刘雅迪

计26

学号：2021010521

step9: 函数

思考题

1. 你更倾向采纳哪一种中间表示中的函数调用指令的设计（一整条函数调用 vs 传参和调用分离）？写一些你认为两种设计方案各自的优劣之处。

   具体而言，某个“一整条函数调用”的中间表示大致如下：

   ```
   _T3 = CALL foo(_T2, _T1, _T0)
   ```

   对应的“传参和调用分离”的中间表示类似于：

   ```
   PARAM _T2
   PARAM _T1
   PARAM _T0
   _T3 = CALL foo
   ```

   答：我更倾向于采纳传参与调用分离的设计，因为实验的语法是这样设计的。

   - 一整条函数调用：
     - 优点：表达简洁，比较易读。
     - 缺点：实现比较复杂，难以支持变长参数的调用。
   - 传参和调用分离：
     - 优点：实现灵活，便于调试和分析，与实验文档的码风一致。
     - 缺点：代码比较冗长，可读性较差。
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？

   答：①如果寄存器全由caller保存，因为caller必须保存每个可能在callee中被修改的寄存器，所以即时caller只关心几个寄存器，仍然会引入额外的保存和恢复指令，增大开销；如果寄存器全由callee保存，这样函数每次调用都需要保存所有的寄存器，即使大部分寄存器都没有被使用，增大了开销。

   ②因为调用函数时ra寄存器中的返回地址会被函数的返回地址所覆盖，所以需要在进入函数前就保存好ra的值，所以要用caller-saved寄存器。

实验内容

一、词法语法分析

解析：

在`frontend/parser/ply_parser.py`中实现step9中所列出的新增加的语法规范，包括program、parameter_list、params、expression_list、exprs、postfix等。

设计ast节点：

Program类：修改Program类，让其支持多个children。此外，由于在debug的时候发现Namer的transform函数中的program的children和functions函数返回的列表不同，导致failcases中的几个点无法输出错误信息，所以修改了Program类的functions函数：

```
class Program(ListNode[Union["Function", "Declaration"]]):
    """
    AST root. It should have only one children before step9.
    """

    def __init__(self, *children: Union[Function, Declaration]) -> None:
        super().__init__("program", list(children))

    def functions(self) -> dict[str, Function]:
        f = {}
        for func in self:
            if isinstance(func, Function):
                if func.ident.value in f:
                    raise DecafDeclConflictError(func.ident.value)
                else:
                    f[func.ident.value] = func
        return f
        # return {func.ident.value: func for func in self if isinstance(func, Function)}

    ...
```

