编译原理 PA2 实验报告

2017011620 计73 李家昊

工作内容

抽象类的支持

实现方法较简单。在 `ClassDef` 中新建一个抽象方法列表 `abstractMethods`，当 `Namer` 分析每个 `Class` 内的方法时，如果这个方法是抽象的，则将方法名加入列表，如果这个方法重载了一个抽象方法，则将方法名从 `abstractMethods` 中移除。当递归完成后回到 `ClassDef` 时，如果此类非抽象且列表非空，则报错。需要注意 `Main` 不能是抽象的，以及抽象类型不能实例化。

局部类型推导

实现方法较简单。在 `Namer` 的 `visitLocalVarDef` 中，若 `typeLit` 为空，说明使用了 `var` 关键字，此时需要建立 `symbol` ，但不能确定类型，这里统一将其类型赋值为 `BuiltInType.NULL`。然后在 `Typer` 的 `visitLocalVarDef` 中，当 `initVal` 被分析后，将 `initVal` 的类型赋给 `var` 类型变量即可。需要注意特判 `void` 类型，新增函数类型 `TLambda` 。

Lambda 表达式

这部分需要对照测例拟合。

首先将 `Typer` 中 `visitCall` 和 `typeCall` 的权限检查部分迁移到 `visitVarSel` 中，对照测例实现 `Call` 的基本功能，并支持任意函数类型的表达式的调用。

然后仿照 `FormalScope` 新建 `LambdaScope` ，仿照 `MethodSymbol` 新建 `LambdaSymbol`，然后开始对照测例处理各种错误。这里举两个例子，其他错误处理拟合测例即可，不再赘述。

1. 对于正在定义的符号的访问权限控制，需要建立一个栈，每次分析 `initVal` 之前将符号名压栈，分析完后将符号名出栈，若 lambda 表达式中用到了栈中的符号，则报错。
2. 对于捕获符号的直接赋值问题，首先需要修改 `lookupBefore` 函数使其支持 `LambdaScope` ，然后在 `Typer` 的 `visitAssign` 中调用 `lookupBefore` 查找 lambda 表达式定义位置之前的非类作用域下的符号，若找到，则说明对捕获变量赋值，报错即可。

接下来进行 blocked lambda 表达式的类型推导，说明文档给出了类型上界算法，对于类型下界的算法，我的实现如下：

1. 给定类型 $t_1, t_2, \cdots, t_n$ ，选择一个非 `null` 的类型 t_k。若都是 `null`，则返回 `null`；
2. 如果 t_k 是基本类型 `int, bool, string, void` 或数组，检查其他类型是否与 t_k 完全等价，如果是则返回 t_k，不是则返回“类型不兼容“；
3. 如果 t_k 是 `ClassType`：
    1. 令 $p = t_k$ ；
    2. 令 $i = 1, 2, \cdots, n$ ，对每个 t_i，若 $t_i <: p$，则令 $p = t_i$；若既非 $t_i <: p$ 也非 $p <: t_i$，则返回“类型不兼容“。
4. 如果 t_k 是 `FunType`，先检查其他类型是否也都是 `FunType`，且形式与 t_k 相同，如果不是，则直接返回“类型不兼容”，否则：
    1. 设 $t_i = FunType([s_{i1}, s_{i2}, \cdots, s_{im}], r_i)$ ；
    2. 求 $[r_1, r_2, \cdots, r_n]$ 的类型**下界**，设其为 R ；

3. 求 $[s_{1i}, s_{2i}, \cdots, s_{ni}]$ 的类型**上界**，设其为 T_i ；
4. 返回 $FunType([T_1, T_2, \cdots, T_m], R)$ ；

按照算法实现即可，难度不大。

遇到的困难及解决方案

本实验需要进行很多的调试工作，如果采用打印调试法，则每次调试都需要重新编译，非常浪费时间。后来发现可以在 java 执行 jar 文件时打开调试端口，比如

```
java -agentlib:jdwp=transport=dt_socket,server=y,address=*:5005 -jar --enable-
preview build/libs/decaf.jar -t PA2 TestCases\S2\abstract-error-1.decaf
```

然后在 idea 中连上 5005 端口即可调试，这样就方便了很多，显著提高了编程效率。

PA2 相关问题

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?

框架使用 `ScopeStack.findWhile` 方法，在作用域栈中自顶向下遍历每个作用域，在每个作用域中查找符号。在符号定义时使用 `ScopeStack.findConflict` 方法判断是否冲突，如果当前作用域是 `LambdaScope` 或 `LocalScope` 或 `FormalScope` ，则查找非类作用域下的符号，否则，查找类作用域中的符号；在符号引用时使用 `ScopeStack.lookupBefore` 方法，查找 `LambdaScope` 或 `LocalScope` 位于当前位置之前的符号，或其他类型作用域中的符号。

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?

对 AST 的第一趟遍历建立了符号表，第二趟遍历进行了类型检查。第一趟遍历确定了字面类型定义的变量的类型，如 `int a; bool b; string c; int(int) d;` 等等，但未确定 `var` 类型；第二趟遍历确定了其他所有节点的类型，包括形如 `1, false, "1"` 的基本表达式，unary/binary表达式，Print表达式，lambda表达式等等。

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。

使用访问者模式，每个 AST 节点继承 `TreeNode` ，重写其 `accept` 方法，内部调用 `Visitor` 中处理此节点的方法，然后 `Namer` 和 `Typer` 继承 `Visitor`，具体实现对不同节点的处理函数。在某一节点的处理函数中，调用其子节点的 `accept` 方法，利用 java 的多态性，就能调用对应类型的处理函数了。