MiniDecaf-Stage4实验报告

计06 赵翊哲 2020010998

一、实验目的

- 支持函数的声明、定义、调用
- 支持全局变量的声明、定义、访问

二、实验内容

2.1 Step 9

- 修改内容：
  - 增加了函数的声明、定义、调用
  - 支持函数的参数列表
- 实现：
  - 本次修改的部分较多，列举了部分完成实验时所使用的 `TODO`，后续只简要介绍修改的内容，列举部分核心函数作为示例。

  - 前端：
    - `ply_parser.py`：修改了文法识别，允许 program 下有多个函数，每个函数可有0个或多个参数
    - `tree.py`：修改了 Function 节点定义，增加了参数列表，增加了参数列表和调用函数节点
    - `namer.py`：类型检查时对program下的每个函数依次进行类型检查，值得注意的是函数体和参数需要在同一个作用域下进行变量重名检查。同时对于函数的参数和调用时的实参列表都需要一一进行检查
      - 核心代码：

```
```

```
def visitFunction(self, func: Function, ctx: ScopeStack) -> None:
        # TODO: Step9-7 新建符号表
        """首先需要新建一个函数符号 FuncSymbol，Scope(ScopeKind.LOCAL)函数作用域，
        这个作用域既包括函数体内部的变量，也包括函数的所有参数。
        因此，在访问函数体之前，需要先扫描参数列表，
        新增 visitParameter 函数为所有参数建立符号，并存入符号表"""
        symbol = ctx.findConflict(func.ident.value)
        if symbol == None:
            if func.body is NULL:
                symbol = FuncSymbol(func.ident.value, func.ret_t.type, ctx.globalscope, False)
            else:
                symbol = FuncSymbol(func.ident.value, func.ret_t.type, ctx.globalscope, True)
            # 函数名加入全局符号表
            ctx.globalscope.declare(symbol)
            func.setattr("symbol", symbol)
            # localScope = Scope(ScopeKind.LOCAL, True)
            # ctx.open(localScope)
            if not func.params is NULL:
                # 添加参数类型
                for param in func.params:
                    symbol.addParaType(param.var_t)
            if not func.body is NULL:
                func.body.add_params(func.params)
                # 定义函数
                func.body.accept(self, ctx)
            # ctx.close()
        else:
            # 重复声明，要求类型一致
            if func.body is NULL and func.ret_t == symbol.type:
                pass
            elif not func.body is NULL: # 定义函数
                if symbol.definition:   # 已经定义
                    raise DecafGlobalVarDefinedTwiceError(func.ident.value)
                else:
                    # 需要让 param 和 body 在同一个作用域
                    # localScope = Scope(ScopeKind.LOCAL) 会导致函数内的 block 不能新开作用域
                    # ctx.open(localScope)
                    if not func.params is NULL:
                        # 添加参数类型
                        for param in func.params:
                            symbol.addParaType(param.var_t)
                            # 将参数添加到局部作用域
                        # func.params.accept(self, ctx)
                    # 定义函数
                    else:   # 无参数
```

```
                    func.body.add_params(func.params)
                    func.body.accept(self, ctx)
                symbol.definition = True
```

- 中端：生成 tac 码
  - `tacinstr.py`：新增了 `PARAM`，`GETPARAM`，`CALL` 等中间指令，用于指定参数，获取参数到指定变量中，调用函数
  - `tacgen.py`：为每个函数体进行 tac 码生成，并且支持函数调用