Function类：增加参数`params: ParameterList`

Parameter类：指function的某个parameter

ParameterList类：一个parameter的列表

Postfix类：即实验文档中的Call节点，用来调用function的

ExpressionList类：一个expression的列表，作为Postfix的参数

二、语义分析

`frontend/typecheck/namer.py`：

增加`visitFunction`函数、`visitParameter`函数、`visitExpressionList`函数和`visitPostfix`函数。

```
    def visitFunction(self, func: Function, ctx: ScopeStack) -> None:
        # func.body.accept(self, ctx)
        # print(GlobalScope.symbols)
        # print(func.__dict__)
        if ctx.isConflict(func.ident.value) or GlobalScope.lookup(func.ident.value):
            raise DecafDeclConflictError(func.ident.value)
        # breakpoint()
        newSymbol = FuncSymbol(func.ident.value, func.ret_t.type, ctx.currentScope())
```

```
        if func.params is not None:
            for param in func.params.children:
                newSymbol.addParaType(param.var_t.type)
        GlobalScope.declare(newSymbol)
        func.setattr("symbol", newSymbol)
        # print(func.__dict__)
        func_scope = Scope(ScopeKind.LOCAL)
        ctx.addScope(func_scope)
        # ctx.declare(newSymbol)
        func.params.accept(self, ctx)
        # func.body.accept(self, ctx, func.params)
        for child in func.body.children:
            child.accept(self, ctx)
        ctx.popScope()

    def visitParameter(self, param: Parameter, ctx: ScopeStack) -> None:
        if ctx.lookup(param.ident.value):
            raise DecafDeclConflictError(param.ident.value)
        newSymbol = VarSymbol(param.ident.value, param.var_t.type)
        ctx.declare(newSymbol)
        param.setattr("symbol", newSymbol)
        param.ident.setattr('type', newSymbol.type)

    def visitParameterList(self, params: ParameterList, ctx: ScopeStack) -> None:
        for child in params.children:
            child.accept(self, ctx)

    def visitExpressionList(self, exprlist: ExpressionList, ctx: ScopeStack) -> None:
        for child in exprlist.children:
            child.accept(self, ctx)

    def visitPostfix(self, postfix: Postfix, ctx: ScopeStack) -> None:
        # breakpoint()
        # if not ctx.lookup(postfix.ident.value): raise
DecafUndefinedVarError(postfix.ident.value)
        if ctx.lookup_top(postfix.ident.value):
            raise DecafBadFuncCallError(postfix.ident.value)
        func = GlobalScope.lookup(postfix.ident.value)
        if not func or not func.isFunc: raise DecafUndefinedVarError(postfix.ident.value)
        if func.parameterNum != len(postfix.exprlist):
            raise DecafBadFuncCallError(postfix.ident.value)
        postfix.ident.setattr('symbol', func)
        postfix.setattr('type', func.type)
        for expr in postfix.exprlist:
            expr.accept(self, ctx)
```

主要注意function的symbol是全局的，应该放在全局作用域里，在运行function的内容时要开一个局部作用域。

还有注意一些边界情况，如在`visitPostfix`函数里要查看函数名是否在顶部作用域里，而不是查看其是否在所有的作用域里，不然failcases某些案例可能无法通过。

因此我还在`frontend/scope/scopestack.py`中添加了在当前（顶部）作用域查看变量的函数`lookup_top`：

```
def lookup_top(self, name: str) -> Optional[Symbol]:
    return self.currentScope().lookup(name)
```

三、中间代码生成

1.`frontend/tacgen/tacgen.py`:

在`TACFuncEmitter`类中添加`visitParam`函数和`visitCall`函数:

```
    def visitParam(self, value: Temp) -> None:
        self.func.add(Param(value))

    def visitCall(self, label: Label) -> Temp:
        temp = self.freshTemp()
        self.func.add(Call(temp, label))
        return temp
```

在`TACGen`类中:

修改`transform`函数:

考虑真正的parameter的数量与调用:

```
    def transform(self, program: Program) -> TACProg:
        labelManager = LabelManager()
        tacFuncs = []
        tacGlobalVars = program.globalVars()
        for funcName, astFunc in program.functions().items(): #遍历每个函数?
            # in step9, you need to use real parameter count
            emitter = TACFuncEmitter(FuncLabel(funcName), len(astFunc.params.children),
labelManager)
            for child in astFunc.params.children:
                child.accept(self, emitter)
            astFunc.body.accept(self, emitter) #调用不同的visit函数
            tacFuncs.append(emitter.visitEnd())
        return TACProg(tacFuncs, tacGlobalVars)
```

添加`visitPostfix`函数和`visitParameter`函数:

```
    def visitPostfix(self, postfix: Postfix, mv: TACFuncEmitter) -> None:
        # print("visitPostfix")
        for expr in postfix.exprlist.children:
            expr.accept(self, mv)
        for expr in postfix.exprlist.children:
            mv.visitParam(expr.getattr("val"))
        postfix.setattr('val', mv.visitCall(FuncLabel(postfix.ident.value)))

    def visitParameter(self, param: Parameter, mv: TACFuncEmitter) -> None:
        # print("visitparameter")
        param.getattr("symbol").temp = mv.freshTemp()
```

