PA3 实验报告

计科70 徐明宽 2017011310

工作内容

动态错误检查

我在TacEmitter.java中参考`visitClassCast`中对运行时错误的检测，在`visitBinary`中判断运算符为除号或取模时新建了一个label `ok`，当`rhs.val`不为零时直接跳转到`ok`，否则输出错误信息（在RuntimeError.java中加入了`String DIVISION_BY_ZERO`）并停止运行。

抽象类

我将TacGen.java中遍历函数体的语句加入了`ifPresent`判断。

局部类型推导

无。

First-class Functions

扩展call

我在TacEmitter.java的`visitVarSel`中，利用PA2实现好的`expr.isMethod`与`expr.isStaticMethod`（这里发现Typer.java中有一处未对`expr.isStaticMethod`和`expr.receiverClassName`赋值，且未写`expr.setThis();`，还有一处未对`expr.isStaticMethod`赋值，便将它们补上），对于static的情况申请8字节内存，存储`0`和函数指针；对于non-static的情况申请12字节内存，存储`1`、函数指针和this（即`object.val`）。对于non-static的情况，我在FuncVisitor.java中实现了`visitFuncEntry`函数以通过`Temp object, String clazz, String method`获取函数指针；对于static的情况，我在此时暂未实现获取函数指针的代码。

在FuncVisitor.java中仿照`public Temp visitMemberCall(Temp object, String clazz, String method, List<Temp> args, boolean needReturn)`实现了Call节点仅需的`public Temp visitCall(Temp entry, List<Temp> args, boolean needReturn)`函数之后，我在TacEmitter.java的`visitCall`函数中调用了`expr.methodExpr.accept(this, mv);`并对`methodExpr`（可能是）`VarSel`节点返回的`val`对应地址的第一个元素进行了判断，如果是`0`则为static情况，`1`则为non-static情况，后者需要向参数列表的开头加入this指针。与实验说明"当检测到并非方法调用时"不同，我并没有沿用原有的逻辑（毕竟PA1都把Call节点的`receiver`删了啊），而是全部重写。

将方法名直接当做函数使用

我发现TacGen.java中无法直接新建虚表，于是在ProgramWriter.java的`class Context`新建了（全局的）虚表`staticVtbl`，类名为`static`（使用Decaf的关键字即可不和任何一个类重名），而存储的"成员"函数名为`className + "." + funcName`。因此，我在`putOffsets`中对这个虚表进行了特判，以避免函数在`Map`中被存为`"static." + funcName`导致重名。然后我发现ProgramWriter.java又无法判断一个方法是不是静态的，于是仅实现了函数`visitStaticMethod`，而在TacGen.java中对静态方法进行判断。最后，由于静态方法的虚表也需要进行`putVTable`和`putOffsets`，我在

ProgramWriter.java中实现了函数`visitVTablesPostProcess`，并在TacGen.java中调用，以完成建虚表的这些后处理工作。

建好静态方法的虚表后，我便可以在TacEmitter.java的`visitVarSel`中获取静态方法的函数指针了：只需在FuncVisitor.java中再写一个`visitFuncEntry`，将`visitLoadFrom(object)`替换成`visitLoadVTable("static")`即可。

接下来，我遇到了本次实验中的一个**困难**：模拟器找不到`main`方法了。经研究代码，我发现FuncLabel.java中对`main`方法的标签有特判：并不是写做`String.format("_L_%s_%s", clazz, method)`的形式，而是直接写成`"main"`。并且，这也不只是在`prettyString`中这么做，而是在Simulator.java的`private Map<String, Integer> _label_to_addr`中亦为如此。然而，Simulator.java又到这个`Map`中去找了`_L_Main_main`（因为我在静态方法的虚表里用的是这个），那自然是找不到了。

如何解决呢？我认为，在Decaf语言中，`main`方法与其他方法并没有多少不同，同样可以被调用，同样可以被赋值给变量，同样可以直接当做函数使用。因此，我不认为每当想找一个FuncLabel时都应对`main`进行特判，而是应该将`main`的label改为与其他静态方法格式相同。修改了FuncLabel.java中的相关逻辑后这个问题得以解决。

模拟器能运行后，我在测试以下Decaf程序时又遇到了问题：输出为`truefalse`。

```
class Main {
    static void main() {
        Print(even(0));
    }
    static bool even(int x) {
        Print(x % 2 == 0);
        return x % 2 == 0;
    }
}
```

经调试，我发现问题出在我在TacEmitter.java的`visitCall`中（参考原有的代码，直接）对静态方法和非静态方法分别写了`expr.val = ...`，而两次这样的“赋值”在tac代码里是不会赋给同一个临时变量的。我改为`expr.val = mv.freshTemp()`和两次`mv.visitAssign(expr.val, ...)`从而解决了问题。

