李墨珩

计74 2017050025

2019年12月17日

编译原理

PA3 实验报告

新特性0 除零错误

要求当出现“处于零”或是“对0求mod”的时候进行报错，根据提示，参考框架里的两种关于数组的运行时错误检测，在visitBinary（）里添加判断。如图所示：

```
expr.lhs.accept(this, mv);
expr.rhs.accept(this, mv);
if(expr.op.equals(Tree.BinaryOp.DIV) || expr.op.equals(Tree.BinaryOp.MOD)) {
    var z = mv.visitLoad(0);
    var err = mv.visitBinary(TacInstr.Binary.Op.EQU, expr.rhs.val, z);
    var h = new Consumer<FuncVisitor>() {
        @Override
        public void accept(FuncVisitor v){
            v.visitPrint(RuntimeError.CLASS_DIVIDED_BY_0);
            v.visitIntrinsicCall(Intrinsic.HALT);
        }
    };
    emitIfThen(err, h, mv);
}
```

新特性1 抽象类和抽象函数

在TacGen.java里，在接受函数体前加一个判断。如下

```
}
if(!method.isAbstract()) { method.body.get().accept(this, mv); }
mv.visitEnd();
int cnt = 1;
```

新特性3 First-class Functions

扩展call： TacEmitter里原本的visitCall是只处理普通方法的情况。可是对于lambda表达式和函数对象，Tree.Call.method可能是 Tree.VarSel, Tree.Lambda 和 Tree.Call, 而Tree.Call.symbol 对应的 symbol 有 VarSymbol, MethodSymbol 和 LambdaSymbol。于是需要在TacEmitter.visitCall里添加判断逻辑，根据提示，对两种寒素对象都分配在堆上，偏移量0处，都存一个函数指针。 首先，先在FuncVisitor里增加函数visitFunCall，而偏移量为0处的函数指针做为函数的调用入口，并将对象与剩下的参数传入，并调用enter。

而对于将方法名直接当做函数使用 则基于原本的TacEmitter.visirVarSel()只处理VarSymbol的情况，所以需要增加处理MethodSymbol。根据提示，主要分为两种方法：非static和static。

首先，对于静态方法，为了实现动态分发函数变量，需要先建立一个虚表，在ProgramWriter里实现，名为“myvirtualvtable”。参考vtable的写法，在ProgramWriter.Context里也要对offset进行维护，此外也在FuncVisitor里添加getProgramWriter对ProgramWriter进行访问。对于非静态方法，增加一个函数，加入全局虚表并设置offset。

