编译原理PA2

实验内容

抽象类和抽象方法

需要作出的改动：

- 寻找作为程序入口的类`Main`时，需要限定这个类不能是抽象的；
- 扩展`ClassSymbol`、`Tree.ClassDef`、`MethodSymbol`，增加是否抽象的判断接口；
- 每个`ClassSymbol`维护一个抽象函数列表。这个列表在`Namer`的`visitClassDef`中创建，在遍历所有域的时候更新，遍历完成后判断是否有违本类的抽象属性；
- 抽象函数的函数体可以为空，因此在PA1-A时不敢修改的`Tree.ClassDef`的`body`属性可以改为`Optional<Tree.Block>`类型，访问到它的地方酌情修改；
- 函数重载的修改：增加判断，不能以抽象函数覆盖非抽象函数；发生函数覆盖时，如果被覆盖的函数是抽象的，需要将其移出抽象函数列表；
- 修改`Typer`的`visitNewClass`，不允许实例化抽象类；
- 框架没有处理类内重复声明的情况，将其视为重载而不报错，需要保证当前类的`Scope`中不存在这个被所谓将被覆盖的对象。

类型推断

需要修改`Typer`的`visitLocalVarDef`。首先，因为`Namer`不能推断出被赋值的符号的类型，因此在这里进行判断，如果为空，则将右侧表达式的推断类型赋值给它。这里需要特殊判断，不能赋以`void`类型。

函数参数类型检查

这里需要做的事情，是拒绝在参数列表中指定`void`类型作为参数。成员函数的参数在`Namer`中得到了检查，但是新出现的函数类型`TLambda`需要得到检查。这一部分代码在`TypeLitVisited.java`中，需要从头开始写对函数类型的检查。根据我的实现，只要有一个参数为`void`即立即停止对后面参数的解析与检查，该函数类型推断失败。

Lambda表达式作用域

我认为这是本次实验中最复杂、最难处理的特性。

首先按照助教的提示，考虑构造`LambdaScope`。这个类几乎可以仿造`FormalScope`，然而又被`LocalScope`所包含。`LambdaScope`需要对应地构造`LambdaSymbol`类，用于代表出现的任意一个Lambda表达式。它和`MethodSymbol`更加相似，因为都可以作为函数类型的右值。

`ScopeStack`需要对`open(LambdaScope)`和`close()`进行特殊处理，把对应的`LambdaSymbol`加入栈中，这样就可以访问到当前所处的`LambdaScope`。

其次，考虑如何构造`LambdaScope`和`LambdaSymbol`。符号表在`Namer`构造，因此当然要更改`Namer`。首先，重载`visitLambda`，做这样几件事：首先对所有的子节点进行遍历，然后构造函数类型，再构造`LambdaScope`。观察输出，`Lambda`表达式的`Scope`结构与函数非常类似，也是一个`FormalScope`套一个`LocalScope`，因此对于带语句块的Lambda表达式，内部的`LocalScope`可以通过对`Tree.Block`的遍历来构造，对于带表达式的Lambda函数则需要手动创建`LocalScope`。

需要注意的是，Lambda表达式理论上几乎可以出现在任何`Expr`中，因此我扩充代码，对于AST任何节点，只要包含有`Expr`属性，都要进行遍历。比如我可以写出如下代码：

```
for (var i = 0; i < MAXV; i = (fun(int x)=>x*x)(next(i)))
{
    // ...
}
```

如果不遍历到`For`的`update`节点，就会遗漏这个表达式。

最后，考虑变量的访问规则与作用域。体现到`Typer`的`visitVarSel`中，代码非常混乱，因此我制作了以下表格来帮助我理清楚思路：

> 可访问性：

expr\receiver	<empty>	this	a/expr	Classname
局部变量	Yes	X	X	X
Lambda表达式外局部变量	除正在被赋值的符号	X	X	X
类名	Yes	X	X	X
非静态函数	非静态函数	非静态函数	非静态函数	No
静态函数	Yes	Yes	Yes	Yes
成员变量	非静态函数	非静态函数	限子类	No
length		X	限数组	X

