stage-5 实验报告

于新雨 计25 2022010841

新增代码

前端

- ply_parser.py
  - 增加对于函数定义和函数调用的支持，其中我们增加了对函数定义和调用的解析，同时对program,expression_precedence等函数的解析进行完善
  - 对函数定义的支持：

```
def p_declaration_list_1(p):
    """
    declaration_list : declaration Comma declaration_list
    """
    if p[1]:
        p[3].children.append(p[1])
    p[0] = p[3]

def p_declaration_list_2(p):
    """
    declaration_list : declaration
    """
    p[0] = DeclarationList()
    if p[1]:
        p[0].children.append(p[1])


def p_declaration_list_empty(p):
    """
    declaration_list : empty
    """
    p[0] = DeclarationList()

def p_function_def(p):
    """
    function : type Identifier LParen declaration_list RParen LBrace block
RBrace
    """
    p[0] = Function(p[1], p[2], p[7],p[4])
```

  - 对函数调用的支持：

```
def p_expr_list_1(p):
"""
```

```
    expr_list : expression Comma expr_list
    """
        if p[1]:
            p[3].append(p[1])
        p[0] = p[3]

    def p_expr_list_2(p):
        """
        expr_list : expression
        """
        p[0] = [p[1]]

    def p_expr_list_empty(p):
        """
        expr_list : empty
        """
        p[0] = []
    def p_funccall(p):
        """
        funccall : Identifier LParen expr_list RParen
        """
        p[0]=Call(p[1],p[3])
```

- 完善p_program和expression_precedence:

```
    def p_program_1(p):
        """
        program :  program function
        """
        if p[2]:
            p[1].children.append(p[2])
        p[0]=p[1]
```

在p_expression_precedence的docstring中增加函数调用的优先级`unary : funccall`

- tree.py
  - DeclarationList 方便对函数的参数的解析，我们加入DeclarationList类型，其中参考Block的继承关系，我们发现有ListNode这一层包装可以支持这一类的列表类型，所以我们让DeclarationList继承ListNode["Declaration"]

```
    class DeclarationList(ListNode["Declaration"]):
    """
    AST node of function parameter list
    """
    def __init__(self, *children: Declaration) -> None:
        super().__init__("declarationlist", list(children))
```

```
    def accept(self, v: Visitor[T, U], ctx: T):
        return v.visitDecl(self, ctx)
```

- Function 在之前的class Function中，没有考虑传参的情况，我们只需要将node的组成加上DeclarationList，之后修改len函数即可
- Call

```
class Call(Expression):
"""
AST node of function call.
"""

def __init__(self, ident: Identifier, args: [Temp]) -> None:
    super().__init__("func_call")
    self.ident = ident
    self.args = args

def accept(self, v: Visitor[T, U], ctx: T):
    return v.visitCall(self, ctx)
def __getitem__(self, key: int) -> Node:
    return [self.ident,self.args][key]
def __len__(self) -> int:
    return len(self.args) + 1
def __str__(self)->str:
    return f"{self.ident}({','.join(map(str,self.args))})"
```

- tacgen.py 在FuncEmitter中，增加对函数调用语句的支持：

```
    def visitCall(self, func: str, dst:Temp,args: [Temp]) -> None:
        self.func.add(Call(func, dst,args))
```

Tacgen的transform里面增加对于函数参数的支持：

```
            emitter = TACFuncEmitter(Label(LabelKind.FUNC,funcName), 0,
labelManager)
            astFunc.para_list.accept(self, emitter)
```

增加对于传参所用的临时变量的支持,即是将DeclarationList里面所有参数所分配到的临时变量填到自身的“tmps”的attribute中：

```
    def visitDecl(self, decl: DeclarationList, mv: TACFuncEmitter) -> None:
        decl.setattr("tmps",[])
        for child in decl:
            child.accept(self,mv)
            decl.getattr("tmps").append(child.getattr("symbol").temp)
```