2.`utils/tac`：

在`tacinstr.py`中添加Param类和Call类：

```
class Param(TACInstr):
    def __init__(self, param: Temp) -> None:
        super().__init__(InstrKind.PARAM, [], [param], None)
        self.param = param

    def __str__(self) -> str:
        return "PARAM " + str(self.param)

    def accept(self, v: TACVisitor) -> None:
        v.visitParam(self)

class Call(TACInstr):
    def __init__(self, param: Temp, label: Label) -> None:
        super().__init__(InstrKind.CALL, [param], [], label)
        self.param = param
        self.label = label

    def __str__(self) -> str:
        return str(self.param) + " = CALL " + str(self.label)

    def accept(self, v: TACVisitor) -> None:
        v.visitCall(self)
```

此外，相应地修改`tacvisitor.py`、`visitor.py`和`taccop.py`相关函数。

四、目标代码生成

`riscv.py`

1.模仿`SPAdd`类写`FPAdd`类，用于保存和恢复栈帧：

```
    class FPAdd(BackendInstr):
        def __init__(self, offset: int) -> None:
            super().__init__(InstrKind.SEQ, [Riscv.FP], [Riscv.SP], None)
            self.offset = offset

        def __str__(self) -> str:
            assert -2048 <= self.offset <= 2047  # Riscv imm [11:0]
            return "addi " + Riscv.FMT3.format(
                str(Riscv.FP), str(Riscv.SP), str(self.offset)
            )
```

2.实现`Param`类和`Call`类：

```
    class Param(BackendInstr):
```

```
        def __init__(self, src: Reg) -> None:
            super().__init__(InstrKind.PARAM, [], [src], None)
            self.src = src

        def __str__(self) -> str:
            return "beq " + str(self.src)

    class Call(BackendInstr):
        def __init__(self, target: Label) -> None:
            super().__init__(InstrKind.CALL, [], [], target)
            self.target = target

        def __str__(self) -> str:
            return "call " + super(FuncLabel, self.target).__str__()
```

`backend/reg/bruteregalloc.py`:

`accept` 函数:

使用`self.numArgs`记录函数自身的参数，`self.functionParams`记录子函数所使用的参数对应的虚拟寄存器，`self.callerSavedRegs`保存`caller_saved`寄存器。首先使用`bind`将实参与寄存器绑定并保存到栈上，然后再接着分析。

`allocForLoc` 函数:

实现了`allocForParam`函数和`allocForCall`函数。

根据指令类型的不同为每条指令分配寄存器。

若为`Param`类型：保存前八个参数到寄存器中。

若为`Call`类型：调用前先保存caller_saved寄存器，然后将多余的参数保存到栈上，调用后恢复除A0外的所有caller_saved寄存器。

```
    def allocForLoc(self, loc: Loc, subEmitter: RiscvSubroutineEmitter):
        instr = loc.instr
        srcRegs: list[Reg] = []
        dstRegs: list[Reg] = []

        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))
        # instr.fillRegs(dstRegs, srcRegs)
        # subEmitter.emitAsm(instr)
```

```
        if instr.kind == InstrKind.PARAM:
            self.allocForParam(instr, srcRegs, subEmitter)
        elif instr.kind == InstrKind.CALL:
            self.allocForCall(instr, srcRegs, dstRegs, subEmitter)
        else:
            instr.fillRegs(dstRegs, srcRegs)
            subEmitter.emitAsm(instr)

    def allocForParam(self, instr: BackendInstr, srcRegs: list[Reg], subEmitter:
RiscvSubroutineEmitter):
        # 保存前八个参数到寄存器中
        if self.callerParamCount() < self.maxNumParams:
            reg = Riscv.ArgRegs[self.callerParamCount()]
            # 将寄存器解绑，稍后恢复
            if reg.occupied:
                subEmitter.emitStoreToStack(reg)
                self.callerSavedRegs[reg] = reg.temp
                self.unbind(reg.temp)
            subEmitter.emitReg(reg, srcRegs[0])
        self.functionParams.append(instr.srcs[0])

    def allocForCall(self, instr: BackendInstr, srcRegs: list[Reg], dstRegs: list[Reg],
subEmitter: RiscvSubroutineEmitter):
        # 调用前保存 caller-saved 寄存器
        for reg in Riscv.CallerSaved:
            if reg.occupied:
                subEmitter.emitStoreToStack(reg)
                self.callerSavedRegs[reg] = reg.temp
                self.unbind(reg.temp)

        # 保存多余的参数到栈中
        if self.callerParamCount() > self.maxNumParams:
            for (index, temp) in enumerate(self.functionParams[self.maxNumParams:][::-1]):
                subEmitter.emitStoreParamToStack(temp, index)
            subEmitter.emitNative(instr)
            subEmitter.emitRestoreStackPointer(4 * (self.callerParamCount() -
self.maxNumParams))
        else:
            # breakpoint()
            subEmitter.emitNative(instr)
        self.functionParams = []

        # 调用后恢复 caller-saved 寄存器
        for reg, temp in self.callerSavedRegs.items():
            # 返回值寄存器不需要恢复，否则会覆盖
            if reg != Riscv.A0:
                self.bind(temp, reg)
                subEmitter.emitLoadFromStack(reg, temp)
        self.callerSavedRegs = {}
```

