编译原理 PA3 实验报告

郑逢时 2016012177

2019年12月9日

1. 完成过程

本次PA总体没有上次难度、码量大，关键在于对 decaf 语言运行时内存分配的理解，因为函数子、lambda捕获变量等新特性都与运行时内存分配密切相关。

我一共历时一天阅读文档和基础框架代码，一天半写完代码通过测试。

1.0 合并框架

如题，将更新后的PA3框架手动合并入我已有的完整框架中，除了三个与PA3相关的文件外，所有其他有更新的文件我都手动复制粘贴了，目的是希望将来的PA可以少一些冲突。

1.1 跳过对抽象方法体的遍历

基础框架没有考虑函数体为空的情况，直接对函数体进行tacgen，但是新特性允许无函数体的抽象方法的存在，所以直接运行会遇到 `unreachable` 的错误。

简单修改，在tacgen一个函数之前先判断其是否抽象即可解决此问题。

1.2 除零动态错误检查

这部分是很容易实现的，个人感觉是对本次PA的一个热身，帮助熟悉tac语法和生成tac的套路。

实现的方法是直截了当的，只需在 expr 的`Bin`分支里增加对 Div, Mod 的特判即可。生成这样的tac：先判断除数是否为0，若为零则进入一个label，输出相应的runtime error信息后halt，否则进入正常计算的label。

其中有个小问题，就是我们的rust版本的tacvm会主动检查除数是否为常数0，若是会“自告奋勇”地报运行错误，这样会导致我们过不了测例。因此特判一下除数是常数0的情况，这种情况就不生成计算除法的语句了，而直接runtime error后halt。相当于静态期多做了一些事情。

1.3 扩展call语法

这部分扩展需要 VarSel 和 Call 两个节点同步配合来完成。首先为了方便，我按照建议，统一了直接call一个函数和先捕获其函数指针，再call的逻辑。（前者比如`f();`后者比如`var a = f; a();`）

都先统一构建函数子（Functor），即保存了函数入口指针和可能的对象指针的一个函数对象，再在call的时候统一通过Functor的函数入口指针进入入口函数（就是文档中描述的“新函数”，我称其为 entry function），入口函数利用Functor里可能存的对象指针，通过虚表寻址找到对应的虚函数再完成调用（或者直接调用静态函数）。

所以所完成的工作是按部就班的：

1. 重构call节点的处理，统一为一种case，即直接从functor中 load 到 entry function 的地址，传入functor和参数，调用entry；
2. 更新VarSel节点的处理，新增访问域是 FuncDef 时的处理：load entry function address、this pointer，构造 functor；
3. 在 tac program 的适当位置插入 entry function 的信息。为了避免每次调用函数都去先现场生成entry，造成tac爆炸，同时为了实现的方便，我选择在每个函数生成完tac后，**同步地**生成对应的entry function（尽管有的函数可能没调用到，我们也为其生成entry），编号紧随原函数之后（这样也为了寻址的方便）。如此一来，生成出的entry大概长这样：

```
FUNC<_A.a> {
    return
}

FUNC<_A.a._entry> {
    %1 = *(%0 + 4)
    %2 = *(%1 + 0)
    %2 = *(%2 + 8)
    parm %1
    call %2
    return
}

FUNC<_A.b> {
    return
}

FUNC<_A.b._entry> {
    %1 = *(%0 + 4)
    %2 = *(%1 + 0)
    %2 = *(%2 + 12)
    parm %1
    call %2
    return
}

FUNC<_A.c> {
    return
}

FUNC<_A.c._entry> {
    %1 = *(%0 + 4)
    %2 = *(%1 + 0)
    %2 = *(%2 + 16)
    parm %1
    call %2
    return
}
```

尽管我们的PA3不要求数组长度length的扩展，但我还是实现了，处理方法与上述类似，只需在全局生成一个且一次 length 方法即可。

```
FUNC<length> {
    %1 = *(%0 + 4)
    %1 = *(%1 - 4)
    return %1
}
```

1.4 Lambda 特性

放在下一节探讨。

2. Lambda 特性详细实现流程

2.1 捕获变量

按照建议，首先修改 PA2 的 type pass。

扩展 ast 中关于 lambda 节点的定义，允许保存捕获到的 VarDef，和是否捕获 this 指针（这个比较特殊，发现单独用一个bool来维护反而更简单）。用到的数据结构如下：

```
// captured variables | references
pub cap_list: RefCell<IndexSet<Ref<'a, VarDef<'a>>>>,
pub cap_this: Cell<bool>
```

在 typeck 中维护一个 lambda stack，并在每当扫描到 VarSel 的根部（no owner）的时候，向**比所捕获变量更为内层**的 lambda stack 中的 lambda 节点的捕获列表中添加该 VarDef。捕获this也是同理（注意 no owner 的 FuncDef 也可以捕获this）。

特别地，对 This 节点的动作也要修改，需要让 lambda stack 中所有的lambda都去捕获这个 this。

完成以上扩展后，我手写了一些测例测试了捕获变量功能的正确性。

2.2 Lambda 的 tacgen

回到 PA3，我们先实现对 ast 中所有 lambda 块的扫描，定义一个方法来构建 lambda 匿名函数的 tac program。

现在我们进入 lambda 的被调用者视角。

我们会认为传入的参数是 n+1 个，其中 n 是decaf程序里 lambda 表达式吃的参数个数，第零个参数是 lambda 函数对象的基址。类似于对 FuncDef 的处理，我们需要先注册参数的寄存器编号：

```
for (idx, p) in l.params.iter().enumerate() { // l: Lambda
    self.var_info.insert(Ref(p), VarInfo { off: 1 + idx as u32 });
    // offset 0 reserved for functor
}
```