    核心代码：

```
def transform(self, program: Program) -> TACProg:
        # TODO: Step9-11 为每个函数体进行 TAC 生成
        functions = program.functions()
        globals = program.globals()
        func_idents = [func for func in functions.keys()]
        pw = ProgramWriter(func_idents, globals)
        self.pw = pw
        nextTempId = 0
        for func_ident in func_idents:
            if func_ident == 'main':
                mv = pw.visitMainFunc()
                mv.nextTempId = nextTempId
                functions[func_ident].body.accept(self, mv)
                mv.visitEnd()
            elif not functions[func_ident].body is NULL:
                # 只处理定义函数
                paramLen = functions[func_ident].params.__len__
                fv = pw.visitFunc(func_ident, paramLen)
                fv.nextTempId = nextTempId
                index = 0
                # if len(functions[func_ident].params) < 9:
                for param in functions[func_ident].params:
                    param.accept(self, fv)
                    fv.visitGetParam(param.getattr("symbol").temp, index)
                    index += 1
                functions[func_ident].body.accept(self, fv)
                fv.visitEnd()
                nextTempId = fv.nextTempId
            # 不确定是否要额外处理只声明的函数
        return pw.visitEnd()
def visitCall(self, call: Call, mv: FuncVisitor) -> None:
        """依次访问所有调用参数，并将它们的返回值(getattr("val"))依次执行 PARAM 指令，
        然后执行 CALL 指令，并新建临时变量为函数设置返回值。"""
        index = 0
        # if len(call.arguments.children) < 9:
        for arg in call.arguments.children:
```

```
            arg.accept(self, mv)
        for arg in call.arguments.children:
            mv.visitParam(arg.getattr("val"), index)
            index += 1
        call.setattr("val",
mv.visitCall(mv.ctx.getFuncLabel(call.ident.value)))
```

- 后端：step9核心的难点和修改范围在后端，前端和中端的代码修改和之前的step是一脉相承的，而这次的后端需要理解寄存器的分配机制，并且在此基础上为传参时指定约定好的寄存器，并且进行caller&callee寄存器的保存，同时需要处理超过八个参数时压栈传参的情况，即需要事先判断好栈帧的布局，确定将参数保存到栈上所需要的偏移量。
  - `riscvasmemitter.py`：增加了对`PARAM`、`GETPARAM`、`CALL`等指令的select instr支持，并且修改了`RiscvSubroutineEmitter`，在函数开始时保存RA，在结束返回前从栈中获取RA。
  - `bruteregalloc.py`：主要修改的代码在为每条指令分配寄存器的函数中，需要针对参数相关的指令做额外的处理，包括选择特定的寄存器以及判断参数个数是否需要压栈传参等。

    核心代码：

```
def allocForLoc(self, loc: Loc, subEmitter: SubroutineEmitter):
        instr = loc.instr
        srcRegs: list[Reg] = []
        dstRegs: list[Reg] = []
        # TODO: step9-15 为函数传参进行特殊指令生成

        if type(instr) == Riscv.Param:
            if instr.index > 7: # 压栈
                temp = instr.srcs[0]
                reg = self.allocRegFor(temp, True, loc.liveIn, subEmitter)
                subEmitter.emitNative(Riscv.NativeStoreWord(reg, Riscv.SP,
4 * (instr.index - 8)))
            else:
                temp = instr.srcs[0]
                reg = Riscv.ArgRegs[instr.index]
                if reg.occupied:
                    subEmitter.emitStoreToStack(reg)
                    subEmitter.emitComment("  spill {}
({})".format(str(reg), str(reg.temp)))
                    self.unbind(reg.temp)
                dstRegs = [reg]
                srcRegs.append(self.allocRegFor(temp, True, loc.liveIn,
subEmitter))
                mv_instr = Riscv.Move(reg, self.bindings[temp.index])
                subEmitter.emitNative(mv_instr.toNative(dstRegs, srcRegs))
        elif type(instr) == Riscv.GetParam:
            if instr.index > 7: # 出栈
                temp = instr.dsts[0]
                reg = self.allocRegFor(temp, False, loc.liveIn, subEmitter)
```

```
                subEmitter.emitNative(Riscv.NativeLoadWord(reg, Riscv.SP,
subEmitter.nextLocalOffset + 4 * (instr.index - 8)))
            else:
                temp = instr.dsts[0]
                if isinstance(temp, Reg):
                    dstRegs.append(temp)
                else:
                    dstRegs.append(self.allocRegFor(temp, False,
loc.liveIn, subEmitter))
                reg = Riscv.ArgRegs[instr.index]
                srcRegs = [reg]
                mv_instr = Riscv.Move(dstRegs[0], reg)
                subEmitter.emitNative(mv_instr.toNative(dstRegs, srcRegs))
        elif type(instr) == Riscv.Call:
            # 保存 caller-saved 寄存器
            temp_list = {}
            for i in range(len(Riscv.CallerSaved)):
                if Riscv.CallerSaved[i].isUsed():
                    temp_list[Riscv.CallerSaved[i]] =
Riscv.CallerSaved[i].temp
                    subEmitter.emitStoreToStack(Riscv.CallerSaved[i])
                    self.unbind(Riscv.CallerSaved[i].temp)
            temp = instr.dsts[0]
            if isinstance(temp, Reg):
                dstRegs.append(temp)
            else:
                dstRegs.append(self.allocRegFor(temp, False, loc.liveIn,
subEmitter))
            srcRegs = [Riscv.A0]
            mv_instr = Riscv.Move(dstRegs[0], srcRegs[0])
            subEmitter.emitNative(instr.toNative(dstRegs, srcRegs))
            subEmitter.emitNative(mv_instr.toNative(dstRegs, srcRegs))
            subEmitter.emitStoreToStack(dstRegs[0])
            # 恢复 caller-saved 寄存器
            for reg in temp_list.keys():
                subEmitter.emitLoadFromStack(reg, temp_list[reg])
                self.bind(temp_list[reg], reg)
        else:
            for i in range(len(instr.srcs)):
                temp = instr.srcs[i]
                if isinstance(temp, Reg):
                    srcRegs.append(temp)
                else:
                    srcRegs.append(self.allocRegFor(temp, True, loc.liveIn,
subEmitter))

            for i in range(len(instr.dsts)):
                temp = instr.dsts[i]
                if isinstance(temp, Reg):
```

```
                    dstRegs.append(temp)
                else:
                    dstRegs.append(self.allocRegFor(temp, False,
loc.liveIn, subEmitter))
            subEmitter.emitNative(instr.toNative(dstRegs, srcRegs))
```