`allocRegFor`函数：

```
def allocRegFor(
```

```
    self, temp: Temp, isRead: bool, live: set[int], subEmitter: RiscvSubroutineEmitter
):
    if temp.index in self.bindings:
        return self.bindings[temp.index]

    for reg in self.emitter.allocatableRegs:
        if (not reg.occupied) or (not reg.temp.index in live):
            subEmitter.emitComment(
                "  allocate {} to {}  (read: {}):".format(
                    str(temp), str(reg), str(isRead)
                )
            )
            if isRead:
                # subEmitter.emitLoadFromStack(reg, temp)
                # 如果是存储在栈上的参数，利用 FP 从栈中加载
                if (self.maxNumParams <= temp.index < self.numArgs):
                    subEmitter.emitLoadParamFromStack(reg, temp.index)
                # 否则，利用 SP 从栈中加载
                else:
                    subEmitter.emitLoadFromStack(reg, temp)
    ...
```

`backend/subroutineinfo.py`:

修改info的结构，增加`numArgs`参数，即函数的参数:

```
class SubroutineInfo:
    # def __init__(self, funcLabel: FuncLabel) -> None:
    #     self.funcLabel = funcLabel
    def __init__(self, funcLabel: FuncLabel, numArgs: int) -> None:
        self.funcLabel = funcLabel
        self.numArgs = numArgs

        self.offsets: Dict[str, int] = {}
        self.size = 0

    def __str__(self) -> str:
        return "funcLabel: {}".format(
            self.funcLabel.name,
        )
```

`backend/riscv/riscvasmemitter.py`:

1.`RiscvInstrSelector`类

增加`visitParam`函数和`visitCall`函数:

```
        def visitParam(self, instr: Param) -> None:
            self.seq.append(Riscv.Param(instr.param))

        def visitCall(self, instr: Call) -> None:
            self.seq.append(Riscv.Call(instr.label))
            self.seq.append(Riscv.Move(instr.param, Riscv.A0))
```

2. `RiscvSubroutineEmitter`类

实现`emitNative`函数、`emitReg`函数、`emitRestoreStackPointer`函数、`emitStoreParamToStack`函数和`emitLoadParamFromStack`函数，用于保存参数到寄存器或栈上以及恢复栈指针：

```
    def emitNative(self, instr: BackendInstr):
        self.buf.append(instr)

    def emitReg(self, dst: Reg, src: Temp):
        self.buf.append(Riscv.Move(dst, src))

    # restore stack after calling a function
    def emitRestoreStackPointer(self, offset:int) -> None:
        self.buf.append(Riscv.SPAdd(offset))

    # store some param to stack
    def emitStoreParamToStack(self, src: Temp, index: int) -> None:
        self.buf.append(Riscv.SPAdd(-4))
        self.buf.append(Riscv.NativeLoadWord(Riscv.T0, Riscv.SP, self.offsets[src.index] +
4 * index + 4))
        self.buf.append(Riscv.NativeStoreWord(Riscv.T0, Riscv.SP, 0))

    # load some param from stack
    def emitLoadParamFromStack(self, dst: Reg, index: int) -> None:
        self.buf.append(Riscv.NativeLoadWord(dst, Riscv.FP, 4 * (index - 8)))
```

