加visitLambda对其进行类型推导与检查。实现类型推导主要参考实验指导书上的算法，实现了getReturnType，upperbound， lowerbound。

对于如何收集return类型的问题，学习了同学的模拟栈结构，每进入一个嵌套作用域加入一个新的list，用来收集严格当前作用域内的return值。

- 函数变量与函数调用

  由于做出这样的修改之后，对于一个函数名的出现，不再仅仅只可能是加（）的函数调用，还可能是对于函数变量的引用，涉及到vaesel, 因为varsel相当于对变量的引用。原有的varsel中分无receiver和有receiver两种情况来处理，因此在这两种情况下都需要修改对于成员方法的支持，并且还要进行权限检查，即静态方法不能调用非静态方法，无receiver的情况下：

```
if(symbol.get().isMethodSymbol()){
    var method = (MethodSymbol)symbol.get();
    expr.symbol =method;
    expr.type = method.type;
    if(method.isMemberMethod()){
        expr.isMemberMethodName = true;
        if(ctx.currentMethod().isStatic()&&!method.isStatic()){
            issue(new
RefNonStaticError(expr.pos,ctx.currentMethod().name,expr.name));
        }else{
            expr.setThis();
        }
    }
    return;
}
```

另外对于call中，将原来确定的各个域合并成了一个Expr，调用它的accept 得到类型，如果是FunType，就可以进行调用。

```
    @Override
    public void visitCall(Tree.Call expr, ScopeStack ctx) {
        expr.func.accept(this, ctx);//访问expr
        //Log.fine("func %s",expr.func.toString());
        if (expr.func.type.hasError()) {
            expr.type = BuiltInType.ERROR;
            return;
        }
        if (!expr.func.type.isFuncType()) {
            issue(new NotCallableError(expr.pos,
expr.func.type.toString()));
            expr.type = BuiltInType.ERROR;
            return;
        }
        if (expr.func instanceof Tree.VarSel) {
            var v1 = (Tree.VarSel) expr.func;
            if (v1.isArrayLength) {
                expr.isArrayLength = true;
                expr.type = BuiltInType.INT;
                if (!expr.args.isEmpty())
                    issue(new BadLengthArgError(expr.pos,
expr.args.size()));
                return;
            }
        }
        typeCall(expr, ctx);
    }
```

- Lambda表达式的作用域

  这部分参考实验指导书，为lambda新添加lambdaScope，一个函数定义的时候是一个formalScope中套一个localScope， 一个lambda表达式定义的时候是一个lambdaScope套一个localScope，localScope和lambdaScope可以相互嵌套。

  框架中原有一个`isFormalOrLocalScope()`函数，用来检查定义符号冲突，现在由于先加入的这种嵌套关系，将其改为`isFoemalOrLocalOrLambdaScope()`。

  对于不能对捕获的外层的非类作用域中的符号直接赋值这一限制，需在visitAssign中进行修改，首先找到节点所在的FormalScope或LambdaScope，然后通过判断它的上一层如果不是一个ClassScope，那么他就是一个lambdaScope。

  对于引用正在定义的变量这一问题，采用全局记录正在定义的变量的方法，在访问initval之前加入正在定义的变量列表，在访问完成之后从列表中删除，而在Varsel检查中需要添加判断条件当前引用的符号不在正在定义的变量列表中。

```
varListStack.add(stmt.name);
var initVal = stmt.initVal.get();
localVarDefPos = Optional.ofNullable(stmt.id.pos);
initVal.accept(this, ctx);
localVarDefPos = Optional.empty();
varListStack.remove(varListStack.size()-1);
```

三、实验思考

Q1. 实验框架中是如何实现根据符号名在作用域中查找该符号的？在符号定义和符号引用时的查找有何不同？

答：在TopLevel 中实例化一个作用域栈，将其作为一个参数进行传递，从当前栈顶向下逐层查找即可找到对应符号 。定义时需要根据当前作用域类型的不同以不同的方式检查冲突 ，引用时只需要检查符号的存在性 。

Q2. 对 AST 的两趟遍历分别做了什么事？分别确定了哪些节点的类型？

答：第一次遍历，构造符号表，确定了ClassDef， MethodDef， LocalVarDef（除了var）的类型；第二次遍历，对访问到的symbol进行类型检查和推导，如 var和 lambda。

Q3. 在遍历 AST 时，是如何实现对不同类型的 AST 节点分发相应的处理函数的？请简要分析。

答：抽象类Visitor负责声明不同结点类型的accept方法，然后在具体的节点类中重载accept抽象方法，调用当前结点类的访问函数。由于java具有动多态特性，在访问不同类型节点时，调用accept方法，会调用到重写的accept方法从而调用到不同类的visit方法。

四、实验小结

本次实验，对lambda的语义分析这部分感觉实现起来比较困难，主要是对于某些非固定算法流程的东西，由于自己对框架的理解还不是很到位，所以添加起来很困难，比如函数调用这块，花了很长时间去理解。完成实验之后，也的确使我对一些类有了更新的理解。另外本次实验持续的时间比较长断断续续有一周的时间，从开始的摸索，到逐渐明白，这期间造机的队友们给了我很大帮助，让我能继续写下

去，在此表示感谢。