PA3 Report

计71 钟闰鑫 2017010306

动态错误检查

定义新的运行时错误报错字符串

在处理`expr`的函数中，当处理到`Div`或者`Mod`节点时，添加新的代码使得其在执行操作前先检查除数是否为0，逻辑类似于其他运行时错误，因此直接copy过来略微修改即可

- 由于rust的tacvm的检查，处理方法为先将操作符右端表达式的值放到一个寄存器，再进行判断

抽象类

由于虚函数调用已在基础框架中实现，因此只需要对于抽象函数，跳过它的body处理即可

局部类型推导

PA2已经把类型推导出来，这里没有工作量

First-class Functions

扩展call

现在需要更改对于call的处理逻辑，处理call的时候，先对于call中的func这个expr进行处理，得到对应的函数对象(经过PA2的检查这里就不用再次检查了)，然后根据指导书上的建议进行函数对象的调用，即先load出函数指针，然后将本身作为参数传入，再传入其他参数，然后调用(按照虚函数调用的方式)，现在Call的逻辑就非常简单了

```
Call(c) => {
    //func obj
    let func_reg = self.expr(c.func.as_ref(), f);
    let func_ptr = self.reg();
    let args = c.arg.iter().map(|a| self.expr(a, f)).collect::<Vec<_>>();
    let hint = CallHint {
        arg_obj: c.arg.iter().any(|a| a.ty.get().is_class()),
        arg_arr: c.arg.iter().any(|a| a.ty.get().arr > 0),
    };
    let ret = if let TyKind::Func(ty) = c.func.as_ref().ty.get().kind {
        if ty[0] != Ty::void() {
            Some(self.reg())
        } else {
            None
        }
    } else {
        unreachable!("only func ty is callable")
    };
    f.push(Load {
        dst: func_ptr,
        base: [func_reg],
```

```
        off: 0,
        hint: MemHint::Obj,
    }).push( Param {
        src: [func_reg]
    });
    for a in args {
        f.push(Param { src: [a] });
    }
    f.push(Tac::Call {
        dst: ret,
        kind: CallKind::Virtual([Reg(func_ptr)], hint),
    });
    Reg(ret.unwrap_or(0))
```

方法名直接当做函数使用

由于Call对于所有函数对象的处理已经被统一，考虑到静态/非静态函数有着不同调用方式，因此当Varsel解析到一个函数对象时，需要重新生成一个函数来辅助调用原来的函数，并将新生成的函数的对象返回回去

由于在PA2中更改了Varsel的结构，现在当varsel处理发现是一个FuncDef的时候，将会首先生成一个新的TacFunc，在这个TacFunc里添加对于对应的FuncDef函数调用，最后将TacFunc的函数对象返回回去

为了能够添加新的TacFunc，更改了TacGen结构

```
struct TacGen<'a> {
    func: std::vec::Vec<TacFunc<'a>>,
    alloc: Option<&'a Arena<TacNode<'a>>>,
    ...
}
```

在Tac生成开始时将外层的alloc给到TacGen，这样在处理过程中就能调用其来分配空间

在处理过程中生成的新函数将会被放在func里，在所有都处理完毕后，这个func将会被append到原来最初用来存函数定义的结构中

```
fn program(mut self, ...) {
    self.alloc = Some(alloc);
    //...处理
    tp.func.append(&mut self.func);
    self.alloc = None;
}
```

在Varsel处理FuncDef时，按照指导书上的建议做即可，有如下几个遇到的坑需要注意

- 在创建新函数并在新函数里插入TacNode的时候，其内部的寄存器分配需要注意，得从参数个数(以及this)之后开始分配，以避免函数内部对参数寄存器进行了覆盖

  这一点按照原来框架调用的范式写就没问题(主要是保存当前寄存器编号和label编号，然后重新开始，新函数完成后恢复寄存器编号等)

- 使用virtual调用和使用static调用是不同的，前者给出寄存器，调用寄存器指向地址对应函数，后者直接调用寄存器

Lambda表达式

Lambda表达式主要工作量在于expr处理lambda表达式的部分和debug

捕获变量

捕获部分需要更改PA2中的内容，实际上PA2已经完成了绝大多数工作，之前更改了PA2中lookup_before的逻辑，能够返回找到的变量是否位于当前lambda外，用这个标志结合在PA2 type_pass中的判断逻辑即可判断是是否应该捕获变量

