编译原理PA3

实验内容

运行时错误检查：除以零

此任务的目标是向生成的TAC代码中增加错误处理部分。需要修改的函数是`TacEmitter::visitBinary`，对于`DIV` `MOD`两种运算符，使用`emitIfThen`来产生判断被除数是否等于0的语句，如果是则打印运行时错误信息并退出程序。框架其他程序保持不变。

> 这种办法是修改生成的程序来处理错误，而不是模拟器支持对运行时错误的处理，这就与Java虚拟机有着本质的不同，更像是我在代码中加了assert。

扩展Call和构建函数对象

就我个人而言，这两部分内容是交叉完成的，因为我构造函数入口地址的过程并不顺利，直到我发现可以通过`LoadVTbl`实现对函数入口地址的获取。

对现有框架的修改体现在`TacEmitter::visitCall`与`TacEmitter::visitVarSel`。前者不需要对被调用的函数分类讨论，因为它的`VarSel`一定能返回一个函数对象，因此`visitCall`的任务就是取出这个对象的首地址处的函数入口地址，然后把这个对象作为唯一参数，发射一条`IndirectCall`指令。`TacEmitter::visitVarSel`则需要对成员方法、静态方法、`length`函数变量分类讨论，对于前三者，构造不同的函数对象。

这些函数对象包装的函数入口地址是编译器自行创建的辅助函数，这些函数与各自的函数唯一对应，接收一个函数对象和其他传入参数作为形参，调用它对应的那个成员函数或静态函数。我既需要创建函数体，又需要创建生成函数对象的代码，总结起来做了如下的事情：

- 入口标记`FuncLabel`：需要在创建类的虚表时一同创建，故我扩充了`ProgramWriter::visitVTables`、`ProgramWriter::buildVTableFor`。这些入口标记需要加入到`ProgramWriter.Context`中，也要加入到每个类一张、单独的虚表里，这张新表我命名为`原类名+`，Label的名字保持被调用者的函数名。需要为这个类的（包括继承自基类的）每一个成员函数与它的静态函数创建一个辅助函数，虽然抽象函数不必创建但是也不要紧。`array.length`需要单独处理，因为这个函数的指令对于任意的数组都是一样的，因此我单独为其创建了一张虚表。
- 函数体：我为`ProgramWriter`增加`buildArrayLength`和`visitFuncObject`，为`FuncVisitor`增加函数`visitStaticFuncCaller`和`visitMemberFuncCaller`，处理了三种情况下的“函数调用”函数，后两者都可以在已有函数基础上扩充，前者也可以把`TacEmitter::visitCall`中的代码搬运过来。容易踩坑的地方在于成员函数调用，要从`this`的虚表中通过偏移量获取函数地址，不可以使用`FuncLabel`与`DirectCall`，**而测试样例中恰恰缺少这样的测试点。**
- 函数对象：我在`TacEmitter`中构造了`emitStaticFunctionObj`、`emitMemberFunctionObj`与`emitArrayLengthFuncObj`函数。加载函数地址是这样实现的：通过被调用者（`MethodSymbol`的`owner`）类型加载其虚表`LoadVTbl`，再加载被调用的调用者函数offset，就可以通过`LoadFrom`加载出真正的函数地址。另，原框架的`LoadFrom`只能是一个`Temp base`与一个`int offset`，但是很容易就可以改成`Temp base, Temp offset`，从而可以一步完成函数入口的加载。

Lambda语法实现流程

确定捕获的变量

这一工作需要在PA3阶段之前完成，因为在翻译为TAC码之前，需要事先确定被捕获的变量有哪些，被捕获的变量有多少个。首先确定，需要捕获的只有局部变量和This指针，成员变量、成员函数、静态函数不需要捕获。于是修改`Typer`的`visitVarSel`和`visitThis`，处理单独的一个This、带`This`的成员变量或函数、没有`receiver`的变量。

> Lambda表达式中单独的This、成员函数的引用是容易被忽略的情形，而不幸的是忽略了这二者都可以通过所有的测试点，足见测试点之弱。下面提供一个加强的测试点：

```
class Main {
    void ok1() {
        Print("ok1\n");
    }
    void emmm() {
        var test1 = fun() {
            ok1();
        };
        var test2 = fun() => this;
        test1();
        test2().ok1();
    }
    static void main() {
        (new Main()).emmm();
    }
}
```

