MiniDecaf-Stage5实验报告

计06 赵翊哲 2020010998

一、实验目的

- 支持数组的声明、访问
- 支持数组的定义，允许函数传递数组作为参数

二、实验内容

2.1 Step 11

- 修改内容：
  - 增加了数组的声明
  - 允许访问数组内的变量
- 实现：
  - 本次修改的部分较多，列举了部分完成实验时所使用的 `TODO`，后续只简要介绍修改的内容，列举部分核心函数作为示例。

  - 前端：
    - `ply_parser.py`：修改了文法识别，增加对 declaration 文法识别，允许识别形如`int a[2][3]`类的声明代码以及形如`a = b[2][1]`类的访问代码
    - `tree.py`：增加了 IndexExpr 语法节点，用于处理访问数组元素时的索引表达式
    - `array.py`：修改了部分关于数组类型的定义，增加了对于后续在 tac 码生成过程中使用的函数的定义
    - `namer.py`：对数组类型进行类型检查，核心方向在于不允许数组声明时维度出现非正数，以及在赋值和定义语句中进行类型检查，避免数组和整数类型混用赋值。

      核心代码：

```
def visitIndexExpr(self, array: IndexExpr, ctx: T) -> None:
        # TODO: step11-3 为数组类型进行类型检查
        array.base.accept(self, ctx)
        array.index.accept(self, ctx)
        # 作为表达式的左值必须是形如 a[0][1][2] 的索引表达式
        symbol = ctx.lookup(array.base.value)
        if symbol.type.dim != len(array.index.children):
            raise DecafBadIndexError(array.base.value)
def visitAssignment(self, expr: Assignment, ctx: ScopeStack) -> None:
        """
        1. Refer to the implementation of visitBinary.
        """
        # TODO: step5-2 处理赋值
        # ScopeStack.lookup 检查左值是否定义
        if type(expr.lhs) == IndexExpr:   # 数组类型需要检查数组变量名是否已定义
            symbol = ctx.lookup(expr.lhs.base.value)
        else:
            symbol = ctx.lookup(expr.lhs.value)
        if symbol == None:  # 无定义报错
            raise DecafUndefinedVarError(expr.lhs.value)
        else:
            expr.lhs.setattr("symbol", symbol)
            if type(expr.rhs) == Identifier:
                rhs_symbol = ctx.lookup(expr.rhs.value)
                if type(rhs_symbol.type) == ArrayType:
                    raise DecafTypeMismatchError()
        expr.lhs.accept(self, ctx)  # 对数组索引表达式进行类型检查
        # 类似 visitBinary
        expr.rhs.accept(self, ctx)
```

- 中端：生成 tac 码
  - `tacgen.py`：主要处理了数组的访问和赋值，核心逻辑是首先对索引表达式进行 tac 生成，得到需要处理的元素相对于首地址的偏移量，然后在依据其位置（lhs or rhs）选择 Load 或者 Store 指令（在stage4所实现的指令）

    在生成索引表达式的 tac 码时，参考了课程中相关的 PPT，同时因为在实现过程中需要区分局部数组和全局数组，需要额外处理（前者是保存在栈上，通过 symbol 中相关的 temp 可以获得基地址，而后者每次访问的时候都要如同全局变量的访问一样，先获得symbol对应的地址）