2.2 Step 10

- 修改内容：
  - 支持全局变量的声明、定义、访问
- 实现：
  - 本次修改的部分较多，列举了部分完成实验时所使用的 `TODO`，后续只简要介绍修改的内容，列举部分核心函数作为示例。

  - 前端：
    - `tree.py`：修改了 program 节点，运行 declaration 作为其孩子节点
    - `ply_parser.py`：修改了 program 的文法
    - `namer.py`：修改了 visitDeclaration 函数，在进行类型检查的时候需要判断变量是否为全局变量，如果是全局变量的话需要检查是否使用了常量初始化，否则需要报错

      核心代码：

```
def visitDeclaration(self, decl: Declaration, ctx: ScopeStack) -> None:
        """
        1. Use ctx.findConflict to find if a variable with the same name
has been declared.
        2. If not, build a new VarSymbol, and put it into the current scope
using ctx.declare.
        3. Set the 'symbol' attribute of decl.
        4. If there is an initial value, visit it.
```

```
        """
        # TODO: Step5-3 处理声明
        # ScopeStack.findConflict : 报错 or 定义 VarSymbol 对象
        symbol = ctx.findConflict(decl.ident.value)
        if symbol != None:
            raise DecafGlobalVarDefinedTwiceError(decl.ident.value)
        # ScopeStack.declare : 加入符号表
        else:
            # TODO: Step10-3 判断全局变量作用域，并且全局变量只支持常量初始化，需要
进行语义检查
            isGlobal = ctx.isGlobalScope()
            symbol = VarSymbol(decl.ident.value, decl.var_t.type, isGlobal)
            ctx.declare(symbol)
            # Declaration.setattr : VarSymbol 存入 AST
            decl.setattr("symbol", symbol)
            # 初值表达式
            if decl.init_expr != NULL:
                if type(decl.init_expr) == Call and isGlobal:
                    raise DecafGlobalVarBadInitValueError(decl.ident.value)
                decl.init_expr.accept(self, ctx)
```

- 中端：
  - `tacinstr.py`：新增了 `Load`，`LoadSymbol`，`Store` 等中间指令，用于在访问全局变量的时候获取变量的绝对地址，并且根据地址获得全局变量的值或储存值到内存中。
  - `tacgen.py`：在访问变量的时候判断是否是全局变量，是的话需要额外进行标注（以便后续的 Store 指令使用），并且生成对应的 LoadSymbol & Load 的三地址码