增加对于call语句的支持：

```
    def visitCall(self, that: Call, mv:TACFuncEmitter) -> None:
        para_list=[]
        for i in that.args:
            i.accept(self, mv)
            para_list.append(i.getattr("val"))
        that.setattr("val", mv.freshTemp())
        mv.visitCall(that.ident.value,that.getattr("val"),para_list)
```

- namer.py
  - 对于函数定义的支持
    - 我们在作用域栈中查询当前函数名，如果查到重复的就报错`DecafDeclConflictError`。我们接着用该函数的名称，返回值，当前作用域定义一个FuncSymbol并且针对其中参数添加ParaType,然后将当前FuncSymbol放入作用域中，并且将函数的symbol属性设置为这个FuncSymbol
    - 我们接着压作用域，并且visit参数表和函数体

```
def visitFunction(self, func: Function, ctx: ScopeStack) -> None:
    #create function symbol in current scope
    if not ctx.lookup(func.ident.value):
        var=FuncSymbol(func.ident.value,func.ret_t.name,ctx.top())
        for i in func.para_list.children:
            var.addParaType(i.var_t.type)
        ctx.top().declare(var)
        func.setattr('symbol',var)
    else:
        raise DecafDeclConflictError(func.ident.value)
    ctx.push(Scope(ScopeKind.LOCAL))
    func.para_list.accept(self, ctx)
    for child in func.body:
        child.accept(self, ctx)
    ctx.pop()
```

  - 对于函数调用的支持
    - 我们增加对函数声明和参数数量的检查，并且把call的symbol设置为对应的FuncSymbol，之后依次对参数进行访问

```
    def visitCall(self, that: Call, ctx: T) -> None:
    if not ctx.lookup(that.ident.value):
        raise DecafUndefinedVarError(that.ident.value)

    that.setattr("symbol",ctx.lookup(that.ident.value))
    if len(that.args)!=len(ctx.lookup(that.ident.value).para_type):
        raise DecafBadFuncCallError(that.ident.value)
```

```
        for arg in that.args:
            arg.accept(self, ctx)
```

- utils/tacinstr 增加对Call类型的序列化的方法和accept方法

```
class Call(TACInstr):
    def __init__(self, func: str,dst:Temp, args: [Temp]) -> None:
        super().__init__(InstrKind.CALL,[],args,None)
        self.func = func
        self.args=args
        self.dst=dst
    def __str__(self) ->str:
        ret_str=self.dst.__str__()+" = call "+self.func+"("
        for i in self.args:
            ret_str+=i.__str__()+","
        ret_str=ret_str[:-1]+")"
        return ret_str
    def accept(self, v: TACVisitor) -> None:
        v.visitCall(self)
```

- 此外还有一些辅助的函数等，就不列出来了

后端

后端的增加总体来说比较复杂

- dataflow 一开始试图将call指令单独当作一个block，但是和助教讨论后发现过于复杂，就放弃了，所以dataflow基本上没加什么
- reg/bruteregalloc
  - 首先在init里面给自身增加一个成员callersaves
  - 主要添加较多的是alloc_for_loc函数，即是对每一条语句判断寄存器分配的函数，添加对call语句的支持如下：
    - 我们先判断该语句是否为call语句，对于call语句，我们先处理caller-saves,对于已经用过的caller-saves就存在栈上
    - 此外我们注意到所有的参数此时都在寄存器里面，我们就把超过8个的参数放到栈上
    - 接着我们判断是否有已有参数用到了传参寄存器，如果有的话也存到栈上
    - 接着我们按照顺序把8个或者以下的传参寄存器填满就行了。
    - 在调用完call语句之后，我们去恢复所有存在栈上的caller-saves
- riscvasmemitter
  - 对emitLoadFromStack函数增加如下：

```
        if src.index not in self.offsets:
        if src.index<8:
            self.buf.append(Riscv.Move(dst,Riscv.ArgRegs[src.index]))
        else:
```

```
                self.buf.append(Riscv.NativeLoadWord(dst, Riscv.SP, (src.index-
7)*4+self.nextLocalOffset))
```

