Decaf PA2 实验报告

计 76　张翔　2017011568

2019 年 11 月 11 日

1　实验简述

本阶段实验在 PA1-A 构建的 AST 基础上，进行语义分析（符号和类型检查），从而生成带有符号和类型信息的 AST。

2　主要工作

2.1　抽象类的支持

由于 PA1-A 我对 abstract 的实现是使用 `modifier` 类，在类和方法的定义上已无需多加修改。这里只需要修改方法重写时的检查规则，新的规则为：函数类型兼容的情况下，抽象方法可以被普通或抽象方法重写，普通方法可以被普通方法重写。另外增加了一个未被重写的抽象方法列表，在类继承时，父类的未被重写抽象方法将加入子类的列表，重写时则删除对应表项，从而可以检查非抽象类是否重写了所有抽象方法。

对于 new 抽象类的情形，修改 new 表达式检测类是否为抽象即可。

2.2　Var 本地类型推导

这个实现依赖于对 `Expr` 节点的类型推导，当它的类型被确定后，可以在 `LocalVarDef` 中确定对应变量的类型。

2.3　一等函数

2.3.1　修改 `VarSel` 和 `Call` 节点

这里修改了 `VarSel` 节点，使其支持选择任意可被访问的类成员（变量和函数）以及数组的 `length` 方法，并迁移了 `Call` 的权限检查。对于 `Call` 节点，处理方法变得简单，只需要检查 `Callee` 是否是函数类型、检查调用参数是否匹配即可。

2.3.2　增加 `LambdaScope` 与 `LambdaSymbol`

为了方便，增加了 `LambdaScope` 类，功能与表现上类似于 `FormalScope`，同时也有一些新的成员变量，如记录当前 `return` 语句的列表、记录当前禁止符号（指 Lambda 表达式要赋给的符号，它不能被当前 Lambda 访问）的列表；`LambdaSymbol` 类似于 `MethodSymbol`，它与 `ScopeStack` 的 `currentMethod` 方法兼容，意味着打开一个 `LambdaScope` 后，通过该方法就可以拿到当前的 `LambdaSymbol` 以及它的作用域，这样当访问 `return` 语句时，可

以识别当前是否为 LambdaScope，从而将返回语句加入到返回语句列表中（为返回类型推导奠定基础），同时也可以改变符号引用查找时的行为（屏蔽对禁止符号列表中的符号的访问）。

2.3.3 Lambda 返回类型推导

当访问完 Lambda 表达式体后，可以进行返回类型推导，如果是表达式 Lambda，可直接依赖 Expr 节点的类型确定；如果是块 Lambda，可以访问 LambdaScope 中收集到的返回语句列表，利用指导书中介绍的类型上界（下界）算法来递归确定返回值的类型。类型下界的求法和类型上界对称，唯一不同之处是对 Class 类型的处理，上界的寻找只需要确定列表中所有类在继承树中的最近公共祖先，而下界的存在需要列表中的所有类在继承树中是一条链状结构（因为 Decaf 语言不支持多继承），此时需要寻找这条链中离根节点最远的那个类作为最后的下界。

2.3.4 Lambda 表达式变量捕获的处理

对于 Lambda 表达式内赋值的处理，修改 visitAssign 函数，根据当前作用域是否为 LambdaScope 来决定操作，禁止对捕获的非类作用域的变量赋值；对于禁止访问 Lambda 表达式赋给的符号，则修改 visitVarSel 查找符号名的逻辑，对于禁止符号列表中的符号，直接返回无法寻找到即可。Lambda 表达式发生嵌套时，父 Lambda 的禁止变量表将传递给子 Lambda。这里的禁止变量表只在类型检查阶段有效，从而不会妨碍名称检查部分对于变量名冲突的检查（如 Lambda 表达式内定义了与 Lambda 要赋给的变量同名的符号）。

2.3.5 一些实现上的考量

Lambda 表达式与方法定义类似，都有形参表且可以通过字面量得到其类型。原来框架中 Namer 的实现是将 Lambda 当成普通表达式，而类型检查阶段不做表达式类型推断的工作，不会访问普通表达式，从而 Lambda 表达式的 symbol 变量为空，在 Typer 阶段会引发 NullPointerException。如果在 Typer 阶段再去确定 Lambda 表达式的相关类型，则 Typer 会重复一些 Namer 可以做的名称检查工作，从而使得代码非常 dirty。因此我实现时采用了 Namer 阶段确定 Lambda 表达式参数类型，Typer 阶段推断返回值的模式，而这种实现需要修改 Namer 中所有包含 Expr 的 AST 节点的访问函数，从而确保所有 Lambda 表达式都能在这一阶段被预先处理。

3 思考题

- **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？**

  实验框架将符号信息存在对应作用域类的 symbol 表中（框架里使用 TreeMap 实现，用 HashMap 也可），作用域存在栈中，需要时可在栈中取得当前的开作用域并根据符号名称查找符号。符号定义时的查找是为了防止重定义同名符号，如果当前是 LocalScope 或者 FormalScope，则符号不能与到 FormalScope 为止定义过的或者是全局符号相冲突，如果是 ClassScope 或者是 GlobalScope，则不能与当前开作用域中的任意符号冲突；对于符号引用，则只需检查当前位置之前的开作用域中距离最近的位置是否存在相应符号即可。

- **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？**

  第一遍主要是符号名称、字面类型、方法重写相关的检查，如是否定义了重名的符号，变量声明时是否错用成 `void`，或者方法重写是否符合规范等，同时也确定了 Lambda 表达式的形参类型，这一遍可以确定 `MethodDef`，`VarDef` 以及普通 `LocalVarDef`；第二遍是类型检查、本地类型推导，如检查变量引用是否合法，检查变量初始化时类型是否和定义相匹配，推导 Lambda 表达式返回值类型，推导 var 以及表达式的类型，检查方法的返回值是否和定义兼容，检查函数调用时是否调用了 Callable 表达式等，这一遍可以确定 `Lambda`, var 关键字定义的 `LocalVarDef`, `Expr` 等节点的类型。

- **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。**

  利用访问者模式，通过 Java 的 RTTI 机制将访问操作动态转发到相应的处理函数。这种实现中，AST 的节点类都有一个 `accept` 接口，传入一个 `Visitor` 类，相应的具体类会重写该接口，如 `Lambda` 节点在这里调用 `visitor.visitLambda`，从而可以将访问者的访问操作自动转发到 `visitLambda` 函数。