Stage 6 报告

程思翔 2021010761

> *写在前面:*
>
> 这一个 Stage 的实现中对着前中后端的报错疯狂调 Bug, 基本实现过程就是 [构造测例] -> [看报错] -> [调代码], 好在终于是实现地较为完备, 痛并快乐着 (×).
>
> 报告或许会较长, 尽管尝试简化对实现思路的叙述, 最后还是保留了如下内容, 以便用尽可能简洁的描述, 将实现过程清晰地展示出来.

1 实验内容

1.1 step 10

词法语法分析: 修改 `frontend/ast/tree.py` 中 `Program` 节点的定义, 并添加 `globalVars()` 方法传递**全局变量**键值对. 在 `frontend/parser/ply_parser.py` 中参考已有实现, 给出相关 `CFG` 文法:

```
# frontend/ast/tree.py
class Program(ListNode[Union["Function", "Declaration"]]):
    ...
    def globalVars(self) -> dict[str, int]:
        return {decl.getattr('symbol').name:
decl.getattr('symbol').initValue for decl in self if
isinstance(decl, Declaration)}
    ...
```

语义分析: 只需修改 `frontend/typecheck/namer.py` 的 `visitDeclaration` 部分, 如果当前作用域为全局作用域, 修改 `symbol.isGlobal = True` 即可, 并为**全局变量**设置**初始值** `symbol.initValue`.

中间代码生成: 在 `utils/tac/tacinstr.py` 中参照已有实现, 添加全局变量地址加载、全局变量加载和存储的 TAC 指令:

```python
# utils/tac/tacinstr.py
class LoadAddress(TACInstr):
    # LOAD SYMBOL ADDRESS...
class LoadIntLiteral(TACInstr):
    # LOAD SYMBOL...
class StoreIntLiteral(TACInstr):
    # STORE SYMBOL...
```

在 `utils/tac/tacgen.py` 中 `TACFuncEmitter` 类实现与之有关的 `Visitor` 模式方法, 修改 `TACGen` 类的 `visitAssignment` 及 `visitIdentifier` 方法, 判断访问标识符和赋值操作是否对**全局变量**进行:

```python
# utils/tac/tacgen.py
class TACGen(Visitor[TACFuncEmitter, None]):
    def visitIdentifier(self, ident: Identifier, mv:
TACFuncEmitter) -> None:
        #! 标识符是全局变量
        if ident.getattr("symbol").isGlobal:
            ident.setattr('val',
mv.visitLoadIntLiteral(ident.getattr('symbol')))
        #! 否则
        else:
            ident.setattr('val', ident.getattr('symbol').temp)

    def visitAssignment(self, expr: Assignment, mv:
TACFuncEmitter) -> None:
        expr.rhs.accept(self, mv)
        #! 左值是全局变量
        if expr.lhs.getattr('symbol').isGlobal:
            mv.visitStoreIntLiteral(expr.lhs.getattr('symbol'),
expr.rhs.getattr("val"))
        #! 否则
        else:
            expr.lhs.accept(self, mv)
            expr.setattr(
```

```
                "val",
mv.visitAssignment(expr.lhs.getattr("symbol").temp,
expr.rhs.getattr("val"))
            )
```

目标代码生成: 在 `utils/riscv.py` 中参照已有实现, 添加**全局变量**地址获取、加载和存储的 RISC-V 指令:

```
# utils/riscv.py
class LoadAddress(TACInstr):
    # LOAD SYMBOL ADDRESS...
class LoadIntLiteral(TACInstr):
    # LOAD SYMBOL...
class StoreIntLiteral(TACInstr):
    # STORE SYMBOL...
```

在 `backend/riscv/riscvasmemitter.py` 中, `RiscvAsmEmitter` 对象初始化时打印全局变量至 `.data` 区, 并为 `RiscvInstrSelector` 实现相应全局变量指令访问方法:

1.2 step 11

词法语法分析: 修改 `frontend/ast/tree.py`, 通过 `Program` 的 `globalVars()` 方法返回**全局变量**与**全局数组**键值对; 修改 `Declaration` 节点定义, 添加数组维度声明 `init_dim`; 同时添加索引运算节点 `IndexExpr`:

```
# frontend/ast/tree.py
class IndexExpr(Expression):
    def __init__(self, base: Expression, index: Expression) ->
None:
        super().__init__("index_expr")
        self.base = base
        self.index = index
```

