PA2 实验报告

计科70 徐明宽 2017011310

工作内容

合并前一阶段的工作

我使用`git am`打了patch，在这过程中发现Namer.java中有我在PA1-A中为了能编译通过而修改的与patch不同的部分，于是我把我自己的修改撤销掉以后成功打上了patch。

抽象类

方便起见，我在ClassSymbol.java中、在MethodSymbol.java中、在Tree.java的`Classdef`类和`MethodDef`类中加入了函数`isAbstract()`。

由于主类不能是抽象的，我在Namer.java中的`visitTopLevel`中加入了相应判断。

为了输出类类型的`modifiers`（即`ABSTRACT`），我在ClassSymbol.java中加入了`modifiers`，并修改了Namer.java中的`createClassSymbol`。

为了正确输出抽象方法的`FormalScope`，我修改了PrettyScope.java与FormalScope.java，将后者中的`nested`变量改为`Optional<LocalScope>`类型，并在Namer.java中的`visitMethodDef`函数内对于抽象方法不执行`ctx.open(formal);`等代码段，也即使得`nested`变量为`Optional.empty()`。

在Typer.java的`visitMethodDef`函数中，我加入判断使得只有非抽象方法才执行该代码段。

对于错误1，我实现了NotAbstractClassError.java，在Tree.java中为`ClassDef`类新增了成员变量`public List<MethodDef> unOverriddenMethods;`，参考实验指导书在Namer.java中的`visitClassDef`函数内实现了相应逻辑。在实现时，我只考虑了没有报`BadOverrideError`或`DeclConflictError`错误的方法，也即加入了判断`((Tree.MethodDef) field).symbol != null`。

对于错误2，我实现了InstantAbstractClassError.java，在Typer.java中的`visitNewClass`函数内实现了相应逻辑。

局部类型推导

我在Namer.java的`visitLocalVarDef`中对于需要类型推导的情况暂不去检查类型，在Typer.java中已经推导出`initVal`的类型以后将其类型赋给需要类型推导的变量，并检查若结果为`void`类型则报错。

First-class Functions

函数调用

为了调对局部类型推导的测例var-error-1.decaf，我先实现了这一部分。

由于需要将大段逻辑从`Call`中移到`VarSel`中，我在Tree.java中删去了`Call`的成员变量`methodName`与成员方法`setThis()`，在`VarSel`中加入了成员变量`public boolean isArrayLength = false;` `public boolean isStaticMethod = false;` `public Optional<String> receiverClassName;`。这样，我便可以在Typer.java中让`visitVarSel`中的`expr`返回（存储）足够多的信息，以此在`visitCall`函数中调用`typeCall`。对于`expr`返回的不是函数类型的情况，我实现了NotCallableTypeError.java并直接报错。

在`visitCall`中，我将`NotClassFieldError`、`RefNonStaticError`与`FieldNotFoundError`的报错位置由以前的`call.pos`改为了`call.methodExpr.pos`。

函数类型

我实现了BadFunArgTypeError.java，并在TypeLitVisited.java中实现了`default void visitTLambda(Tree.TLambda typeLambda, ScopeStack ctx)`，在返回类型与所有参数类型`accept`之后再去（当有参数为`void`类型时）报错并将该函数类型的`type`设为`ERROR`（而不是有错就直接设为`ERROR`并返回）。

在局部类型推导部分，我去掉了Typer.java的`visitLocalVarDef`中要求不能为函数类型的检查。

Lambda表达式作用域

我参考LocalScope.java实现了LambdaScope.java，参考MethodSymbol.java实现了LambdaSymbol.java。在LocalScope.java中，我在构造函数中加入了`parent`为`LambdaScope`的情况判断，并将`public List<LocalScope> nestedLocalScopes()`改为了`public List<Scope> nestedLocalOrLambdaScopes()`。

在Scope.java中，我将`isFormalOrLocalScope`改为了`isFormalOrLocalOrLambdaScope`，并修改了所有用到该函数的地方。

在VarSymbol.java中，我修改了`isParam()`的判断。

在Namer.java中，我参考`visitMethodDef`实现了`visitLambda`。为了使得这个函数会被访问到，我修改了函数`visitLocalVarDef, visitIf, visitWhile, visitFor`并简单实现了以下函数：`visitExprEval, visitAssign, visitBinary, visitCall, visitClassCast, visitClassTest, visitIndexSel, visitNewArray, visitPrint, visitReturn, visitUnary, visitVarSel`。

在Typer.java的`visitVarSel`中，我加入了`LambdaSymbol`的相关判断。

在Symbol.java中，我不得已删去了`type`的`final`修饰符；在VarSymbol.java中，我对于局部类型推导实现了新的构造函数；在LambdaSymbol.java中与VarSymbol.java中，我实现了函数`public void setType(Type type)`。以此，我能在Namer.java中定义lambda表达式与需要局部类型推导的变量的symbol，并修改ScopeStack.java中的`lookupBefore`函数，满足实验要求。

我实现了AssignCaptureError.java，在LocalScope.java中加入了成员变量`public final Scope parent;`，并在Typer.java的`visitAssign`中加入了lambda表达式中的对捕获的外层的非类作用域中的符号直接赋值的判断。

在Namer.java的`visitLocalVarDef`中，我将`def.initVal.ifPresent(objects -> objects.accept(this, ctx));`移到了所有`ctx.declare(symbol);`语句的后面。

