stage-5

计13 沈佳茗

一、实验内容

1. 前端

首先在 `frontend/ast/tree.py` 中新增 Parameter/Call 节点，更改 Function 节点。

Function 节点新增 params 变量（`List[Parameter]`），用来保存函数的参数，同时修改 `__getitem__` 与 `__len__` 函数。

Parameter 节点包含 var_t, ident, init_expr 三个成员变量，分别表示参数类型，标识符和初始值。

Call 节点包含 ident, argument_list 两个成员变量，分别表示标识符和参数列表。

然后修改 frontend/parser 文件夹中的文件。首先在 `frontend/lexer/lex.py` 中增加保留字 ','（逗号）。

然后在 `frontend/parser/ply_parser.py` 中增加有关参数以及参数列表 (parameter) 定义，表达式列表的定义，以及函数定义与调用有关的语法。

```
def p_parameter(p):
    """
    parameter : type Identifier
    """
    p[0] = Parameter(p[1], p[2])

def p_parameter_list_base(p):
    """
    parameter_list_base : type Identifier Coma
    """
    p[0] = Parameter(p[1], p[2])


def p_parameter_list_prefix_first(p):
    """
    parameter_list_prefix : parameter_list_base
    """
    p[0] = [p[1]]

def p_parameter_list_prefix(p):
    """
    parameter_list_prefix : parameter_list_prefix parameter_list_base
    """
    p[1].append(p[2])
    p[0] = p[1]

def p_parameter_list(p):
    """
    parameter_list : parameter_list_prefix parameter
```

```
    """
    p[1].append(p[2])
    p[0] = p[1]


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

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

def p_function_def_no(p):
    """
    function : type Identifier LParen RParen LBrace block RBrace
    """
    p[0] = Function(p[1], p[2], p[6], [])

def p_expression_list_base(p):
    """
    expression_list_base : expression Coma
    """
    p[0] = p[1]


def p_expression_list_prefix_first(p):
    """
    expression_list_prefix : expression_list_base
    """
    p[0] = [p[1]]


def p_expression_list_prefix(p):
    """
    expression_list_prefix : expression_list_prefix expression_list_base
    """
    p[1].append(p[2])
    p[0] = p[1]

def p_expression_list(p):
    """
    expression_list : expression_list_prefix expression
    """
    p[1].append(p[2])
    p[0] = p[1]

def p_call_multi(p):
```

```
    """
    call : Identifier LParen expression_list RParen
    """
    p[0] = Call(p[1], p[3])

def p_call_one(p):
    """
    call : Identifier LParen expression RParen
    """
    p[0] = Call(p[1], [p[3]])

def p_call_no(p):
    """
    call : Identifier LParen RParen
    """
    p[0] = Call(p[1], [])

def p_expression_precedence(p):
    """
    multiplicative : unary
    unary : postfix
    postfix : primary
       | call
    """
     p[0] = p[1]
```

2. 中端

2.1 语义分析

在类型检查时新增 `visitFunction`, `visitParameter`, `visitCall` 函数，同时修改 `visitBlock` 函数以满足 Function body 的检查需求。

在 `visitFunction` 中，首先寻找是否存在已定义的同名的函数符号(FuncSymbol)，如果有则报错。否则，为这个函数新建一个 FuncSymbol。然后检查函数的参数列表中是否存在同名的参数，如果有则报错；否则，在 FuncSymbol 中加入参数。使用 setattr 将func 的symbol 属性设为新建的 Funcsymbol，然后再作用域栈中声明。新建一个作用域 `Scope(ScopeKind.FORMAL)`，将其压栈，依次对参数，函数体进行类型检查，最后将作用域弹出。

`visitParameter` 的作用是为参数建立符号并存入作用域栈中，并将 param 的 symbol 属性设为 new_varsymbol。

`visitCall` 首先根据调用的函数名在作用域栈中寻找函数符号，如果没找到或者该符号不是函数则报错。然后比较 call 的调用参数与该函数符号中保存的参数列表长度是否一样，如果不一样则报错。接着对 call 参数列表中的每一个参数进行检查。最后比较 call 的参数列表与函数的参数列表中的每一项的类型是否一致，如果不一致则报错。