    核心代码：

```
def visitIdentifier(self, ident: Identifier, mv: FuncVisitor) -> None:
        """
        1. Set the 'val' attribute of ident as the temp variable of the
'symbol' attribute of ident.
        """
        # TODO: step10-6 加载全局变量方式为：首先加载全局变量符号的地址，然后根据地址
来加载数据。因此，需要定义两个中间代码指令，完成全局变量值的加载
        symbol = ident.getattr("symbol")
        if symbol.isGlobal:
            for globalVar in self.pw.globals:
                if globalVar == ident.value:    # 变量名匹配
                    src = mv.visitLoadSymbol(ident.value)
                    dst = mv.visitLoadGlobalVar(src, 0)
                    symbol.temp = dst
                    symbol.isGlobal = True
                    ident.setattr("addr", src)
                    ident.setattr("global", True)   # 为之后给全局变量赋值提供
信息
                    # print("test: ", ident.getattr("global"))
```

```
                    ident.setattr("addr", src)
                    ident.setattr("val", symbol.temp)
                    break
            else:
                # TODO: step5-4 设置该表达式的返回值 val 为该变量对应符号里的 symbol
                ident.setattr("global", False)
                ident.setattr("val", ident.getattr("symbol").temp)
```

除此之外，需要在赋值语句中判断表达式左侧是否为全局变量，若是则需要根据预先设置的地址通过 STORE 指令进行赋值。

核心代码：

```
def visitAssignment(self, expr: Assignment, mv: FuncVisitor) -> None:
        """
        1. Visit the right hand side of expr, and get the temp variable of
left hand side.
        2. Use mv.visitAssignment to emit an assignment instruction.
        3. Set the 'val' attribute of expr as the value of assignment
instruction.
        """
        # TODO: step5-6 设置赋值语句的中间代码（参考 visitBinary 实现）
        # TODO: step10-9 将对全局变量的赋值特殊处理
        expr.lhs.accept(self, mv)
        expr.rhs.accept(self, mv)
        temp = expr.lhs.getattr("symbol").temp
        # 使用 temp 添加 TAC 指令，而非 var
        isGlobal = expr.lhs.getattr("global")
        # print(expr.lhs)
        if isGlobal:
            addr = expr.lhs.getattr("addr")
            mv.visitStore(expr.rhs.getattr("val"), addr, 0)
        else:
            mv.visitAssignment(temp, expr.rhs.getattr("val"))
        # 设置表达式 val
        expr.setattr("val", temp)
```

- 后端：后端的处理较为简单，只需根据全局变量是否设置初值将其分类为bss段和data段，并且为新增的TAC代码选择对应的 Riscv 指令
- 核心代码：

```
def __init__(
        self,
        allocatableRegs: list[Reg],
        callerSaveRegs: list[Reg],
        globals: dict[str, Declaration]
    ) -> None:
        super().__init__(allocatableRegs, callerSaveRegs)
```

```
        self.globals = globals
        self.bss = {}
        self.data = {}


        # the start of the asm code
        # int step10, you need to add the declaration of global var here
        # TODO: Step10-7 处理全局变量的声明: .data & .bss
        for key, value in self.globals.items():
            if value.init_expr is NULL: # 无初值
                self.bss[key] = value
            else:   # 有初值
                self.data[key] = value
        if self.bss:    # 生成 bss 段
            self.printer.println(".bss")
            for key, value in self.bss.items():
                self.printer.println(".globl " + key)
                self.printer.println(key + ":")
                self.printer.println("    " + ".space 4")
        if self.data:    # 生成 bss 段
            self.printer.println(".data")
            for key, value in self.data.items():
                self.printer.println(".globl " + key)
                self.printer.println(key + ":")
                self.printer.println("    " + ".word " +
str(value.init_expr.value))

        self.printer.println(".text")
        self.printer.println(".global main")
        self.printer.println("")

# TODO: Step10-8 为 Load 和 LoadSymbol 选择对应的 RISCV指令
def visitLoad(self, instr: Load) -> None:
    """从地址中获取全局变量"""
    self.seq.append(Riscv.LoadGlobalVar(instr.dst, instr.src, instr.offset))

def visitStore(self, instr: Store) -> None:
    """向地址为全局变量赋值"""
    self.seq.append(Riscv.Store(instr.data, instr.addr, instr.offset))

def visitLoadSymbol(self, instr: LoadSymbol) -> None:
    """获取全局变量的地址"""
    self.seq.append(Riscv.LoadSymbol(instr.dst, instr.symbol))
```

