Stage 5 报告

程思翔 2021010761

1 实验内容

1.1 step 9

词法语法分析: 修改 `frontend/ast/tree.py`, 定义 `Call`, `Parameter`, `ParameterList`, `ExpressionList` 节点, 在 `frontend/parser/ply_parser.py` 中参考已有实现, 给出相关 `CFG` 文法.

语义分析: 修改 `frontend/typecheck/namer.py`, 完成语义分析. `visitFunction` 中首先需要检查函数是否重复声明, 同时为了将函数参数与函数体置于同一个作用域, 需要修改 `func.body.accept`, 避免访问 `block` 而新开一个作用域:

```
# frontend/typecheck/namer.py
def visitFunction(self, func: Function, ctx: ScopeStack) -> None:
    if GlobalScope.lookup(func.ident.value):
        raise DecafDeclConflictError(func.ident.value)
    symbol = FuncSymbol(func.ident.value, func.ret_t.type, GlobalScope)
    for param in func.params.children:
        symbol.addParaType(param.var_t)
    GlobalScope.declare(symbol)
    func.setattr('symbol', symbol)

    ctx.open()
    func.params.accept(self, ctx)
    for child in func.body.children:
        child.accept(self, ctx)
    ctx.close()
```

`visitCall` 中首先需要检查函数名是否被同级作用域声明的变量覆盖, 然后检查调用参数是否符合数目:

```
# frontend/typecheck/namer.py
def visitCall(self, call: Call, ctx: ScopeStack) -> None:
    if ctx.lookup(call.ident.value):
      raise DecafBadFuncCallError(call.ident.value)

    func = GlobalScope.lookup(call.ident.value)
    if not func or not func.isFunc:
        raise DecafUndefinedFuncError(call.ident.value)
    if func.parameterNum != len(call.args):
        raise DecafBadFuncCallError()

    call.ident.setattr('symbol', func)
    for arg in call.args:
        arg.accept(self, ctx)
```

中间代码生成: 使用传参和调用分离模式, 在 `utils/tac/tacop.py` 中添加两类指令类型定义:

```
# utils/tac/tacop.py
@unique
class InstrKind(Enum):
    # Function call.
    CALL = auto()
    # Function parameter.
    PARAM = auto()
```

在 `utils/tac/tacinstr.py` 中参照已有实现定义了 `Call` 与 `Param` 两种 TAC 指令类, 在 `utils/tac/tacgen.py` 中实现与之有关的 `Visitor` 模式方法:

```
# utils/tac/tacgen.py
class TACFuncEmitter(TACVisitor):
    def visitParam(self, value: Temp) -> None:
        self.func.add(Param(value))
```

```
5
6       def visitCall(self, label: Label) -> Temp:
7           temp = self.freshTemp()
8           self.func.add(Call(temp, label))
9           return temp
10
11  class TACGen(Visitor[TACFuncEmitter, None]):
12      def visitParameter(self, param: Parameter, mv:
    TACFuncEmitter) -> None:
13          # 分配虚拟寄存器
14          param.getattr('symbol').temp = mv.freshTemp()
15
16      def visitCall(self, call: Call, mv: TACFuncEmitter) -> None:
17          for arg in call.args.children:
18              arg.accept(self, mv)
19          for arg in call.args.children:
20              mv.visitParam(arg.getattr("val"))
21          call.setattr('val',
    mv.visitCall(FuncLabel(call.ident.value)))
```

目标代码生成: 在 `utils/riscv.py` 中定义了 `Call` 与 `Param` 类指令用于寄存器分配的标识, 同时仿照 `SPAdd` 指令实现了 `FPAdd` 指令用于保存和恢复栈帧.

```
1   # utils/riscv.py
2   class Call(TACInstr):
3       def __init__(self, target: Label) -> None:
4           super().__init__(InstrKind.CALL, [], [], target)
5           self.target = target
6       def __str__(self) -> str:
7           return "call " + super(FuncLabel, self.target).__str__()
8
9   class Param(TACInstr):
10      def __init__(self, src: Temp) -> None:
11          super().__init__(InstrKind.PARAM, [], [src], None)
```

`backend/subroutineemitter.py` 实现了 `emitReg`, `emitStoreParamToStack`, `emitRestoreStackPointer` 等方法用于保存参数到寄存器, 保存参数到栈中以及恢复栈指针.

