

# Decaf PA3 实验报告

计 76 张翔 2017011568

2019 年 12 月 1 日

## 1 实验简述

本阶段实验将 PA2 生成的带标注的 AST 转换成 TAC 中间代码，使得 Decaf 程序可以在 TAC Simulator 上运行。

## 2 主要工作

### 2.1 除零检测

这个特性的实现是非常简单的，只需要在 `visitBinary` 时增加判断除数是否为 0 的 TAC 代码，判断为 0 则报错并调用 `halt` 退出即可。

### 2.2 抽象类

之前实现时我将方法体改成了 `Optional<Body>` 从而支持抽象方法，因此本阶段访问方法体时利用 `ifPresent` 函数即可防止访问抽象方法的方法体而引发错误。

### 2.3 var 局部类型推导

这个新特性属于 PA2，类型推导成功后和普通变量没有区别，无需额外处理。

### 2.4 一等函数

为了实现一等函数这个新特性，所有函数调用都被统一成对具有 `Callable Type` 的表达式进行调用的操作，大部分操作都在 `VarSel` 这类节点中进行，`Call` 节点只处理调用本身。

为了实现方便，这里也定义一种统一的“函数对象”，其在堆区分配，内存布局如下

	4		...	
	func_ptr		extra fields	

所有可被调用的 `Expr` 都会有一个指向这种函数对象的指针。记这个指针为 `base_ptr`，调用方法为 `(*base_ptr)(base_ptr, ...args)`。这里为了解方便，借用了 C 函数指针的记号，`args` 为实际需要传递的参数。这样，原来的方法、Lambda 均可封装成函数对象，从而统一调用方式，方便处理。

### 2.4.1 封装类方法

类方法有 static 和 non-static 之分，它们的封装方式如下

```
static ReturnType ClassName::Func(...params) {...} // static method

// wrapper for static method
ReturnType staticWrapper(base_ptr, ...params) {
    return (&ClassName::Func)(...params);
}

ReturnType ClassName::Func(this, ...params) {...}
// non-static method, `this` is a hidden parameter

// wrapper for non-static method
ReturnType nonStaticWrapper(base_ptr, ...params) {
    auto this = *(base_ptr + 4);
    return (&ClassName::Func)(this, ...params);
}
```

由此可以很容易看出为何在堆区为函数对象分配空间时，对于 `staticWrapper` 只需要 4 bytes，对于 `nonStaticWrapper` 需要 8 bytes，因为后者在 `base + 4` 处还需要存储一个 `this` 指针。另外也可以看出调用函数对象时，第一个参数传送函数对象的基地址的必要性，因为函数对象需要利用该地址来访问 `extra fields` 中存储的内容。

这里有个比较 tricky 的实现。在 `Wrapper` 函数中，调用静态方法使用 `DirectCall` 配合 `Label` 即可，非静态方法通过 `this` 指针配合虚表也可调用，但函数对象基地址处存放的是指向 `Wrapper` 的指针，需要计算 `Wrapper` 函数的地址，但 Java 版本的 TAC 模拟器不支持直接通过 `Label` 获取函数地址。在不修改 TAC Simulator 的情况下，只能创建一个全局的虚表，给 `Wrapper` 赋予一个经过适当 mangle 的 `Label`（如 `className$methodName`），然后 `Wrapper` 的地址就可以表示为相对于全局虚表的偏移量。

### 2.4.2 封装数组长度 length 方法

虽然实验没有强制要求实现这个特性，但基于上面 `Wrapper` 的原理，实现数组长度函数对象也十分简单。考虑到数组长度是存在在数组基地址 -4 的位置，可以设计如下 `Wrapper`

```
// wrapper for array length
int arrayLengthWrapper(base_ptr, ...params) {
    auto arr_ptr = *(base_ptr + 4);
    return *(arr_ptr - 4);
}
```

创建这个函数对象时，类似类的 non-static 方法，`base_ptr + 4` 位置存放数组指针即可。

### 2.4.3 封装 Lambda 表达式

类似于类 non-static 方法的 `this` 指针, Lambda 表达式的函数对象创建时需要将被捕获变量的指针/值复制到 extra fields 部分, 这样 Wrapper 就可以通过 `base_ptr` 去访问被捕获变量, 如下

```
ReturnType LambdaWrapper(base_ptr, ...params) {
    auto captured_var_0 = *(base_ptr+4);
    auto captured_var_1 = *(base_ptr+8);
    ... // more captured vars
    return ([...captured_vars](args...) { ... })(...params);
}
```

这里仍然借助了 C++ 的语法来表示。此时 Lambda 表达式的变量捕获如同在 Wrapper 中发生一样。需要注意的是, 对于数组、类这种变量, 实际是捕获的是它们的指针。

被捕获变量列表已在 PA2 中实现, 具体是通过 LambdaScope 栈来实现的, 每个 LambdaScope 对象有三个列表: 当前捕获变量、当前定义变量、当前禁止变量列表。访问 Lambda 表达式节点时, 遇到局部变量定义则放入当前定义变量表中; 禁止变量表是防止诸如 `var f = fun() { f; }` 这种 Lambda 表达式访问赋值变量的情况, 该表从外层 LambdaScope 向内传递; 捕获变量表存放当前捕获变量, 当内层 LambdaScope 弹栈时, 会将内层捕获变量与外层定义变量做差集运算, 结果放入外层捕获变量表, 其含义为内层 Lambda 捕获非外层 Lambda 中定义的变量时, 也需要外层捕获。

通过捕获变量列表, 可以生成一个 offset 表 (从 4 开始), 用以表示各个捕获变量相对于 `base_ptr` 的偏移量。函数变量创建时需要分配  $4(1 + \text{\#Captured Vars})$  空间, 将被捕获变量的值/指针写入 `base_ptr+4` 之后的空间。

LambdaWrapper 是在 TacEmitter 访问 AST 中 Lambda 节点时构建的。访问时同样维护了一个 `Stack<LambdaScope>`, 当进入 Lambda 节点时将其作用域压栈, 退出时弹栈。此外, 对于 This 和 VarSel 节点中局部变量类型的访问需要进行特殊处理 (当 VarSel 为类成员变量/方法时无需特殊处理, 因为对 This 处理后已经可以得到类访问的正确地址)。当 `Stack<LambdaScope>` 非空时, 检查 This 或 VarSel 是否在捕获变量表中, 若为真, 将其访问替换为 `base_ptr+offset` 的形式, 从而确保能捕获到正确的值。

Lambda 表达式的 Wrapper 与 Lambda 表达式一一对应, 不像类方法/数组长度 Wrapper 一样可以被复用, 因此在 Label mangle 的时候需要留意, 这里使用类似于 `Lambda$row$col` 的命名方式保证 Label 的唯一性。

使用上述的 Wrapper 思想来实现对各种 Callable Expr 的调用是大有裨益的, 它很好地统一了不同类型 (Lambda、类方法或者数组方法) 的调用方式, 使得调用者只需要取出函数指针并传参即可, 而无需关注是否传递了 `this` 指针或者变量捕获等种种细节。当然这种函数对象的封装需要间接取址, 相较于静态调用, 一定程度上也增加了性能开销。