

STAGE5 REPORT: 函数

赵宇峰 2021013376 致理书院

1. 前端

i)

词法分析与语法分析

根据规范的要求，加上了相应文法（分情况记录到 ply_parser 中，此处略去）

Function	返回类型 <code>return_type</code> ，函数名 <code>ident</code> ，参数列表 <code>params</code> ，函数体 <code>body</code>	函数
Parameter	参数类型 <code>var_type</code> ，变量名 <code>ident</code>	函数参数
Call	调用函数名 <code>ident</code> ，参数列表 <code>argument_list</code>	函数调用

AST 节点：

Function:增加了 `paramList`

Call:增加了 `argument_list`

Parameter: 参数类型 `var_type`，变量名 `ident`

```
class Parameter(Declaration):
    def __init__(self, var_type: TypeLiteral, ident: Identifier):
        super().__init__(var_type, ident)
        self.var_type = var_type
        self.ident = ident

    def accept(self, v: Visitor[T, U], ctx: T) -> Optional[U]:
        return v.visitParameter(self, ctx)
```

为了方便在 Program 的子节点加入函数，重载了 Program 的自增符号。

ii)语义分析

1) Program:

2) 对 function 的类型检查

需要建立局部作用域，分别检查函数、参数和函数体，同时考虑声明冲突和重定义

```
def visitFunction(self, func: Function, ctx: ScopeStack) -> None:
    #建立函数符号
    func_symbol=FuncSymbol(func.ident.value,func.rettype.type,ctx.get_current_scope())
    for param in func.parameterList:
        func_symbol.addParaType(param.var_type.type)
    #考虑可能的同名变量冲突:
    if not ctx.findConflict(func.ident.value):
        #如果没有声明则声明并放入当前作用域符号表
        ctx.declare(func_symbol)
        func.setattr('symbol', func_symbol)
```

```

else:
    if isinstance(ctx.lookup(func.ident.value)
,FuncSymbol) == False:
        raise DecafDeclConflictError(func.ident.value)
    else:#冲突而且是函数类型，则声明
        ctx.declare(func_symbol)
        func.setattr(func_symbol)

if func.body is None:
    pass
if not func_symbol.defined:
    func_symbol.defined = True
else:
    #排除函数定义两次的情况
    raise DecafFunctionDefinedTwiceError(f"function {func.ident.value} is defined
twice")
self.defined = True
with ctx.local():
    #再增加函数本身的作用域，并分别检查内部的参数和函数体
    for parameter in func.parameterList:
        parameter.accept(self, ctx)
    for stmt in func.body.children:
        stmt.accept(self, ctx)

```

同时实现了 visitCall 和 visitParameter:

```

def visitParameter(self, that: Parameter, ctx: T) -> None:
    return self.visitDeclaration(that, ctx)

```

考虑几类常见的错误，如未声明、参数长度不一致。由于本框架内只有 int 类型，暂未考虑检验（之后可能要有修正）

```

def visitCall(self, call: Call, ctx: ScopeStack) -> None:
    func: FuncSymbol = ctx.lookup(call.ident.value)
    if func is None or not func.isFunc:
        raise DecafUndefinedFuncError(call.ident.value)
    if len(call.argument_list) != func.parameterNum:
        raise DecafBadFuncCallError(f"Invalid parameter length, expecting
{func.parameterNum} parameters, but got {len(call.argument_list)}")
    call.setattr('symbol',func)
    for arg in call.argument_list:
        arg.accept(self, ctx)

```

2. 中间代码生成

Entry of this phase

```

def transform(self, program: Program) -> TACProg:
    handler=Handler(program.functions().values(),[(name,

```

```

decl.getattr('symbol').initValue)for name])
for funcName, astFunc in program.functions().items():
    if astFunc.body is NULL:
        continue
    argnum = len(astFunc.parameterList)
    emitter = handler.visitFunc(funcName, argnum)
    astFunc.accept(self, emitter)
    emitter.visitEnd()
return handler.visitEnd()

```

```

class Handler:
    def __init__(self, funcs: List[Function]) -> None:
        self.funcs = []
        self.globalDecls = globalDecls
        for func in funcs:
            self.funcs.append(func)
            self.labelManager.putFuncLabel(func.ident.value)

    def visitMainFunc(self) -> TACFuncEmitter:
        entry = MAIN_LABEL
        return TACFuncEmitter(entry, 0, self.labelManager)

    def visitFunc(self, name: str, numArgs: int) -> TACFuncEmitter:
        entry = self.labelManager.getFuncLabel(name)
        return TACFuncEmitter(entry, numArgs, self.labelManager)

    def visitEnd(self) -> TACProg:
        return TACProg(self.labelManager.funcs, self.globalDecls)

```

使用了中间的类型 handler 来处理所有的函数 tac 生成，并区分了 main 函数和其他函数，在本次实现中使用了 Param—Call 的方式来实现函数调用。具体可以看 call 的实现：

```

def visitCall(self, call: Call, mv: TACFuncEmitter) -> None:
    param_temp = []
    for arg in call.argument_list:
        arg.accept(self, mv)
        if not (arg.getattr('val')):
            raise SyntaxError('Error in arg fetching!')
        param_temp.append(arg.getattr('val'))
    ret = mv.freshTemp()
    call.setattr('val', ret)
    funcLabel = mv.labelManager.getFuncLabel(call.ident.value)
    mv.visitCall(funcLabel, ret, param_temp)