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`, `visitFunction` 中记录当前函数声明的**局部数组**; 在 `visitDeclaration` 中依据标识符类型进行初始化, 如果是数组则调用 `ArrayType.multidim`; 添加 `visitIndexExpr` 中递归实现数组索引访问.

```
# frontend/typecheck/namer.py
def visitDeclaration(self, decl: Declaration, ctx: ScopeStack) ->
None:
    ...
    if decl.init_dim:
        decl_type = ArrayType.multidim(decl.var_t.type, *
[dim.value for dim in decl.init_dim])
        symbol = VarSymbol(decl.ident.value, decl_type)
        self.arrays.append(symbol)
    else:
        decl_type = decl.var_t.type
        symbol = VarSymbol(decl.ident.value, decl_type)
    ...

def visitIndexExpr(self, expr: IndexExpr, ctx: ScopeStack) ->
None:
    if isinstance(expr.base, Identifier) and not
ctx.lookupOverStack(expr.base.value):
        raise DecafUndefinedVarError(expr.base.value)
    expr.base.accept(self, ctx)
    expr.index.accept(self, ctx)
    #! 根据 base 类型设置 expr 的类型
    if isinstance(expr.base, Identifier):
        expr.setattr('type',
expr.base.getattr('symbol').type.indexed)
    else:
        expr.setattr('type', expr.base.getattr('type').indexed)
```

通过对节点的 `setattr('type')` 与 `getattr('type')` 操作实现类型运算一致性检查, 一共有 `INT` 与 `ArrayType` 两种类型, 如果相应运算类型不一致, 则会抛出异常 `DecafBadReturnTypeError`.

中间代码生成: 在 `utils/tac/tacgen.py` 中 `TACFuncEmitter` 类实现了依地址的读写数组元素的方法:

```
# frontend/tacgen/tacgen.py
class TACFuncEmitter(TACVisitor):
    def visitLoadByAddress(self, addr: Temp):
        dst = self.freshTemp()
        self.func.add(LoadIntLiteral(dst, addr, 0))
        return dst

    def visitStoreByAddress(self, value: Temp, addr: Temp):
        self.func.add(StoreIntLiteral(value, addr, 0))
```

修改 `TACGen` 类的 `visitIndexExpr`, `visitAssignment` 及 `visitIdentifier` 方法, 设置数组索引表达式的地址, 并判断访问标识符和赋值操作是否对数组进行:

```
# utils/tac/tacgen.py
class TACGen(Visitor[TACFuncEmitter, None]):
    def visitIdentifier(self, ident: Identifier, mv:
TACFuncEmitter) -> None:
        # 数组类型 -> 设置数组地址
        if isinstance(ident.getattr('symbol').type, ArrayType):
            ident.setattr('addr',
mv.visitLoadAddress(ident.getattr('symbol')))
        ...

    def visitIndexExpr(self, expr: IndexExpr, mv: TACFuncEmitter)
-> None:
        expr.base.setattr('slice', True)
        expr.base.accept(self, mv)
        expr.index.accept(self, mv)
```

```
        #! 递归计算当前索引的偏移量
        addr = mv.visitLoad(expr.getattr('type').size)
        mv.visitBinarySelf(tacop.TacBinaryOp.MUL, addr,
expr.index.getattr('val'))
        mv.visitBinarySelf(tacop.TacBinaryOp.ADD, addr,
expr.base.getattr('addr'))
        expr.setattr('addr', addr)
        #! 递归完毕，通过地址获得数组元素值
        #! `slice` 属性表示为数组切片，无需获取数据
        #! 保证递归结束只有完整的索引表达式设置了返回值
        if not expr.getattr('slice'):
            expr.setattr('val', mv.visitLoadByAddress(addr))

    def visitAssignment(self, expr: Assignment, mv:
TACFuncEmitter) -> None:
        # 索引类型 -> 访问数组地址
        if isinstance(expr.lhs, IndexExpr):
            expr.lhs.setattr('slice', True)
            expr.lhs.accept(self, mv)
            mv.visitStoreByAddress(expr.rhs.getattr('val'),
expr.lhs.getattr('addr'))
            expr.setattr('val', expr.rhs.getattr("val"))
        ...
```