在 `backend/riscv/riscvasmemitter.py` 中修改 `emitEnd` 打印 Riscv 指令的逻辑, 进入函数时将 `fp`, `ra` 寄存器存储到栈上, 保存 `callee_saved` 寄存器; 函数结束时, 从栈上恢复 `fp`, `ra` 寄存器和 `callee_saved` 寄存器:

```
# backend/riscv/riscvasmemitter.py
def emitEnd(self):
    ...
    # store RA, FP and CalleeSaved regs here
    self.printer.printInstr(Riscv.SPAdd(-self.nextLocalOffset))
    self.printer.printInstr(Riscv.NativeStoreWord(Riscv.RA,
Riscv.SP, 4 * len(Riscv.CalleeSaved)))
    self.printer.printInstr(Riscv.NativeStoreWord(Riscv.FP,
Riscv.SP, 4 * len(Riscv.CalleeSaved) + 4))
    self.printer.printInstr(Riscv.FPAdd(self.nextLocalOffset))

    ...
    # load RA, FP and CalleeSaved regs here
    self.printer.printInstr(Riscv.NativeLoadWord(Riscv.RA,
Riscv.SP, 4 * len(Riscv.CalleeSaved)))
    self.printer.printInstr(Riscv.NativeLoadWord(Riscv.FP,
Riscv.SP, 4 * len(Riscv.CalleeSaved) + 4))
    self.printer.printInstr(Riscv.SPAdd(self.nextLocalOffset))
```

在 `backend/reg/bruteregalloc.py` 中, 使用 `self.functionParams` 记录子函数所使用的参数, 使用 `self.callerSavedRegs` 保存 `caller_saved` 寄存器. 在函数开始先将实参绑定到寄存器中, 然后分析语句. `allocForLoc` 为每行指令分配寄存器时, 若为 `Param` 类型, 累计参数小于 8 时直接分配参数寄存器; 若为 `Call` 类型, 先保存 `caller_saved` 寄存器, 将多余参数插入栈中, 调用后恢复 `caller_saved` 寄存器.

2 思考题

2.1 step 9

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.

   答: 我更倾向采纳传参和调用分离, 这与实验文档给出的参考中间风格一致, 更接近目标语言.

   一整条函数调用:

   - **优势**:
     - 调用过程封装在一个指令中, 语义清晰, 可读性好.
     - 更接近高级语言, 有助于保留源代码结构和语义.
   - **劣势**:
     - 整条函数调用指令可能不够精确, 再特定架构下不能满足精细控制的需求.

   传参和调用分离:

   - **优势**:
     - 与实验文档给出的参考中间风格一致, 更接近目标语言.
   - **劣势**:
     - 需要增加 `Param` 指令描述参数传递, 提高了实现难度.
     - 中间表示可读性略差.

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: 如果寄存器都由 `caller` 保存, `callee` 可能只使用很少几个, 恢复寄存器开销过大; 如果寄存器都由 `callee` 保存, 函数调用结束时恢复所有用到的寄存器开销过大. 引入 `callee-saved` 和 `caller-saved` 两类寄存器, 编译器可以让 `callee` 保存函数调用后依然有效的值 (如返回地址), 让 `caller` 保存函数调用过程后不再使用的值 (如函数参数).

调用函数时, `ra` 中当前返回地址会被调用函数的返回地址替代, 因此需要在进入函数前保存好 `ra` 的值, 这应当由 `caller` 来完成.