然后令这个 lambda 实例入栈（为了后面扫描的过程中判断当前是否在lambda作用域，选择适当的语法动作的方便等），新建一个 `TacFunc::empty`，对lambda的 block 或者是 expr 做扫描，递归地生成 tac（具体递归干了些什么，在 lambda context 中应该注意什么，留给2.4节讨论）。

扫描完毕后会构建出一个 lambda 对应的一个匿名函数。这里会遇到匿名函数如何编号的问题，放在下一节探讨。

扫描完毕后记得从lambda栈中 pop 出这个lambda。

一个小细节是匿名函数的命名问题，最初我采用 `lambda@(xx,yy)` 这种直截了当的写法，但会出现奇怪的 tacvm Parser error，原因是 `@ ( , )` 这些符号不识别。用 `lambda_xx_yy` 就好了。

2.3 Lambda Functor 的构建

扩展 expr 节点，新增对 Lambda 类型节点的处理。

此处我们需要站在调用者的视角，由于每个 lambda 函数体通篇只会出现一次（或者被某个函数类型变量捕捉，在别处只是调用），我们可以就地调用 2.2 节中写好的方法，生成出其对应的匿名函数tac程序，然后需要构建一个指向这个程序的 functor。

按照文档中的约定，我们先 _Alloc 出适当空间来存 functor（具体大小可通过 2.1 节得到的捕获列表和捕获this与否来推算），然后通过 LoadFunc，先将匿名函数指针存入 functor 的基址，再将所捕获的局部变量（`self.var_info` 中查找）存入后续地址，最后存入捕获到 this 指针（`Reg(0)`，或者没捕获就不存）。

这样 lambda functor 就构造好了，将其寄存器返回。

但有种情况我们没cover到，就是当调用者本身就在一个 lambda context 中时，要再创建一个更内层的 lambda functor，此时就不一定能通过 `self.var_info` 找其捕获变量的寄存器了。因为内层lambda 捕获的变量有可能同样也是外层lambda的捕获变量，这样只能通过先从外层 lambda 的 functor 的对应偏移量的地址处 load 到内层要的变量，再 store 到内层 functor 的适当偏移量处。对 this 指针的处理也是同理。

构建完 lambda functor 后，勿忘将当前 lambda 函数的编号计数器递增。

2.4 Lambda Context 中的语法动作

为了复用代码，我将对lambda块的扫描完全委派给了 block, expr, stmt 这些 visitor。相应的代价是我们无时无刻要问当前扫描处的语境是否在一个 lambda 块中（称为 lambda context，否则称 normal context），如果是，那么在遍历到被 lambda 捕获的变量时要格外小心，需要从 functor 中load这个变量，而不从之前的寄存器里找。

为此，重构 VarSel！（重构起来真是毫不含糊hhh）

具体地，在遇到局部作用域里的 VarDef 时（no owner），问这个变量是否被捕捉到，若确实是，那么非常不幸，我们需要在 lambda 的捕获列表中找到它的index，然后从 functor 的对应位置（小心偏移量的计算） load 出它。由于我们构造 functor 时装填捕获变量的顺序与捕获列表中它们的顺序是一致的，所以这里不会张冠李戴。

对 this 的访问也要小心，现在同样要问当前是否在 lambda 中，若确实（可以 assert 一下这个 lambda 确实捕获过 this），那么同上处理，注意 this 是最后装入 functor 的。

如此实现后，应该就没有什么大问题了，最终能通过所有测试点和我自己编的一些刁钻测试。（比如 lambda 可以捕获 FuncDef 的一个函数指针等等）

3. 所遇困难及解决方案

大部分困难都在完成流程中有所交代了，这里只提几个比较有代表性的。

3.1 Lambda 匿名函数的编号问题

由于 FuncDef 的扫描过程中我们并不知道一个函数里面会遇到多少个 lambda，所以匿名函数的编号很难跟原有函数、入口函数的编号协调，一旦index无法确定，LoadFunc 也很难用对。对此我的解决方案是，让所有匿名函数的编号从所有有名函数（new函数，FuncDef的函数，以及其对应的入口函数）的后面开始编。

由于有名函数的个数是可以在 tacgen 之前就确定好，所以这个匿名函数起始编号我们可以事先推算出来，用一个全局计数器标记，以后每扫完一个 lambda 块，就让计数器加一。这样就可以既不扰乱有名函数的编号，又能正确地 LoadFunc 到匿名函数的地址了。

3.2 函数调用的 tac 不方便 Debug 的问题

我想很多人可能遇到类似问题，就是在扩展完 call 后，编译器"理论上"能构建函数子和调用函数子了，但 run test 会什么都不返回，一脸茫然。

所幸助教大大为我们提供了强大的 tacvm 报错功能，可以在运行时输出一个 xxx.info 的东西，里面汇报了运行时日志，如果出访存无效（不对齐之类的）、栈错误等，会汇报对应的错误信息。根据调用栈和寄存器的情况，容易确诊问题出在哪次函数调用上。比如我用这种方法查出了一个空返回类型 lambda 匿名函数忘记补上 return 这句话、和一处偏移量计算忘了乘4的bug。

此外，本人也通过 rust 为我们提供的 debug 函数来debug。最好只对单个测试样例debug，然后可以在tacgen源码里加一些 `assert!, dbg!, println!` 宏，看看程序是否按预期进入了正确的分支，或者每个状态下变量值是否正确等等。我用这种方法查出了某个错误捕获 this 的bug等问题。