我们注意到这个函数可能在对于函数传参的load上面用到，此时我们需要进行特判，如果想要寻找的index不在自己记录的保存的offset里面，则是对可能的函数参数进行load操作，我们根据参数的index判断传参的位置，然后进行move或者load操作即可

- 此外，在`emitEnd`函数中进行了一些维护栈帧的操作，和对RA的恢复
- 增加对call的visit函数：

```
        def visitCall(self, instr: Call)->None:

self.seq.append(Riscv.Call(instr.dst,instr.args,Label(LabelKind.FUNC,instr.f
unc)))
        self.seq.append(Riscv.Move(instr.dst,Riscv.A0))
```

- utils/riscv
  - 我们增加对Call类型的支持：

```
    class Call(TACInstr):
    def __init__(self,dst: Temp, src: [Temp],target:Label)->None:
        super().__init__(InstrKind.SEQ,[dst],src,target)
        self.target=target

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

注意此时call的类型还是SEQ的

- 增加对于Store语句的支持如下

```
    class StoreExpression(TACInstr):
    def __init__(self,src:Temp,offset:int)->None:
        super().__init__(InstrKind.SEQ,None,[src],None)
        self.offset=offset
    def __str__(self) -> str:
        return "sw
"+Riscv.FMT_OFFSET.format(str(self.srcs[0]),str(self.offset),str(Riscv.SP))
```

思考题

1. 我更倾向于用"一整条函数调用"的方式

- "一整条函数调用"的优势：
  1. 个人感觉在这种类型单一的设计中，把传参和函数调用分开可能意义不大，一般情况下不是很需要单独标注

    2. 一整条函数调用的方式同样可以保存传参信息，在当前这种情况下统一在对待call语句时处理更加简单方便
    3. 简洁，可读性好
- “一整条函数调用”的缺点：
    1. 可拓展性可能较差，比如当传参处理有针对字长或者浮点数专用寄存器的时候将call语句和传参分离可能较好
    2. 如果参数太多的话可能影响代码可读性
- “传参和调用分离”的优势：
    1. 可拓展性较好，当传参处理较为复杂比如需要考虑浮点数或者字长时，将call和传参分离更加适合
    2. 可以应对各种数目的参数
- “传参和调用分离”的缺点：
    1. 不同PARAM语句之间需要维护是第几个参数的信息，会比统一处理要繁琐一些
    2. 无法在处理传参时完全明确各个param之间的顺序，可能不利于编译优化

2.
    - 除了传参需要caller-saves以外，同时设计caller-save和callee-saves的一个主要的incentive是变量生命周期，我们在一个生成的tac中用到的变量可以分为两种，一种是生命周期较长，可以跨越好几个函数调用的变量，还有就是用于局部计算的生命周期较短的变量，而caller-saves和callee-saves便于我们同时对于这两种情况进行处理
    - 对于生命周期较短，可能只在局部用到的变量，一般在每个函数调用中都会具有，所以我们需要有几个寄存器可以直接被函数拿来用做这种简单的计算之类的东西，所以就有caller-saves的寄存器
    - 对于生命周期较长的变量，在函数调用中有一个规律就是如果把函数调用关系视作一个树的话，那么叶子节点会占据相当大的开销，此时我们需要有callee-saves，就是callee尽量不去动这些寄存器，如果实在需要用到则要自己保存
    - 为什么ra是caller-save: 因为在调用约定中没有说callee有保存ra的职责，而且如果用jal或者jalr指令调用子函数则ra可能直接就被覆盖了，这样就会导致子函数返回时返回到错误的地址，所以比较保险的做法还是将ra看作caller-save