在 `visitBlock` 中，增加对上一层作用域是否是函数作用域的判断，如果是函数的作用域，则在当前作用域中加入函数作用域中的符号（参数符号）。

2.2 中间代码生成

由于现在 Program 中的函数不止一个，所以修改 Program 节点，让其children 变为 `List[Function]`，在 `functions` 函数中返回所有 `children` 的 `ident.value`。

在 utils/tac.py 中，新增 `Param`，`CALL` 类作为添加参数、调用函数的指令，并修改 `TACFunc` 类，增加 parameters 变量。

由于现在 Program 中的函数不止一个，所以修改 Program 节点，让其children 变为 `List[Function]`，在 `functions` 函数中返回所有 `children` 的 `ident.value`。

在 frontend/tacgen.py 中，修改 `TACFuncEmitter` 的函数 visitEnd，使得可以为函数添加函数参数；新增 `visitParam`，`visitCall` 函数，分别为增加一条添加参数的指令和增加一条函数调用的指令。

在 `TACGen` 类中，修改 `transform` 函数，在构造 `TACFuncEmitter` 时使用实际的参数个数，并在调用 visitEnd 时增加函数的参数列表 `param_list`。新增 `visitParameter` 和 `visitCall` 函数。在 `visitParameter` 中为参数符号绑定寄存器 (_Tx)，若有初值则设置初值；在 `visitCall` 中，首先完成参数列表部分 TAC 代码的生成，然后调用 `visitParam` 添加参数，最后设置返回值并使用 `mv.visitCall` 生成 TAC 代码中的函数调用语句。

3.后端

在 `backend/bruteregalloc.py` 中 `BruteRegAlloc` 类中，在 `for loc in bb.allSeq()` 的循环中判断指令类型是否是 `Call` 指令，如果是，则代表是函数调用指令，首先保存 caller_save 寄存器（判断寄存器是否被占用，以及当前是否还有用(liveIn)），将所有需要保存的寄存器保存到栈上，然后解绑。然后开始进行传参操作，使用 `Riscv.NativeStoreWord` 将所有参数保存到栈上。接着产生一条调用指令，并将返回值与 A0 绑定。最后恢复 caller_saved 变量，如果被保存的变量又被占用，则将新的值存到栈上并恢复旧的值。

在类 `RiscvSubroutineEmitter` 中，修改 `emitEnd` 函数，增加保存 `FP, SP,RA` 的内容。在 `RiscvAsmEmitter` 类 中的 `seletInstr` 中，在依次访问函数中的指令之前，先使用 `Riscv.LoadArgStack` 将参数从栈上读出来。

二、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”的中间表示。这种方式与 Riscv 的代码的相似性更高，在从三地址码到汇编代码的过程中可以每一条 PARAM 指令对应一条 Riscv 中的设置参数的指令，call 指令直接对应汇编代码中的 call 指令；但这种方法在处理分配寄存器的时候需要注意哪些寄存器还在 liveIn 中（正确设置 继承自 TACInstr 类中的 dsts 和 rscs），否则可能会导致寄存器分配的时候出错。

“一整条函数调用” 的方式与汇编代码的差异性较大，在翻译的时候需要处理更多的东西，但好处在于在一条指令中包含了全部需要用到的作为参数的寄存器，可以方便的知道参数的个数以及需要处理的内容。

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？

   若寄存器完全由 caller/callee 中的一方保存，则意味着 caller/callee 需要将全部寄存器进行压栈/退栈操作，需要对全部寄存器做两次 I/O 操作，会大大降低运行的效率。同时有些寄存器在函数调用中起传参作用，callee 没有必要保存；ra 起到保存返回地址的作用，在调用的过程中会被修改，由 callee 保存是不现实的；a0 保存返回值，如果由 callee save 则在返回前会被重新修改为先前的值，这不是理想的行为。所以引入 callee-saved 和 caller-saved 两类寄存器。

   在函数调用时，新的返回地址会被写入 ra 中，旧的 ra 会被覆盖，如果是 callee-saved，则无法回复原来的值，所以只能是 caller-saved 寄存器。