Lambda 表达式返回类型

我在Tree.java中的`Stmt`里新增了变量`public Type returnType`，在BuiltInType.java中新增了`BuiltInType INCMP`（Incompatible），并规定任何类型都是`INCMP`的子类型。这样，推导出该类型即表明“类型不兼容”。之后我在Type.java中实现了函数`public Type upperBound`与`public Type lowerBound`，用于求出两个`Type`的上下界。为了处理`Stmt`里`returnType`可能为空的问题，我又在`Stmt`里实现了函数`public void updateReturnType`。以此，我修改了Typer.java中的函数`visitBlock, visitFor, visitIf, visitWhile, visitReturn`以更新return type。

我实现了IncompatRetTypeError.java，当block lambda的返回类型为`INCMP`时即报错。当block lambda的`returns`为`false`时，报`MissingReturnError`。

函数变量

我去掉了Typer.java的`visitAssign`中要求不能为函数变量的检查。为了对于对一个类的成员方法进行赋值的情况报错，我实现了AssignMethodError.java，并在Tree.java的`VarSel`中加入了成员变量`public boolean isMethod = false;`，以区分函数变量与成员方法。

回到函数调用

我在Typer.java中的`visitIndexSel`中加入了对于`expr.array.type`为`ERROR`时不再报错的检查。

我实现了BadLambdaArgCountError.java，简化了Typer.java中的`visitCall`使其不再调用`typeCall`。

回答问题

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？

框架中根据符号名在作用域中查找该符号的逻辑实现在了ScopeStack.java中，具体为`findConflict`和`lookupBefore`这两个函数，它们都调用了（前者还调用了（可以改成私有的）公有函数`lookup`，而`lookup`也调用了`findwhile`）一个私有函数`findwhile`，该函数直接**从内向外**遍历当前作用域栈，查找是否有满足要求的符号（要求即上述两个函数传入`findwhile`的参数，包括对作用域的限制条件与对符号的限制条件；而在一个作用域中查找即直接在`TreeMap`（平衡树）中查找该字符串），如果有多个满足要求的话返回第一个（即最内层的）。

在符号定义时，我们采用的是`findConflict`，即当当前作用域为非类作用域时对作用域的限制条件为非类作用域或全局作用域（否则无限制），对符号无限制。由于在Namer.java中定义一个符号时还没有访问到后面的符号，我们无需考虑一个符号与后面的符号重名的问题——毕竟报错是后面再次发现同样名称的符号时干的事情。

在符号引用时，我们采用的是`lookupBefore`，即对作用域无限制，但对局部作用域中的符号限制在当前定义变量的位置之前（如果当前正在定义变量的话），以避免形如`int x = x + 1;`的情况。在Typer.java中，我们已经在上一阶段定义了所有的符号（因此可以查找到后面的符号，这也是与`findConflict`的一个不同之处），因此必须对符号加以位置的限制（在实验中的lambda表达式作用域部分，我们还要加以“不是当前正在定义的符号”这一限制）。

考虑下例：

```
var a = fun(int u) {
    var b = fun(int v) {
        int c = c;
    };
    int c;
};
```

在符号定义时，在定义第一个`int c`时第二个还没有定义，在定义第二个`int c`时第一个又在内层，因此`findConflict`查找不到；在符号引用时，`int c = c;`要引用`c`，而此时第一个`int c`正在定义，第二个`int c`位置又在后面，因此会报错`undeclared variable 'c'`。

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？

第一趟遍历，即Namer.java中，我们检查了冲突的与其他一些不合法的定义（如定义一个类型为`void`的变量等），检查了类的继承关系形成森林，建立了所有作用域（`Scope`）与相应的符号（`Symbol`）表（虽然其中部分类型未知），定位了主类`Main`与主函数`static void main()`。概括地说，我们明确了程序声明了哪些标识符。

由于`Namer`继承了`TypeLitVisited`，我们在第一趟遍历中即确定了`TInt, TBool, TString, TVoid, TClass, TArray`以及实验中实现的`TLambda`节点的类型。在Namer.java中，我们也确定了`ClassDef, MethodDef, LocalVarDef`（如果不需要局部类型推导的话）节点的类型。

第二趟遍历，即Typer.java中，我们进行了自顶向下的类型检查，确定了（除`TopLevel`外）其余所有节点的类型，并在发现错误时根据具体情况报相应的错。换句话说，我们明确了每一处使用的标识符对应于哪一处的声明，以及各语句和表达式是否类型正确。

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。

参考实验说明中的"访问者模式"小节，我们让每个节点都支持一个负责转发的方法`accept`，如：

```
@Override
public <C> void accept(Visitor<C> v, C ctx) {
    v.visitTopLevel(this, ctx);
}
```

再将遍历AST的`Namer`和`Typer`等都实现为一个访问者的实例，这样，对于任意的节点（`TreeNode`），我们只要调用它的`accept`方法，那么按照 OOP 的动态分派机制，根据这个`TreeNode`的具体类型，相应的处理函数将被调用。具体来说，执行`node.accept(v)`时：

- 如果`node instanceof TopLevel`，那么相当于调用的是`TopLevel`类的`accept`方法，该方法会调用`v.visitTopLevel(node, ctx);`
- 如果`node instanceof ClassDef`，那么相当于调用的是`ClassDef`类的`accept`方法，该方法会调用`v.visitClassDef(node, ctx);`
- ……

致谢

感谢罗承扬、杨家齐、以及编译原理群中的同学们和助教们与我的讨论。

声明

我于2019年11月17日将代码和报告（包括实验报告中问题的回答） 交给了傅舟涛同学参考。

我于2019年11月17日将代码交给了蒋佳轩同学、胡亦行同学参考。

我于2019年11月18日将报告交给了胡亦行同学参考。

我于2019年11月25日将代码和报告交给了卢睿同学参考。

我于2019年11月25日将代码交给了尹龙晖同学参考。

我于2019年11月29日将代码交给了潘慰慈同学参考。