编译原理PA3 实验报告

————计62 李祥凡 2016011262

————计62 李祥凡 2016011262

实验总述

为README中提到的decaf新语言特性实现了抽象语法树扫描，并在扫描翻译的过程中生成tac语句，使得翻译产生的中间tac代码能被执行并输出正确的结果。

具体实现

1. 类的浅复制的支持

scopy(dst, src)

在TranPass2.java中实现了visitOCStmt方法，会调用Translater.java中的新实现的genOCStmt方法，会使用genLoad方法将src类中的内存逐4字节拷贝到dst类的内存空间中，内存空间中的每4个字节可能是基本数据类型，可能是虚函数表指针，也可能是Class的指针，不管是什么，直接复制过来便可实现类的浅复制 (基本数据类型的拷贝是一份数值相同的新数据；指针的拷贝会得到指向同一目标的新指针)

这里我对之前实现的OCStmt语法节点做了一些修改，使scopy的第一个参数不再以String的类型传递，而是以Ident类的形式传递 (由于这里scopy的第一个参数只会是一个标识符而不会是类的成员变量，故Ident节点生成的时候直接将它的Receiver置为null即可)，这样修改是为了便于在语法节点生成的过程中获得第一个参数对应的类的信息，比如我必须知道这个类的内存空间大小以完成内存的复制，而dst(Ident类型)对应的Class的内存大小可由

((ClassType)(((Expr)(dst)).type)).getSymbol().getSize()来获得。

2. sealed的支持

README中说任何类不能以sealed class为父类，这一语法规定的判断与报错在PA2已经给予了实现，PA3中便没有对sealed作任何处理。

3. 支持串行条件卫士语句

if { E1 : S1 ||| E2 : S2 ||| ... ||| En : Sn } ;

```java
    @Override
    public void visitGuardStmt(Tree.GuardStmt guardstmt) {

        for(Tree.IfSubStmt ifSubStmt : guardstmt.lst ) {
            ifSubStmt.accept(this);
        }
    }

    @Override
    public void visitIfSubStmt(Tree.IfSubStmt ifSubStmt) {

        ifSubStmt.expr.accept(this);
        if (ifSubStmt.stmt != null) {
            Label exit = Label.createLabel();
            tr.genBeqz(ifSubStmt.expr.val, exit);
            if (ifSubStmt.stmt != null) {
                ifSubStmt.stmt.accept(this);
            }
            tr.genMark(exit);
        }
    }
```

如上图所示，在TranPass2.java中实现了visitGuardStmt和visitIfSubStmt方法，visitGuardStmt方法会对其中的每个分支 (E_i ：S_i) 依次调用visitIfSubStmt方法，ifSubStmt会判断E_i是否成立(条件的val是否为非0)，若成立，执行之后的S_i生成的汇编代码，若不成立，跳转到S_i生成的汇编代码后的exit标号处，执行在此之后的条件判断(E_{i+1} ：S_{i+1})。

4. 支持简单的类型推导

```java
    @Override
    public void visitVarIdent(Tree.VarIdent varIdent) {
        Temp t = Temp.createTempI4();
        t.sym = varIdent.symbol;
        varIdent.symbol.setTemp(t);
        varIdent.val = t;
    }
```

如上图所示，在TranPass2.java中添加了visitVarIdent方法，由于VAR IDENTIFIER同时进行了变量的声明和使用，在visitVarIdent方法中需要同时实现visitVarDef和visitIdent方法中的操作，创造一个4字节的Temp对象并与varIdent的symbol相关联，并将varIdent的val置为该Temp对象。

5. 支持若干与一维数组有关的表达式或语句。

1） 数组初始化常量表达式，形如

E %% n

在TranPass2.java中的visitBinary方法中添加了%%操作符，会调用Translater.java中新增的genArrayRepeat方法:

```java
    public Temp genArrayRepeat(Type type, Temp element, Temp length) {
```

genArrayRepeat方法的参数为 数组元素类型（type)，数组元素对应的Temp对象 (element)，以及数组长度对应的Temp对象 (length)。

该方法调用系统库函数分配一个长度为length+1的内存空间，并将length存在空间的头部，之后会生成tac语句来构造一个循环，依次复制length个数的element到该内存空间中。

这里对Class类型和非Class类型所做的处理是不同的，type参数传递了数组元素的类型

若type为ClassType，则调用前面实现的浅复制方法genOCStmt生成一个element的拷贝，将该拷贝放入对应的内存空间中

若type不是ClassType，则将element自身直接放入对应的位置即可，这样数组中的每个元素已经是相互独立的了。

```
449         if(type.isClassType())
450         {
451
452             Temp clone = Temp.createTempI4();
453             genOCStmt(((ClassType)(type)).getSymbol(),clone,element);
454             genStore(clone,obj,0);
455
456         }
457         else {
458             genStore(element, obj, 0);
459         }
```

2） 数组下标动态访问表达式

E [E1] default E’

在TranPass2.java中添加了visitIndexedDefault方法，访问了E、E1、E' 后调用Translater.java中新增的genIndexedDefault方法

genIndexedDefault方法中生成的tac语句会根据索引E1是否合法 (是否大于等于0且小于数组长度)来选择是跳转到选择 E[E1]的标号处还是跳转至选择 E‘ 的标号处。

3） 数组迭代语句，形如

foreach (var x in E while B) S

或者

foreach (Type x in E while B) S （这里，Type 为用户指定类型）

在TranPass2.java中添加了visitForeachStmt方法，直接产生该语句的tac代码，没有为该语句在Translater.java中新增方法。

这部分的处理我借鉴了visitForLoop的实现，因为本质上foreach语句与for循环是很相似的。

首先，在循环开始前，访问变量 x 和数组 E，获得它们的 val 值，之后跳转至 cond 标号处执行 B 条件的判断(若没有while，则B是为 true的) 以及数组索引未越界的判断 (是否尚未扫描至数组E的末尾)，条件判断结果若为假，则跳转至最后的exit标号处跳出循环，若为真，则跳转至loop标号处，将E中的对应元素赋值给x，执行loop标号后的S中所有语句对应的tac语句。

这里，有一个Temp对象作为指针用来记录当前扫描的E中元素位置，每轮循环开始前，要将该T指针向后移4个字节，并将所指的元素赋值给 x，更新x的值，每轮循环结束后，都会跳转至cond标号处执行一次条件判断。

这里要特别注意是不能访问未分配的内存空间的，会报错，所以扫描至数组E的末元素后要特判一下，不能再往后扫描一个位置了，应该直接退出。

在foreach循环中要支持break语句，这里的实现我直接模仿了visitForLoop中对break语句的支持，即在foreach循环开始时将退出标号exit放入标号栈loopExits中，在foreach语句退出时从loopExits栈中pop出这个exit标号。

实验总结

本阶段的工作量比前几个阶段都大得多，一开始我没有理解整个Translater的架构，就匆忙地开始做，导致浪费了不少时间。后面了解了生成并执行tac语句的机制后就好办多了。在本次PA过程中发现了前几个阶段没有暴露出来的一些实现的疏漏之处，比如说PA2中没有实现对foreach中对break语句的支持等。