目标代码生成: 在 `utils/riscv.py` 中实现了 `addi` 的 RISC-V 指令, 用于加载通过 `TACFunc.arrays` 传递并保存在栈的**局部数组** (仍通过 `Program.globalVars` 加载**全局数组**地址):

```python
# utils/riscv.py
class ImmAdd(TACInstr):
    def __init__(self, dst: Temp, src: Temp, value: int):
        super().__init__(InstrKind.SEQ, [dst], [src], None)
        self.value = value

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

进入一个函数前, 提前将其中的**局部数组压栈**, 在 `backend/subroutineinfo.py` 中计算得到各数组偏移量及占用栈帧大小:

```python
# backend/subroutineinfo.py
class SubroutineInfo:
    def __init__(self, funcLabel: FuncLabel, numArgs: int, arrays:
Dict[str, VarSymbol]) -> None:
        self.offsets: Dict[str, int] = {}
        self.size = 0

        for name, symbol in self.arrays.items():
            self.offsets[name] = self.size
            self.size += symbol.type.size
```

在 `backend/riscv/riscvasmemitter.py` 中, `RiscvAsmEmitter` 对象初始化时打印**全局数组**至 `.bss` 区, `RiscvInstrSelector` 中通过偏移量实现**局部数组**访问:

```python
# backend/riscv/riscvasmemitter.py
class RiscvAsmEmitter(AsmEmitter):
    class RiscvInstrSelector(TACVisitor):
        def visitLoadAddress(self, instr: LoadAddress) -> None:
            if instr.symbol.isGlobal:
                ...
            else:
                self.seq.append(Riscv.ImmAdd(instr.dsts[0],
Riscv.SP, self.info.offsets[instr.symbol.name]))
```

此时的 `self.nextLocalOffset` 及保存 `RA`, `FP` 和 `Callee-saved` 寄存器时需要额外加上 `self.info.size` 由**局部数组**占用的栈帧大小.

1.3 step 12

代码流读入: 在 `main.py` 中给出 `memset` 函数 `fill_csx` 的**字符串表示**, 命名方式是为了防止可能的函数重名 (虽然根据给出的测例, 没有名为 `fill_csx` 的函数, 但更好的方法是在语法分析后动态命名). 随后直接将代码加入输入的 `code` 前:

```python
# main.py
memsetFunc = r"""
int fill_csx(int array[], int cnt) {
    for (int i = 0; i < cnt; i = i + 1) {
        array[i] = 0;
    }
    return 0;
}
"""

def step_parse(args: argparse.Namespace):
    code = memsetFunc + "\n" + readCode(args.input)
    r: Program = parser.parse(code, lexer=lexer)
    ...
```

词法语法分析: 修改 `frontend/ast/tree.py`, 修改 `Parameter` 节点定义, 添加维度声明 `init_dim` 标识**数组传参**; 同时添加**数组初始化列表**节点 `InitList`.

```python
# frontend/ast/tree.py
class InitList(Node):
    def __init__(self, init_list: List[IntLiteral]):
        super().__init__("init_list")
        self.init_list = init_list
        self.value = [item.value for item in init_list]
```

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`, 在 `visitFunction` 中记录函数声明中进行传参的**参数数组**; 在 `visitParameter` 中依据参数类型进行标识符 `symbol` 生成, 如果是数组则调用 `ArrayType.multidim` 生成相应的数据类型.

```python
# frontend/typecheck/namer.py
def visitParameter(self, param: Parameter, ctx: ScopeStack) ->
None:
    ...
    #! 为数组参数, 检查并生成相应的数据类型
    if param.init_dim:
        for index, dim in enumerate(param.init_dim):
            if index == 0 and dim is NULL:
                continue
            if dim.value <= 0:
                raise DecafBadArraySizeError()
        decl_type = ArrayType.multidim(param.var_t.type, *
[dim.value if dim else None for dim in param.init_dim])
        symbol = VarSymbol(param.ident.value, decl_type)
    ...
```

中间代码生成: 在 `utils/tac/tacgen.py` 中修改 `TACGen` 类的 `visitIdentifier` 方法, 对**全局数组与局部数组**直接加载数组地址, 对**参数数组**加载相应虚拟寄存器; 修改 `visitDeclaration` 方法, 若为**带初始化列表**的**局部数组**声明, 先调用 `fill_csx` 函数进行内存清零, 然后逐一进行元素初始化:

