Stage-4 实验报告

沙之洲 2020012408

Step 9

实验内容

step 9 的实际实验内容非常多且琐碎。这里只展示一些重要的部分，具体的细节可以看代码实现。

首先在 parse 部分，我们要支持函数的声明和定义，此二者的区别在于是否有函数体。这里我在 Function 这个 Node 节点上进行了区分

```
86
87  class Function(Node):
88      """
89      AST node that represents a function.
90      """
91
92      def __init__(
93          self,
94          ret_t: TypeLiteral,
95          ident: Identifier,
96          parameter_list : list[Parameter],
97          body: Optional[Block] = None,
98      ) -> None:
99          super().__init__("function")
100         self.ret_t = ret_t
101         self.ident = ident
102         self.parameter_list = parameter_list
103         self.body = body or NULL
104
```

接下来我们需要在 namer 和 tacgen 部分不仅仅是对 main 函数进行访问，而是要对于每一个 function 进行访问。这里我们以 namer 为例进行展示

```
        # get all functions
        # if use this , will ignore define conflict error since the return value is
        function_list = program.funct_list
        for item in function_list:
            # deal with funct and global var separately
            if isinstance(item, Function):
                item.accept(self, ctx)
                continue

            # deal with global var
            if isinstance(item, Declaration):
                var_name = item.ident.value
                var_type = item.var_t.type
                var_init_expr = item.init_expr

                # check if it has been declared
                if ctx.globalscope.containsKey(var_name):
                    raise DecafDeclConflictError(var_name)

                # declare the var
                var_symbol = VarSymbol(var_name, var_type, True)
                ctx.globalscope.declare(var_symbol)
                item.setattr("symbol", var_symbol)

                if not var_init_expr:
                    var_init_expr.accept(self, ctx)
                continue

            # we will not deal with things other than function and global var
```

对于后端部分，我们需要做的主要是对于函数调用中 caller save 和 callee save 的处理

对于 caller save，这一部分的实现是在 bruteallocate 文件中，在 save caller save 之后，进行了函数传参

```
        # store to stack
        for reg in used_caller_saved:
            subEmitter.emitStoreToStack(reg)
            caller_save_dict[reg.name] = subEmitter.offsets[reg.temp.index]

        # have to do it by human
        argument_len = len(self.call_argument_list)

        if argument_len <= 8:
            for i in range(argument_len):
                # self.call_argument_list = list[Reg]
                reg_name = self.call_argument_list[i].name
                if reg_name in caller_save_dict:
                    # if this register is stored on stack
                    subEmitter.emitNative(Riscv.NativeLoadWord(Riscv.ArgRegs[i], Riscv.SP, caller_save_dict[reg_name]))
                else:
                    # just do move is ok
                    subEmitter.emitNative(Riscv.NativeMove(Riscv.ArgRegs[i], self.call_argument_list[i]))

        else:
```

对于 callee save，是在 riscvasmemitter 中实现的。

```
282         self.printer.printInstr(Riscv.SPAdd(-self.nextLocalOffset))
283 
284         # in step9, you need to think about how to store RA here
285         # you can get some ideas from how to save CalleeSaved regs
286         for i in range(len(Riscv.CalleeSaved)):
287             if Riscv.CalleeSaved[i].isUsed():
288                 self.printer.printInstr(
289                     Riscv.NativeStoreWord(Riscv.CalleeSaved[i], Riscv.SP, 4 * i)
290                 )
291 
292         # also store ra here
293         self.printer.printInstr(
294             Riscv.NativeStoreWord(Riscv.RA, Riscv.SP, 4 * len(Riscv.CalleeSaved))
295         )
296 
297         self.printer.printInstr(
298             Riscv.NativeStoreWord(Riscv.FP, Riscv.SP, 4 * len(Riscv.CalleeSaved) + 4)
299         )
300 
```

由于我的函数读取参数是基于 FP 的，所以在 FP 也会在被调用函数执行的时候改变，因此也要作为 callee save reg 存在栈上。

思考题

1 不同参数的求值顺序会导致不同的返回结果

```
int f(int a, int b){
    return a;
}

int main(){
    int c = 0;
    return f(c += 1, c += 1)
}
```

如果是从左到右求值，返回结果会是 1 ，如果是从右到左求值，返回结果是 2

2 为何 RISC-V 标准调用约定中要引入 callee-saved 和 caller-saved 两类寄存器，而不是要求所有寄存器完全由 caller/callee 中的一方保存？为何保存返回地址的 ra 寄存器是 caller-saved 寄存器?

由于将寄存器的值保存到栈上实际上涉及到了 memory 的操作，耗时较长。所以要尽可能在函数调用的时候减少这种向栈上存储的行为。

直观上理解，caller save 的意思就是这些寄存器不建议 caller 使用，因为一旦使用就需要在调用函数的时候将这些寄存器保存到栈上。同理对于 callee save 寄存器的意思也是不建议 callee 使用。通过让 caller 使用 callee save 的寄存器，callee 使用 caller save 的寄存器，能够在最大程度上减少函数调用的时候向栈上保存寄存器值的操作，从而提升程序的效率。

对于 ra 寄存器，实际上 ra 寄存器永远是指向当前正在执行的指令的地址。如果 ra 是 callee save 的，那么在函数调用的时候，会在函数跳转之后才保存 ra，但是这个时候 caller 正在执行的地址已经被覆盖了，这样会导致函数结束的 ret 不能返回到正确的位置。因此，ra 寄存器必然是 caller save 的。

Step 10

实验内容

对于本部分，我们首先需要在 parse 实现对于 global var 的解析

```
    p[0] = Program(p[1])

# step 10 codes here
def p_function_list_global_var(p):
    """
    function_list : declaration Semi function_list
    """
    p[0] = [p[1]] + p[3]


```

同理我们需要在 namer 和 tac gen 的时候首先处理 global var，这里和 function 的处理类似，不再赘述。

这里需要注意的是，当我们访问 global var 的时候，需要先找到 global var 的地址，再用 lw 来访问 global var 的值。这一步我是在 tac gen 中就实现了

```
            mv.visitBranch(mv.getContinueLabel())

    def visitIdentifier(self, ident: Identifier, mv: FuncVisitor) -> None:
        """
        1. Set the 'val' attribute of ident as the temp variable of the 's
        """
        symbol : VarSymbol= ident.getattr("symbol")
        if symbol is None:
            breakpoint()
        if symbol.isGlobal:
            var_name = symbol.name
            var_addr = mv.visitLoadGlobalAddr(var_name)
            var_temp = mv.visitLoadFromMem(var_addr, 0)
            ident.setattr("val", var_temp)
        else:
            temp = symbol.temp
            # Identifier 的挂载和 Declaration 的挂载一致
            ident.setattr("val", temp)

```

接下来在后端只需要根据对应的 tac 指令去翻译即可。

思考题

我们规定 inter = a - pc 。这里 pc 指的是当前指令的地址。则对于指令

```
la v0, a
```

会被翻译成

```
auipc t0, inter[31:12]
addi t0, t0, inter[11:0]
```

也有可能翻译成

```
auipc t0, inter[31:12]
ori t0, t0, inter[11:0]
```