> 可修改性

左值符号\左值AST节点	VarSel	IndexSel
局部变量	Yes	Yes
成员变量	Yes	Yes
Lambda表达式外局部变量	No	No
类成员函数/静态函数	No	\
array.length	No	No

我们对`visitVarSel`作出如下行为规定：`Tree.VarSel`类型的`expr`在执行完毕后，其`type`必得到更新；除了找不到符号（报错）或访问`array.length`的情况，`symbol`必得到更新；如下错误要得到处理：未声明的变量、非静态函数引用成员、不存在的类字段、没有访问权限、通过类名调用成员字段。

可修改性体现在`visitAssign`中，要求被赋值的符号属于变量，如果左值属于`Tree.VarSel`，那么首先要`visitVarSel`返回了正确的`symbol`，然后再检查是否是函数、是否是正在被赋值的符号。当然，左值也有可能是`IndexSel`，但是被访问的数组既不可能是函数，也不能是定义一个Lambda表达式（至多是给一个函数数组的某个元素赋值），因此这种情况没有必要判断。

最后，我们还有两个难题需要处理。如何处理正在被赋以Lambda表达式的符号？根据规范，它既不能被引用，也不能被定义。不能被定义的限制容易实现，只需要先定义符号再赋予初值即可，注意根据测例当符号冲突时仍要检测其他错误。不能被引用的限制则比较麻烦，因为在处理初值对应的AST节点时，它并没有和符号相关联，再加上类型是可推导的，不能从符号判定它是否是Lambda表达式。最后根据实验说明提示，我对`ScopeStack`进行扩展，增加一个栈来表示正在被赋值的变量，这样普通的变量也可以纳入统一的框架下。访问该符号统一在`visitVarSel`中处理，如果该符号出现在栈中，那么报错'undeclared variable'。我的代码比较复杂，原因是我起初的理解是这一限制对于赋值语句的两边也要成立。

如何判断被赋值的符号是否在其他的`LambdaScope`中？一种情况是当前的`LambdaScope`内包含这个符号，这种包含不是被其他`LambdaScope`所"间接"包含，也不是直接调用`Scope::find`就能找到，而是可能被若干层`LocalScope`所包起来；另一种情况是这个符号在当前的`ClassScope`中有定义。后一种情况是最简单的，只需要`find`就可以确定了；前一种情况，我通过查找到符号所属的`LocalScope`不断取它的`parent`，直到遇到一个`FormalScope`或`LambdaScope`，然后判断是否是当前所在的`Scope`。这一部分判断在`visitAssign`中处理。

Lambda表达式类型

因为`var`语句的存在，我们必须要能够从语句中推断出返回类型是什么。我需要求出所有返回值类型的共同下界。

首先我抛出一个显然的结论，那就是我们可以对返回值类型以任意顺序两两结合。形式化地：

> 设$T_1,\ldots,T_k$是函数各个执行分支的返回值，$LB(\{T_i\})$是$\{T_i\}$的类型下界，那么
> (1) $LB(T_1,LB(T_2,T_3))=LB(LB(T_1,T_2),T_3)$
> (2) $LB(T_1,\ldots,T_k)=LB(LB(T_1,\ldots,T_{k-1}),T_k)$

这样，我只需要编写函数`typeLowerBound(Type t1, Typr t2)`求出两个类型之间的下界即可。

对于每一类`Stmt`，我们使用如下两个变量：

- `returns`：该语句是否所有的分支末尾都是显式的`return`语句；
- `returnType`：该语句隐式返回了什么类型。

我们可以自底向上，按照如下规则依次进行处理：

语句类型	**`returns`**	**`returnType`**
Return	`true`	返回值的类型（返回空则为`void`）
If	`b1.returns`或`b1.returns && b2.returns`	单分支：`b1.returnType` 多分支：`typeLowerBound(b1.returnType, b2.returnType)`
For	`false`	`typeLowerBound(stmt[*].returnType)`
While	`false`	`loop.body.returnType`
Block	`true` if any`stmt[i].returns==true`	`typeLowerBound(stmt[*].returnType)`，直到第一处返回*