此外，变量可能是跨层捕获的，此时变量需要逐层传入，在每一层都需要出现。比如，最外层第一层的Lambda表达式定义了局部变量`x`，第二层没有引用，作为最内层的第三层引用了，如果第二层不捕获，第三层就会出错。因此，当`Typer`完成了一个Lambda表达式的处理之后，需要把这个表达式的捕获变量加入到外层表达式中。在这个过程中，如果发现一个变量就是在这个外层表达式的作用域内直接定义的（非嵌套），那么需要把它从捕获列表中移出，因为一个局部变量的捕获范围是定义它的那一层到最靠内的使用者之间。

在实践中，因为需要对`This`进行特判，我重载了`VarSymbol`的`equals`函数。这样，这个列表在第三阶段中不会改变，相当于每一个符号被分配了唯一的一个编号，可以在产生TAC代码过程中用于产生参数的序号。

生成Lamdba函数体

生成函数体的行为与`TacEmitter`大体是一样的，除了变量引用会有所区别。因为产生的代码在一个新的函数体内，因此要由`FuncVisitor`生成一个新的`FuncLabel`（`TacEmitter`没有对`ctx`的访问权限），交由一个新的`FuncVisitor`生成Tac码。对于Lambda表达式，我编写了`TacEmitter`的子类`LambdaEmitter`，重载了`visitThis`与`visitVarSel`，还可以存储对应的`LambdaSymbol`和参数个数`argCount`。

`LambdaEmitter::visitVarSel`的不同之处在于，会先检查当前这个局部变量是否是在当前Lambda内定义的，如果是，那就从被捕获的变量列表中找到它的标号，返回对应位置的`ArgTemp`；其他处理完全一样。`visitThis`会从捕获变量列表中寻找`this`。

生成Lambda调用者函数

这个函数的地址是将来要放在函数对象中的，因此这个调用者函数需要位于一个虚表中。我专门为其创建了一张叫做`LambdaCaller$`的表，这个表中的函数是动态更新的，每插入一个函数就要更新一下Offset。我在`FuncVisitor`中添加了函数`buildLambdaFuncCaller`来处理它的创建。和普通函数的调用者函数创建类似，把接受的参数传入，再把函数对象中包含的参数作为被捕获的变量值传入，直接调用Lambda函数即可。

生成Lambda函数对象

Lambda函数对象是在`visitLambda`的过程中生成的。首先考虑如何为被捕获的变量"喂"进去正确的值。问题是在Lambda表达式中，待捕获的变量可能是"被捕获"的，这样不能从`VarSymbol.temp`中取出值，而是应该从参数列表中寻找。针对这个问题，普通的函数与Lambda表达式有着不同的处理。我在`TacEmitter`中添加函数`List<Temp> getTempForCaptured(List<VarSymbol> captured, FuncVisitor mv)`，根据捕获的变量符号寻找对应的Temp。在`TacEmitter`中，只有`this`是需要单独处理的；而在`LambdaEmitter`中，`this`和当前`LambdaScope`中未定义的符号都是要在参数列表中查找的。

然后就可以构造函数对象了。分配`4 * (1 + CapturedArgsCount)`个字节的对象，首位置存放函数入口（仍然通过虚表获取），后面依次存放各个参数的值，即可完成构造。

遇到的困难

主要的困难是在理解如何获取函数指针上面！首先我把函数的地址理解成了一种编译期就可以确定的东西，不同于`FuncLabel`这种只有一个名字，而是有一个数值可以精确标定入口地址。于是我在找哪些接口可以提供函数入口地址，结果就没有找到……再往下走，就要涉及`Simulator`的原理了，然而这肯定不是助教认为我们有必要了解的，于是在这里卡住了……最后回到实验说明文档，才发现之前似懂非懂的一小段提示成为了突破口，于是我一拍脑袋，记录一下函数地址在虚表中的Offset不就可以了嘛！直到我发现这样我仍不能确定是在哪一个虚表中……再看指导书，发现原来有根据类名加载虚表的指令，而且支持跳转到某个寄存器中的地址，到这我才发现该如何获取函数地址……

还有一个就是如何处理捕获的变量。实验说明阐释了"做什么"，但是没讲清楚"怎么做"，而是只给了一个大概的方向，比如它神秘兮兮地说"推荐保存一个Lambda表达式栈"，但是没说这个栈保存了又有什么用，感觉像打哑谜。~~最后我保留了也没用上。~~还是我自己随便搞了一种办法。这里其实并不算困难，但是我起初真的是一脸懵圈，直到后来我从头自己分析了一遍，并且逐渐地把想法落实成代码，再回过头来看实验说明，才发现我实现的并不一样。

这就是我遇到的困难。最后以正能量语录结尾：

> 我们无论遇到什么困难……也不要怕！微笑着面对它！