Lambda表达式

为了精确地知道lambda表达式捕获了哪些变量，我参考自己在PA2阶段在Typer.java的`visitAssign`中实现的相关逻辑，在LambdaScope.java中新增了变量`public final Scope parent`和`private Map<String, VarSymbol> capturedVar`、方法`public void capture(VarSymbol varSymbol)`和`public List<VarSymbol> capturedVars()`，在`visitVarSel`中利用作用域栈获取了哪些`LambdaScope`应该捕获当前`varSel`的变量，而没有像实验说明推荐的那样额外保存一个lambda表达式栈。为方便调试，在PrettyScope.java中加一行`lambdaScope.capturedVar.values().forEach(printer::println);`即可输出所有被Lambda表达式捕获的变量。

值得注意的是，对于`this`中的field，要捕获的是`this`，因此在Typer.java的`visitVarSel`中执行完`expr.setThis();`后应将待捕获的变量改为`this`。

为了新建lambda的虚表，我在GlobalScope.java中新增了变量`public List<LambdaSymbol> lambdaSymbols`，在Namer.java中即将所有`LambdaSymbol`统计了下来。这样，在ProgramWriter.java中新建虚表`lambdaVtbl`（类名为`fun`，使用Decaf的关键字即可不和任何一个类重名）、新增函数`visitLambda`后，我便可以在TacGen.java中调用ProgramWriter.java的

`visitLambda`以建立lambda的虚表，在FuncVisitor.java中再写一个`visitFuncEntry`，在TacEmitter.java的`visitLambda`中申请`(3 + 被捕获的变量数目) * 4`个字节的内存，依次存储`2`、函数指针、被捕获的变量数目，以及按照`pos`的顺序存储每个被捕获的变量。注意我并没有对捕获this的情况进行特判。

在TacEmitter.java的`visitCall`中的lambda表达式这一情况，我使用了`emitWhile`与FuncVisitor.java中新实现的`visitParm`对被捕获的变量进行了传参。

在TacEmitter.java的`visitLambda`中，我需要实现lambda表达式的函数体，又不能破坏声明lambda表达式的函数的原有框架。于是，我通过无条件跳转指令获得了一块不会被访问到的tac代码空间，在这块空间里调用FuncVisitor.java中新实现的`visitLambdaLabel`，将形参的`temp`传入，再实现lambda表达式的函数体。

为了判断当前是在哪个lambda表达式内部，我在TacEmitter.java中新增了变量`Stack<Tree.Lambda> lambdaStack`，如果在某个lambda表达式内部的话在TacEmitter.java的`visitVarSel`和`visitThis`中暴力查找待访问的变量被捕获的位置（如果被捕获的话）。当然，这里可以实现被优化为使用`Map`做到每次`O(log(被捕获的变量个数))`的时间复杂度，但被捕获的变量个数普遍较少，加上实验并不关心运行速度，暴力即有足够好的效果。

这样实现后，我又遇到了一个困难：lambda表达式的函数并不能顺利地按照预想的方式被调用，出现的问题包括但不限于当捕获的变量个数多于外层函数的参数个数时在FuncVisitor.java里`getArgTemp`里访问`argsTemps[index]`会数组越界。解决方式显然应该是为每个lambda表达式新开一个FuncVisitor，即*真正*地新生成一个函数，这只需在FuncVisitor.java中实现一个函数`visitLambdaFunc`（以替换掉之前写的`visitLambdaLabel`，也就是说对于lambda表达式只获得一个FuncLabel对于TacEmitter.java来说并不够用）返回一个新的`FuncVisitor`即可。除此以外，对于实验说明里所写的静态与非静态方法的“新生成一个函数”，均只是开了几个字节的内存存储函数指针等信息罢了，并不需要新开FuncVisitor。

拓展：数组长度的情况

我额外实现了将数组长度当做函数使用的情况。在TacEmitter.java的`visitVarSel`中，对于这种情况，我申请8字节内存，存储`3`和数组指针；在`visitCall`中，我对于这种情况直接`mv.visitAssign(expr.val, mv.visitLoadFrom(entry, -4));`即可。

回答问题

Q1. lambda语法实现的流程?

见上文"Lambda表达式"一节。

Q2. 实现工程中遇到的困难?

见上文“将方法名直接当做函数使用”一节与"Lambda表达式"一节。

致谢

感谢罗承扬同学与我的讨论。

声明

我于2019年12月15日将代码和报告交给了傅舟涛同学参考。