注：* 为了遵循原框架，实际代码中这里不作修改

其实这一套框架可以支持更加宽松的返回值检查方式，完全可以采用第一次遇到的return作为类型推断的依据。

最后，回到`visitLambda`，如果`block.returns==false`那么报错`missing return statement`；如果`typeLowerBound`结果是没有共同下界（新建`BuiltinType.INVALID`来表示）那么报错`incompatible return types`。

接下来需要实现`typeLowerBound`，以及配套的`typeUpperBound`。具体的原理实验说明文档已经很详细，至于`typeUpperBound`，唯一值得一提的改变是在处理`ClassType`的过程中，只需检查二者是否是派生关系，如$t_1 <: t_2$。

函数变量

这一部分对应框架中`Typer::visitVarSel`的扩充，需要额外判断当前取得的符号是否是`MethodSymbol`。这里仍然可以参照`VarSymbol`部分的代码来进行处理，注意静态函数可以通过对象、类名来访问。此外，数组的`length()`方法需要特别处理，此时`receiver`是`ArrayType`，给待解析的`VarSel`型变量`expr`的`type`属性赋以函数类型`()=>int`。此外，借鉴原有框架的`visitCall`，给`Tree.VarSel`增加了``isArrayLength`属性，这样大大方便`visitCall`处理。

函数调用

修改后框架的`visitCall`非常简单，只需要按顺序解析待调用的表达式、参数对应的AST节点，之后检查待调用表达式类型、参数类型、参数数目即可，同时不要忘记数组类型的`length()`方法的解析。

问题思考

1. > 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？

   Java框架定义了多种方法，首先每个`Scope`对象都有`find`、`containsKey`、`get`方法，还为`ClassScope`定制了`lookup`方法，可以在本类和继承自的类中进行查询。其次`ScopeStack`通过`findWhile`方法定制了多种查询方式，比如在全部作用域中查询、在符号定义前的作用域中查询、在局部变量中查询，等等。

   符号定义一般期望“找不到”，但是如果找到了，在特殊情况下（如重写覆盖、屏蔽类成员）仍然可以覆盖，寻找的作用域是全局、类或函数（`findConflict`）。符号引用一般期望“能找到”，寻找的作用域是带有位置限定的局部变量（`lookupBefore`），或者类的成员（`ClassScope::lookup`）。

2. > 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？

   `Namer`构建符号表和各种符号，同时对各种类型进行访问和检查，比如函数类型。`Typer`检查对符号的引用是否正确，保证运算和操作的数据类型正确。第一遍检查“定义得对不对”，第二遍检查“引用、类型对不对”。

   第一遍中确定了节点`TopLevel`、`ClassDef`、`VarDef`、`MethodDef`、`Lambda`、`TypeLit`的派生类。第二遍确定了所有`Expr`派生节点的类型。`LocalVarDef`是第一、第二遍共同确定的。这里的“确定”，我讲的是给AST节点确定`symbol`和`type`的属性值。

3. > 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。

   首先观察到每个`TreeNode`节点都有一个`accept(Visitor<C>, C)`方法，不同的子类以不同的实现重载了该方法，调用`Visitor<>`的不同方法。`Namer`、`Typer`以及后面的`TacEmitter`都实现了`Visitor<>`这一接口。当`Namer`中的一个成员函数调用了`expr.accept(this, ctx)`时，相当于借助`expr`回到了`Namer`中与`expr`相对应的方法。这样设计，每一遍检查的代码可以整合到同一个类中，而不是分散在各个`TreeNode`的派生类类定义中。而且，通过定义不同的`Visitor`子类，各个阶段互相独立，方便修改、扩展。