```python
# utils/tac/tacgen.py
```

```
class TACGen(Visitor[TACFuncEmitter, None]):
    def visitIdentifier(self, ident: Identifier, mv:
TACFuncEmitter) -> None:
        if isinstance(ident.getattr('symbol').type, ArrayType):
            # 全局数组与局部数组
            if ident.getattr("symbol").isGlobal or
ident.getattr('symbol') not in mv.func.p_arrays.values():
                ident.setattr('addr',
mv.visitLoadAddress(ident.getattr('symbol')))
                ident.setattr('val', ident.getattr('addr'))
            # 参数数组
            else:
                ident.setattr('addr',
ident.getattr('symbol').temp)
                ident.setattr('val', ident.getattr('addr'))

    def visitDeclaration(self, decl: Declaration, mv:
TACFuncEmitter) -> None:
        if isinstance(decl.init_expr, InitList):
            #! 调用 `fill_csx` 函数进行初始化
            symbol = decl.getattr("symbol")
            addr = mv.visitLoadAddress(symbol)
            #! size 为 4 -> int 字长
            size = symbol.type.full_indexed.size
            interval = mv.visitLoad(size)
            mv.visitParam(addr)
            mv.visitParam(mv.visitLoad(symbol.type.size // size))
            mv.visitCall(FuncLabel("fill_csx"))
            #! 依次将初始化列表中的值存入数组中
            for value in decl.init_expr.value:
                mv.visitStoreByAddress(mv.visitLoad(value), addr)
                mv.visitBinarySelf(tacop.TacBinaryOp.ADD, addr,
interval)
```

目标代码生成: 在 `backend/riscv/riscvasmemitter.py` 中, `RiscvAsmEmitter` 对象初始化时打印**带初始化列表的全局数组**至 `.data` 区.

由于传参和调用分离, 对于**参数数组**的使用与**普通函数参数**并无区别, 这一步沿用 **step 9** 的实现即可, 后端无需进行其他工作.

2 思考题

2.1 step 10

1. 写出 `la v0, a` 这一 RiscV 伪指令可能会被转换成的指令组合 (两种即可).

   答: 查阅 The RISC-V Instruction Set Manual.

   - non-PIC 可能转换为:

     ```
     auipc v0, delta[31:12] + delta[11]
     addi v0, v0, delta[11:0]
     ```

     其中 `delta` 为 `a` 相对 `PC` 的偏移量, 因为地址在编译时已知, 所以使用 `addi` 加载.

   - PIC 可能转换为:

     ```
     auipc v0, delta[31:12] + delta[11]
     lw v0, v0, delta[11:0]
     ```

     其中 `delta` 为 `a` 在 GOT 中的地址相对 `PC` 的偏移量, 因为 GOT 中的地址可能在运行时进行重定位, 因此需要使用 `lw` 从内存中加载地址.

   - 两种指令组合都存在 `+ delta[11]`, 这保证了在 `delta[11] = 1` 时, 低 12 位经过符号扩展, 最终也能够得到正确的结果.

2.2 step 11

1. C 语言规范规定, 允许局部变量是可变长度的数组 (VLA), 在我们的实验中为了简化, 选择不支持它. 请简要回答, 如果支持一维的可变长度的数组 (类似 `int n = 5; int a[n];`, 但不允许类似 `int n = ...; int m = ...; int a[n][m];`), 而且要求数组仍保存在栈上 (不允许用堆上动态内存申请), 应该在现有的实现基础上做出那些改动?

> 提示: 不能再在进入函数时统一给局部变量分配内存, 离开时统一释放内存.

答: 不能在进入函数时为 VLA 分配内存. 由于 VLA 的大小在编译期确定, 运行到声明 VLA 时, 将当前 `SP` 和 VLA 大小 `size` 保存到栈上 `-size(SP)` 处, 并移动 `SP`, 此时先前存储好的 `SP` 和 `size` 恰位于栈顶; 访问 VLA 的元素时, 计算偏移量 VLA 元素地址; 当 VLA 离开作用域时, 恢复 SP 即可.

2.3 step 12

1. 作为函数参数的数组类型第一维可以为空. 事实上, 在 C/C++ 中即使标明了第一维的大小, 类型检查依然会当作第一维是空的情况处理. 如何理解这一设计?

   答: C/C++ 数组传参时, 数组名对应首元素地址; 函数无需为数组分配内存, 只需要通过首地址和偏移量即可访问到任意数组元素, 根据数组索引计算偏移量不会用到第一维的大小, 因此编译器会将数组参数的第一维视为空.