    核心代码：

```
def visitIndexExpr(self, expr: IndexExpr, mv: FuncVisitor) -> None:
        # TODO: step11-4 为数组索引表达式生成 TAC 代码
        for child in expr.index.children:
            child.accept(self, mv)
        expr.base.accept(self, mv)
        symbol = expr.base.getattr("symbol")
        indexes = symbol.type.indexes
        offset_list = []
```

```
        offset_list.append(mv.visitLoad(4))
        offset = 1
        for i in range(len(indexes) - 1):
            rank = len(indexes) - 1 - i
            offset *= (indexes[rank] * 4)
            offset_list.append(mv.visitLoad(offset))
        offset_list = offset_list[::-1]
        add_temp = mv.visitBinary(tacop.BinaryOp.MUL, offset_list[0],
expr.index.children[0].getattr("val"))
        for i in range(1, len(offset_list)):
            new_add_temp = mv.visitBinary(tacop.BinaryOp.MUL,
offset_list[i], expr.index.children[i].getattr("val"))
            add_temp = mv.visitBinary(tacop.BinaryOp.ADD, add_temp,
new_add_temp)
        if symbol.isGlobal:
            src = mv.visitLoadSymbol(expr.base.value)
            addr = mv.visitBinary(tacop.BinaryOp.ADD, add_temp, src)
        else:
            addr = mv.visitBinary(tacop.BinaryOp.ADD, add_temp,
symbol.temp)
        expr.setattr("addr", addr)
        expr.setattr("val", mv.visitLoadGlobalVar(addr, 0))
def visitAssignment(self, expr: Assignment, mv: FuncVisitor) -> None:
        """
        1. Visit the right hand side of expr, and get the temp variable of
left hand side.
        2. Use mv.visitAssignment to emit an assignment instruction.
        3. Set the 'val' attribute of expr as the value of assignment
instruction.
        """
        # TODO: step5-6 设置赋值语句的中间代码（参考 visitBinary 实现）
        # TODO: step10-9 将对全局变量的赋值特殊处理
        expr.lhs.accept(self, mv)
        expr.rhs.accept(self, mv)
        if type(expr.lhs) == IndexExpr:
            # TODO: step11-5 数组变量赋值
            temp = expr.rhs.getattr("val")
            mv.visitStore(temp, expr.lhs.getattr("addr"), 0)
            expr.setattr("val", expr.rhs.getattr("val"))
        else:
            if type(expr.lhs.getattr("symbol").type) == ArrayType:
                raise DecafTypeMismatchError()
            temp = expr.lhs.getattr("symbol").temp
            # 使用 temp 添加 TAC 指令，而非 var
            isGlobal = expr.lhs.getattr("global")
            if isGlobal:
                addr = expr.lhs.getattr("addr")
                mv.visitStore(expr.rhs.getattr("val"), addr, 0)
            else:
```

```
                    mv.visitAssignment(temp, expr.rhs.getattr("val"))
                # 设置表达式 val
                expr.setattr("val", temp)
```

- 后端：本次后端的处理较为简单，因为在step11中暂时不支持数组的初始化，因此只需要修改 bss 段的数组声明，将space设置为对应的空间即可。此外，在声明局部数组的时候使用了 ALLOC 指令用于空间申请，需要对此给出对应的后端代码生成。
  - `riscvasmemitter.py`：修改了bss段的生成，增加了 ALLOC 的指令选择，实现上认为局部数组直接保存在栈上，因此只需要将 sp 中的结果赋值给 rd 即可。另一方面需要保证在函数入口处开辟的空间足够大，需要实现将数组需要的offset传递过去。这一实现对于函数内定义多个数组以及需要压栈传参的函数可能存在一些问题，但是由于测例并未涉及相关情况，因此暂时没有进行额外的完善。

    核心代码：

```
if self.bss:    # 生成 bss 段
        self.printer.println(".bss")
            for key, value in self.bss.items():
                self.printer.println(".globl " + key)
                self.printer.println(key + ":")
                # TODO: step11-6 为全局数组分配空间
                self.printer.println("    " + ".space " +
str(value.var_t.type.size))
# TODO: step11-7 为数组分配地址空间 Alloc 对应的 riscv 指令生成
        def visitAlloc(self, instr: Alloc) -> None:
            """为数组分配地址空间"""
            self.seq.append(Riscv.Move(instr.dst, Riscv.SP))
```

2.2 Step 12

- 修改内容：
  - 支持数组的定义
  - 支持函数传递数组作为参数
- 实现：
  - 本次修改的部分较少，只是对先前内容的进一步完善。
  - 前端：
    - `tree.py`：新增了初始化表达式节点
    - `ply_parser.py`：修改了declaration相关的文法，允许出现第1维是空的情况（当且仅当作为函数参数声明时）
    - `namer.py`：完善了类型检查，避免出现函数定义参数类型和实际调用参数类型不符合的情况