修改`emitFunc`函数打印riscv码的逻辑：

```
    def emitFunc(self):
        self.printer.printComment("start of prologue")
        self.printer.printInstr(Riscv.SPAdd(-self.nextLocalOffset))
        self.printer.printInstr(Riscv.NativeStoreWord(Riscv.RA, Riscv.SP, 4 *
len(Riscv.CalleeSaved) + self.info.size))
        self.printer.printInstr(Riscv.NativeStoreWord(Riscv.FP, Riscv.SP, 4 *
len(Riscv.CalleeSaved) + self.info.size + 4))
        self.printer.printInstr(Riscv.FPAdd(self.nextLocalOffset))

        # in step9, you need to think about how to store RA here
        # you can get some ideas from how to save CalleeSaved regs
        for i in range(len(Riscv.CalleeSaved)):
            if Riscv.CalleeSaved[i].isUsed():
                self.printer.printInstr(
                    Riscv.NativeStoreWord(Riscv.CalleeSaved[i], Riscv.SP, 4 * i +
self.info.size)
```

```
                )

        self.printer.printComment("end of prologue")
        self.printer.println("")

        self.printer.printComment("start of body")

        # in step9, you need to think about how to pass the parameters here
        # you can use the stack or regs

        # using asmcodeprinter to output the RiscV code
        for instr in self.buf:
            # print(instr.__dict__)
            self.printer.printInstr(instr)

        self.printer.printComment("end of body")
        self.printer.println("")

        self.printer.printLabel(
            Label(LabelKind.TEMP, self.info.funcLabel.name + Riscv.EPILOGUE_SUFFIX)
        )
        self.printer.printComment("start of epilogue")
        # self.printer.printInstr(Riscv.SPAdd(-self.nextLocalOffset))
        self.printer.printInstr(Riscv.NativeLoadWord(Riscv.RA, Riscv.SP, 4 *
len(Riscv.CalleeSaved) + self.info.size))
        self.printer.printInstr(Riscv.NativeLoadWord(Riscv.FP, Riscv.SP, 4 *
len(Riscv.CalleeSaved) + self.info.size + 4))
        # self.printer.printInstr(Riscv.FPAdd(self.nextLocalOffset))

        for i in range(len(Riscv.CalleeSaved)):
            if Riscv.CalleeSaved[i].isUsed():
                self.printer.printInstr(
                    Riscv.NativeLoadWord(Riscv.CalleeSaved[i], Riscv.SP, 4 * i +
self.info.size)
                )

        self.printer.printInstr(Riscv.SPAdd(self.nextLocalOffset))
        self.printer.printComment("end of epilogue")
        self.printer.println("")

        self.printer.printInstr(Riscv.NativeReturn())
        self.printer.println("")
```

五、关于实验框架的一个小问题

在`backend/asm.py`中，`Asm`类的`transform`函数应该在循环外创建emitter，而不是每次循环都创建一个，否则最后打印出来的只有main函数，无法显示其他函数（这个bug找了很久因为我还以为是我backend写的有问题导致的TT）

```
class Asm:
    def __init__(self) -> None:
        pass
```

```
    def transform(self, prog: TACProg):
        analyzer = LivenessAnalyzer()
        emitter = RiscvAsmEmitter(Riscv.AllocatableRegs, Riscv.CallerSaved)
        for func in prog.funcs:
            # print(func.__dict__)
            # emitter = RiscvAsmEmitter(Riscv.AllocatableRegs, Riscv.CallerSaved)
            reg_alloc = BruteRegAlloc(emitter)
            pair = emitter.selectInstr(func) #指令选择将中端TAC代码转换为riscv汇编代码
            builder = CFGBuilder()
            cfg: CFG = builder.buildFrom(pair[0])
            analyzer.accept(cfg)
            reg_alloc.accept(cfg, pair[1])
            # breakpoint()

        return emitter.emitEnd()
```

Honor Code

在目标代码生成的实现中参考了https://github.com/chengsx21/MiniDecaf/tree/main的部分内容，主要是关于每个文件中应该增加或修改哪部分内容。