三、思考题

3.1 Step9 思考题

Q1：MiniDecaf 的函数调用时参数求值的顺序是未定义行为。试写出一段 MiniDecaf 代码，使得不同的参数求值顺序会导致不同的返回结果。

> 在函数调用时如果进行参数求值，本阶段未定义顺序，因此从左向右和从右向左会给出不同的计算结果，在下方的代码中，根据不同的顺序，可能会得到 1 或 2 两种结果。

A：按照我的实现方式，会得到如下的汇编代码，最终的计算结果是 2。

```
int sum(int a, int b) {
  return a + b;
}
int main() {
  int x = 0;
  return sum(x = x + 1, x);
}
```

生成的汇编代码如下：

```
sum:
    # start of prologue
    addi sp, sp, -48
    sw ra, 44(sp)
    # end of prologue

    # start of body
    mv t0, a0
    mv t1, a1
    add t2, t0, t1
    mv a0, t2
    j sum_exit
    # end of body

sum_exit:
    # start of epilogue
    lw ra, 44(sp)
    addi sp, sp, 48
    # end of epilogue

    ret

main:
    # start of prologue
    addi sp, sp, -64
    sw ra, 44(sp)
    # end of prologue
```

```
    # start of body
    li t0, 0
    mv t1, t0
    li t0, 1
    add t2, t1, t0
    mv t1, t2
    mv a0, t1
    mv a1, t1
    sw t0, 48(sp)
    sw t1, 52(sp)
    sw t2, 56(sp)
    call sum
    mv t0, a0
    sw t0, 60(sp)
    lw t0, 60(sp)
    mv a0, t0
    j main_exit
    # end of body

main_exit:
    # start of epilogue
    lw ra, 44(sp)
    addi sp, sp, 64
    # end of epilogue

    ret
```

Q2：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？

A：不管是 caller-saved 还是 callee-saved 实际上都是希望 调用者/被调用者不要使用，因为（被调用者）调用者使用（callee-saved）caller-saved 寄存器意味着要进行访存，把寄存器原先的值保存到栈上供之后使用，这中内存访问的开销比访问寄存器高。因此如果寄存器全部为 caller/callee-saved，意味着被调用者/调用者需要承担100%的压力，因此设置一部分为 caller，一部分为callee，可以让调用者和被调用者都能有一些可以直接使用的寄存器，更加方便。

因为在函数调用的过程之中，ra会发生改变，例如 main 调用了 func1，在进入 func1的时候，ra中所保存的已经是 func1 的返回地址，因此在调用之前就需要将 ra 原先的值压栈保存，即 caller-saved。

3.2 Step10 思考题

Q1：写出 `la v0, a` 这一 RiscV 伪指令可能会被转换成哪些 RiscV 指令的组合（说出两种可能即可）。

A：la 在处理与**位置无关（PIC）**的代码时会被扩展为基于 GOT（Global Offset Table） 的处理，一种 Riscv指令的组合如下（其中GOT[symbol] 表示symbol对应的 GOT表项，存放了symbol的绝对地址）：

```
auipc v0, offset[31:12] # offset 为 GOT[a] 和 pc 之间的偏移量
lw v0, (offset[31:12])v0
```

另一种组合方式

```
auipc v0, offset[31:12] # offset 为 GOT[a] 和 pc 之间的偏移量
addi v0, offset[11:0]
lw v0, (0)v0
```

在**non-PIC**的情况下，伪指令la相当于伪指令lla，这种情况下只需通过 symbol 和 pc 之间的 offset 来获取地址：

```
auipc v0, offset[31:12] # offset 为 symbol a 和 pc 之间的偏移量
addi v0, v0, offset[11:0]
```

四、参考资料

实现过程中主要参考了实验文档和实验思路QA墙

思考题参考了caller-saved & callee-saved、la在non-PIC下的情况、深入理解计算机系统（pdf）