# 编译原理 PA3 实验报告

2017011620 计 73 李家昊

2019 年 12 月 6 日

## 1 工作内容

### 1.1 除零检查

仿照数组初始化时的长度检查实现即可。

### 1.2 扩展 Call

个人认为这一部分的说明文档并不是很清楚，这里用伪代码重新叙述一遍。

#### 1.2.1 统一调用逻辑

非方法调用时，设调用对象为 `object`，并且 `*object = entry`，则将调用 `object(args)` 解释成 `entry(object, args)`。

#### 1.2.2 方法名直接调用

支持将方法名直接当作函数调用，分为以下三种情况：

**非静态成员函数** 若发现 `VarSel` 是非静态成员函数 `foo`，则执行以下逻辑：

```
object = Alloc(8)
*object = entry
*(object + 4) = this
```

注意调用 `object(args)` 会被解释成 `entry(object, args)`，其中 `entry` 为新生成的函数，定义如下：

```
def entry(object, args):
    this = *(object + 4)
    return foo(this, args)
```

**静态成员函数** 若发现 `VarSel` 是静态成员函数 `foo`，则执行以下逻辑：

```
object = Alloc(4)
*object = entry
```

其中 `entry` 为新生成的函数，定义如下：

```
def entry(object, args):
    return foo(args)
```

**数组长度** 虽然不考察，但还是实现了。若发现 `VarSel` 是数组 `foo` 的长度函数，则执行以下逻辑：

```
object = Alloc(8)
*object = entry
*(object + 4) = foo.length()
```

其中 `entry` 为新生成的函数，定义如下：

```
def entry(object):
    return *(object + 4)
```

#### 1.2.3 Lambda 表达式

请参见 Section 2.1。

# 2 PA3 相关问题

## 2.1 Lambda 语法实现的流程

1. 获取 Lambda 表达式捕获的所有变量名

   在 `LambdaSymbol` 内增加一个表，记录所有捕获变量的变量名，表内变量名的顺序就是它们在内存上的最终存储顺序，在 `Typer` 的 `visitVarSel` 中判断并获取当前 Lambda 表达式的所有捕获变量，注意对于成员变量则需要捕获 `this`，对于每一个捕获变量，遍历每个开 Lambda 作用域的 Lambda 符号，若其捕获变量表未包含该变量，则将该变量加入其中。

2. 改变捕获变量的访存地址

   在 `LambdaSymbol` 内增加一个哈希表，记录变量名和寄存器的对应关系，在 `TacEmitter` 中维护一个 Lambda 符号栈，每次开始创建 Lambda 表达式函数体时，将其 Lambda 符号压栈，创建结束后出栈。修改 `visitThis` 和 `visitVarSel` 函数，若 Lambda 符号栈为空，说明当前不在 Lambda 作用

域内，采用原框架的处理即可，否则，在当前 `LambdaSymbol` 的捕获变量表中查找该变量名的索引，记为 $i$，则取出第一个参数偏移量 $4+4i$ 处的值，这就是该变量的地址，同时，需要更新栈内所有 `LambdaSymbol` 的哈希表。

3. 创建 Lambda 表达式函数体

   为了避免重名，Lambda 表达式统一命名为它们的位置。分析到 Lambda 表达式时，首先为其创建函数标签，并添加进全局虚表，更新 `offsets`，以便间接调用时能获取其地址；然后通过 `ProgramWriter` 实例获取一个新的 `FuncVisitor`，创建 Lambda 表达式对应的函数体。

4. 创建 Lambda 对象

   创建完 Lambda 表达式函数体后，Lambda 符号已经出栈。设 Lambda 表达式捕获变量个数为 $n$，则首先为 Lambda 对象申请 $4+4n$ 的内存空间；然后通过全局虚表获取 Lambda 表达式函数体入口，存储在偏移量 0 处；最后按顺序遍历捕获变量表，对于第 $i$ 个变量，若当前仍处于 Lambda 作用域内且上一级 Lambda 表达式包含此捕获变量，则将上一级 Lambda 表达式捕获的变量拷贝到当前 Lambda 对象的偏移量 $4+4i$ 处，否则，利用哈希表找到其对应的寄存器，将其存储到同样的位置。

## 2.2 工程中遇到的困难

实验说明有充分的提示，总体来说难度不大，稍微有些困难的地方在于：

1. 原框架对静态函数是直接调用的，正常来说无法获取其地址，而需要创建一个虚表才能间接获取其地址。具体实现时只创建了一个全局虚表，包含所有类的静态成员函数，Lambda 表达式，以及数组长度函数。

2. 原框架的虚表名称必须与类名一致，否则在创建和访问 `offsets` 的时候会出问题。这里将 `offsets` 的 `key` 改成了 `tableName.className.methodName` 的形式，从而支持全局虚表。

3. 一个比较坑的地方是对嵌套 Lambda 表达式捕获变量的处理，对于内层的 Lambda 表达式，如果它的某个捕获变量已经被外层 Lambda 捕获，那么需要将它从外层中拿出来，再放到内层的指定位置上，而不能直接用这个变量原始的寄存器，原因是这个寄存器和捕获变量之间已经隔了一层或者多层函数了。