更改了LambdaDef这个AST节点，现在可以记录被捕获的变量

```
pub struct LambdaDef<'a> {
    ...
    pub captured_var: RefCell<std::vec::Vec<Ref<'a, VarDef<'a>>>>,
}
```

- 这里在处理捕获的时候，为了实现上的方便，每次都会将this进行捕获(而不是考虑当前函数类型)，并规定位置一定为`ptr + 4`，这样虽然对静态函数来说，它会往`*(ptr + 4)`中存一个无效值，但显然这个无效值也不会在lambda内部用到，因此没有额外的影响(除了多分配了4字节内存)

实际捕获放在type_pass中对于Varsel节点的处理，当没有owner的时候考虑是否捕获

```
else if let Some(lam) = self.lambda_stack.last() { //如果当前在一个Lambda中
    if out_of_lambda {
        lam.captured_var.borrow_mut().push(Ref(var));
    }
}//out_of_lambda来源于之前的lookup_before
```

对于嵌套lambda，捕获变量需要传递，并且需要去重，这里同样可以用到`lookup_before`来判断

具体逻辑放在type_pass中对于expr的处理中lambda表达式处理末尾，当结束一个lambda表达式处理时，首先在lambda栈中弹出当前lambda，然后判断是否仍然位于一个lambda中，若是则开始进行捕获变量的传递以及去重

- 这里只需要考虑传递到上一层而不是之前所有lambda(因此其实也没有必要用lambda栈，每次临时手动保存一下上次的lambda存在情况就好)，因为递归调用最终内层lambda捕获变量会逐层向上传递

```
self.lambda_stack.pop();
if let Some(last_lam) = self.lambda_stack.last() {
    let mut new_elem: std::vec::Vec<Ref<'a, VarDef<'a>>> =
lam.captured_var.borrow().iter().filter_map(|&v| {
        let (_, out_of_lambda) = self.scopes.lookup_before(v.name, e.loc);
        if !out_of_lambda {
            None
        } else {
            Some(v)
        }
    }).collect();
    last_lam.captured_var.borrow_mut().append(&mut new_elem);
}
```

处理lambda表达式

捕获完成后，需要增加在TacGen中expr对于LambdaDef的处理

在TacGen中添加cur_lambda指示当前进入的lambda表达式，并在每次进入lambda表达式处理的时候进行维护

```
struct TacGen<'a> {
    ...
    cur_lambda: Option<&'a LambdaDef<'a>>,
}
```

lambda表达式的具体处理与之前函数对象的处理类似，首先先生成一个新的函数，然后在新的函数中处理lambda表达式的body或者expr，再构造一个新的函数，内部实现对lambda函数的调用，最后返回这个新的函数的函数对象

有以下几个容易出错的地方

- 在处理lambda函数前要保存现场，即保存好寄存器编号情况，label编号情况等，在处理完lambda之后将其恢复，保存和恢复的位置很关键
- 额外需要保存的是原框架中定义的var_info，这个结构是为了存储对应var的寄存器编号，由于lambda表达式也有参数，因此需要首先保存原来的var_info，然后将lambda表达式对应的参数在处理前插入到var_info中(需要注意恒捕获this导致的offset)，再处理完成后对var_info进行恢复
- 对于var_info，由于捕获变量也需要通过var_info进行查找，因此在实际进入lambda表达式处理前，也要将被捕获的变量插入到var_info之中去
- 处理lambda表达式的block时候，类似原来框架中FuncDef的处理，需要最后判断一下是否返回值为void，如果block的返回值不为void，则最后插入一句`Tac::Ret`；而对于lambda表达式的expr处理，则直接先处理expr，然后插入一句`Tac::Ret`

至此，大部分工作都已完成，剩下的就是debug

其中一个比较坑的bug就是

- 原始框架中由于类对象所在位置一般为`Reg(0)`处(对于非static对象)，因此存在很多需要类对象的时候，默认返回`Reg(0)`，但是引入lambda表达式之后，情况有所不同，因为在lambda表达式中类对象的是在`*(Reg(0) + 4)`处，因此所有涉及到这种默认类对象返回`Reg(0)`的地方都得重新考虑，加上判断cur_lambda存在与否，有必要则先从`Reg(0) + 4`中load出来然后再返回