```

首先，遍历所有参数表达式，为每个参数表达式求值，将其结果存储在临时变量中。然后，

分配一个新的临时变量 `mv.freshTemp` 用于存储函数调用的返回值，并将其与函数调用节点绑定。最后通过调用 `mv.visitCall` 访问函数,生成 `call[list[identifier(***)]`这样的 tac 表示。

3. 后端

处于简单实现的考虑，这里并没有使用 caller-callee 约定，而是采取了统一传参的办法，并统一压栈以避免可能的冲突

实现 Riscv 的 call 指令：

```
class Call(TACInstr):
    @classmethod
    def emitcall(self, call):
        if not isinstance(call, Call):
            raise DecafBadFuncCallError('Wrong calling')
        return self(call.label, call.srcs, call.dsts[0])

    def __init__(self, funcLabel: Label, param_list: List[Temp], dst: Temp):
        super().__init__(InstrKind.CALL, [dst], param_list, funcLabel)

    def __str__(self) -> str:
        return f"call {self.label.name}"Riscvasmmemmiter:
```

并在 `riscvasmmemmiter` 中增加了 `visitcall` 方法。

参数传递遵循以下的流程：

保存所有活跃变量寄存器。

准备参数，完成传参。

执行汇编中的函数调用指令，开始执行子函数直至其返回。

拿到函数调用的返回值，作为函数调用表达式的值。

恢复所有活跃变量寄存器。

具体的，在 `bruteregalloc.py` 中：

```
def accept(self, graph: CFG, info: SubroutineInfo) -> None:
    subEmitter = self.emitter.emitSubroutine(info)
    for reg in self.emitter.allocatableRegs:#将 A0~A7 置可用
        reg.used = False
    for temp, reg in zip(subEmitter.info.argTemps, Riscv.ArgRegs):#
        if(reg is not None):
            self.bind(temp, reg)
        else:
            raise RuntimeError("No available reg for args")
    if (len(graph) > 0 and len(graph.nodes[0].liveIn)>0) :
        for tempindex in graph.nodes[0].liveIn:#对于所有的活跃变量
            if self.bindings.get(tempindex) is not None:#首先一律压栈
                subEmitter.emitStoreToStack(self.bindings.get(tempindex))
    self.bindings.clear()#这些活跃变量即可释放
    for reg in self.emitter.allocatableRegs:
        reg.occupied = False
```

```

for bb in graph.iterator():
    # you need to think more here
    # maybe we don't need to alloc regs for all the basic blocks
    if bb.label is not None:
        subEmitter.emitLabel(bb.label)
    self.localAlloc(bb, subEmitter)#保持原来为所有基本块分配寄存器的设计,迭代
进行
    subEmitter.emitEnd()

```

在 localAlloc 方法中传递参数时，我们将超过 8 个寄存器限制的参数压栈，并在需要时调用它们（我们同时使用了 index 记录是超过的第几个参数，我们把这些参数统一绑定到 T0 以备使用）：

```

if len(bb.locs[0].instr.srcs) > 8:
    subEmitter.emitNative(Riscv.SPAdd(-4 * (len(bb.locs[0].instr.srcs)-8)))
    subEmitter.adjustSP(-(len(bb.locs[0].instr.srcs)-8) * 4)
    for temp in bb.locs[0].instr.srcs[8:]:
        subEmitter.emitLoadFromStack(Riscv.T0, temp)
        self.bind(temp, Riscv.T0)
        subEmitter.emitNative(Riscv.NativeStoreWord(reg, Riscv.T0, 4 *
temp.index))
        self.unbind(temp)

```

现在就可以读取所需参数到寄存器中并且发射指令了

```

for temp, reg in zip(bb.locs[0].instr.srcs, Riscv.ArgRegs):
    subEmitter.emitLoadFromStack(reg, temp)
    self.bind(temp, reg)
subEmitter.emitNative(bb.locs[0].instr.toNative(bb.locs[0].instr.dsts,
bb.locs[0].instr.dsts))
完成后，将所有压入内容退栈，取消绑定
if len(bb.locs[0].instr.srcs) > 8:
    subEmitter.emitNative(Riscv.SPAdd(-4 * (len(bb.locs[0].instr.srcs)-8)))
    subEmitter.adjustSP(-4 * (len(bb.locs[0].instr.srcs)-8))
if len(bb.locs[0].instr.srcs) > 0:
    self.unbind(bb.locs[0].instr.srcs[0])
返回值绑定在 A0 上
self.bind(bb.locs[0].instr.dsts[0], Riscv.A0)
subEmitter.emitStoreToStack(Riscv.A0)

```

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
```

我更倾向于采用传参和调用分离的办法，一方面实现起来更加简洁，另一方面也为可能的不定参数的情况提供了便利。但是这种方法出错的概率更大，如果有多个函数调用，或者函数里面套函数，容易混淆。

而如果使用一整条函数调用会更加清晰，但实现起来更加复杂。同时，不定参数的设计会比较难处理，而且出现较多参数调用的时候指令会非常冗长。

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？

在 callee 端口上，使用 callee-saved 寄存器的目的是保存被调用的函数可能修改的寄存器值，以确保函数返回后这些寄存器的值保持不变。这有助于避免在函数调用之间不必要地保存和恢复这些值，提高性能。在 caller 端口，caller-saved 寄存器用于保存 caller 在函数调用期间需要保持不变的寄存器值。这种协作性质确保在调用函数时，caller 保存的寄存器值不会被调用函数修改，而在返回时可以正确地恢复，避免错误。

在本函数调用“子函数”时，ra 本来存的是调用本函数的“父函数”的返回地址，现在要调用子函数，很自然的策略就是由本函数存储它。而子函数和父函数“隔了两代”，让子函数来存 ra 会非常棘手。