在函数被建立后，分配8个字节的内存，在偏移量0处存新建函数的指针，而在偏移量为4的地方存对象指针。对于静态的，不同点是不能透过虚表来查询，而是通过ctx. getFuncLabel(),后再通过DirectCall()来调用原函数。

```
@Override
default void visitCall(Tree.Call expr, FuncVisitor mv) {
    if (expr.isArrayLength) {
        var array = expr.receiver.get();
        array.accept(this, mv);
        expr.val = mv.visitLoadFrom(array.val, -4);
    } else {

        expr.args.forEach(arg -> arg.accept(this, mv));
        var temps = new ArrayList<Temp>();
        expr.args.forEach(arg -> temps.add(arg.val));
        expr.receiver.get().accept(this, mv);
        // System.out.println(expr);
        if(expr.symbol.isMethodSymbol() && expr.receiver.get() instanceof Tree.VarSel) {
            if(((MethodSymbol)expr.symbol).isStatic()) {
                if(((MethodSymbol)expr.symbol).type.returnType.isVoidType()) {
                    mv.visitStaticCall(((MethodSymbol)expr.symbol).owner.name, expr.symbol.name, temps);
                } else {
                    expr.val = mv.visitStaticCall(((MethodSymbol)expr.symbol).owner.name, expr.symbol.name, temps, true);
                }
            } else {
                var ob = ((Tree.VarSel)expr.receiver.get()).receiver.get();
                if (((MethodSymbol)expr.symbol).type.returnType.isVoidType()) {
                    mv.visitMemberCall(ob.val, ((MethodSymbol)expr.symbol).owner.name, expr.symbol.name, temps);
                } else {
                    expr.val = mv.visitMemberCall(ob.val, ((MethodSymbol)expr.symbol).owner.name, expr.symbol.name, temps, true);
                }
            }

        } else {
            FunType fun;
            if (expr.symbol.isVarSymbol()) {
                fun = (FunType)((VarSymbol)expr.symbol).type;
            } else if (expr.symbol.isLambdaSymbol()) {
                fun = ((LambdaSymbol) expr.symbol).funtype;
            } else {
                fun = ((MethodSymbol) expr.symbol).type;
            }
            if (fun.returnType.isVoidType()) {
                mv.visitFuncCall(expr.receiver.get().val, expr.symbol.name, temps);
            } else {
                expr.val = mv.visitFuncCall(expr.receiver.get().val, expr.symbol.name, temps, true);
            }
        }
    }
}
```

```
if(!lambdaStack.isEmpty()) {
    var m = lambdaStack.peek();
    if(expr.symbol.isVarSymbol() && ((VarSymbol)expr.symbol).isMemberVar() ) {
        var ob = mv.visitLoadFrom(mv.getArgTemp(0), 4);
        expr.val = mv.visitMemberAccess(ob, ((VarSymbol)expr.symbol).getOwner().name, expr.name);
        return;
    } else if(m.currSymbol.containsKey(expr.symbol)) {
        expr.val = mv.visitLoadFrom(mv.getArgTemp(0), m.currSymbol.get(expr.symbol));
        return;
    }
}

if(expr.symbol.isVarSymbol()) {
    if(((VarSymbol)expr.symbol).isMemberVar()) {
        var ob = expr.receiver.get();
        ob.accept(this, mv);
        expr.val = mv.visitMemberAccess(ob.val, ((VarSymbol)expr.symbol).getOwner().name, expr.name);
    } else {
        expr.val = ((VarSymbol)expr.symbol).temp;
        System.out.println("form tac");
    }
} else if(expr.symbol.isMethodSymbol()) {
    var methods = ((MethodSymbol)expr.symbol);
    if(methods.isStatic()) {

        mv.getProgramWriter().addFuncLab(".globl::" + methods.getOwner().name, methods.name);
        var newFunc = mv.getProgramWriter().visitFunc(".globl::" + methods.getOwner().name, methods.name, methods.type.arity() + 1);
        newFunc.visitVarSelStatic(methods.getOwner().name, methods.name, !methods.type.returnType.isVoidType(), methods.type.arity() + 1);
        newFunc.visitEnd();
        var addr = mv.visitIntrinsicCall(Intrinsic.ALLOCATE,true, mv.visitLoad(4));
        var vt = mv.visitLoadVTable(".globl");
        var funcVal = mv.visitMemberAccess(vt, ".globl::" + methods.getOwner().name, methods.name);
        mv.visitStoreTo(addr, funcVal);
        expr.val = addr;
        System.out.println("form tac");
    } else {

        var ob = expr.receiver.get();
        ob.accept(this,mv);
        mv.getProgramWriter().addFuncLab(".globl::" + methods.getOwner().name, methods.name);
        var newFunc = mv.getProgramWriter().visitFunc(".globl::" + methods.getOwner().name, methods.name, methods.type.arity() + 1);
        newFunc.visitVarSel(methods.getOwner().name,methods.name, methods.type.returnType != null, methods.type.arity() + 1);
        newFunc.visitEnd();
        var alloc = mv.visitIntrinsicCall(Intrinsic.ALLOCATE, true, mv.visitLoad(8));
        var vv = mv.visitLoadVTable(".globl");
        var FunctionPoint = mv.visitMemberAccess(vv, ".globl::" + methods.getOwner().name, methods.name);
        mv.visitStoreTo(alloc, FunctionPoint);
        mv.visitStoreTo(alloc, 4, ob.val);
        expr.val = alloc;
    }
}
```

新特性4 Lambda表达式

这一部分需要改动和完善pa2中的部分内容。首先，先在Typer.java里添加额外的栈来记录lambda表达式，在Tree.NewLambda里添加变量，用来记录捕获变量和所需变量的信息，来满足pa3的需求，加capture（）函数，用来判断此变量是否需要被捕获。捕获分为两种，对于局部变量，就从栈顶开始加入变量，而对于this或是当前域，就需要将当前的lambdastack里的所有lambda的needed设为true。

接着，在TacEmitter里添加对对visitLambda函数的重载，构造函数对象。首先，先申请内存，后新建函数，设置offset，加入虚表里。访问函数体，并加入返回。最后通过全局虚表获得函数指针，存在内存偏移量为0的地方。