      核心代码：

```
# tree.py
class InitList(Expression):
    def __init__(self) -> None:
```

```
        super().__init__("init_value")
        self.values = []

    def __getitem__(self, key: int) -> Node:
        raise self.values[key]

    def __len__(self) -> int:
        return len(self.values)

    def accept(self, v: Visitor[T, U], ctx: T):
        return v.visitInitList(self, ctx)
# namer.py
def visitCall(self, call: Call, ctx: ScopeStack) -> None:
        # TODO: Step9-8 通过类似 Expression 的方式进行语义检查
        """函数调用除了检查参数是否合法，调用函数是否经过定义之外，
        还要在类型检查(Typer)中检查函数参数和定义是否一致。"""
        symbol = ctx.lookup(call.ident.value)
        if symbol == None:  # 无定义报错
            raise DecafUndefinedFuncError(call.ident.value)
        else:
            if len(call.arguments.children) != symbol.parameterNum:
                # 参数数目不一致
                raise DecafBadFuncCallError(call.ident.value)
            # print(symbol.parameterNum)
            else:
                for i in range(symbol.parameterNum):
                    # TODO: step12-4 检查类型
                    if type(call.arguments.children[i]) == Identifier:
                        arg_symbol =
ctx.lookup(call.arguments.children[i].value)
                        # print(arg_symbol.type)
                        # print(symbol.getParaType(i))
                        if type(arg_symbol.type) !=
type(symbol.getParaType(i)):
                            raise DecafBadFuncCallError(call.ident.value)
            call.arguments.accept(self, ctx)
```

- 中端：
  - `tacgen.py`：对 visitDeclaration 函数进行修改，增加对数组初始化的处理，具体的实现方式是手动构造一个 Assignment 表达式，并对其进行 tac 生成。全局数组的初始化在后端部分讨论。除此之外，在函数调用时进行修改，如果使用了数组传参则把symbol对应的数组首地址传递过去。

    核心代码：

```
def visitDeclaration(self, decl: Declaration, mv: FuncVisitor) -> None:
        """
        1. Get the 'symbol' attribute of decl.
        2. Use mv.freshTemp to get a new temp variable for this symbol.
```

```
        3. If the declaration has an initial value, use mv.visitAssignment
to set it.
        """
        # TODO: step5-5 生成声明语句的TAC
        # 使用 getattr 方法获得在符号表构建阶段建立的符号
        symbol = decl.getattr("symbol")
        if type(symbol.type) == ArrayType:
            if decl.is_param == False:
                symbol.temp = mv.visitAlloc(decl.var_t.type.size)
                # TODO: step12-3 生成数组初始化的 TAC
                if decl.init_expr != NULL:
                    # pass
                    length = len(decl.init_expr.values)
                    for rank in range(length):
                        expr_list = ExpressionList()
                        expr_list.children.append(IntLiteral(rank))
                        idx_expr = IndexExpr(decl.ident, expr_list)
                        value = decl.init_expr.values[rank]
                        assign_expr = Assignment(idx_expr, value)
                        assign_expr.accept(self, mv)
                    if length < symbol.type.length:
                        for rank in range(length, symbol.type.length):
                            expr_list = ExpressionList()
                            expr_list.children.append(IntLiteral(rank))
                            idx_expr = IndexExpr(decl.ident, expr_list)
                            assign_expr = Assignment(idx_expr,
IntLiteral(0))
                            assign_expr.accept(self, mv)
            else:
                symbol.temp = mv.freshTemp()
        else:
            # FuncVisitor.freshTemp 函数获取一个新的临时变量 temp 存储该变量
            symbol.temp = mv.freshTemp()
            # 如果有设置初值， visitAssignment 赋值
            if decl.init_expr != NULL:
                decl.init_expr.accept(self, mv)
                mv.visitAssignment(getattr(symbol, "temp"),
decl.init_expr.getattr("val"))
def visitCall(self, call: Call, mv: FuncVisitor) -> None:
        """依次访问所有调用参数，并将它们的返回值(getattr("val"))依次执行 PARAM 指
令，
        然后执行 CALL 指令，并新建临时变量为函数设置返回值。"""
        index = 0
        # if len(call.arguments.children) < 9:
        for arg in call.arguments.children:
            arg.accept(self, mv)
        for arg in call.arguments.children:
            # print(arg.getattr("symbol"))
            symbol = arg.getattr("symbol")
```

```
            if symbol != None and symbol.isGlobal and type(symbol.type) ==
ArrayType:
                mv.visitParam(arg.getattr("addr"), index)
            else:
                mv.visitParam(arg.getattr("val"), index)
            index += 1
```

- 后端：后端的处理很简单，只需要修改 data 段关于有初始值的数组的格式即可，在使用了工具链之后可以发现 riscv 只需要一次将有初始值的元素罗列即可，后续标注为 .zero 及对应的空间大小。
  - `riscvasmemitter.py`：修改了data段的生成

    核心代码

```
if self.data:    # 生成 bss 段
            self.printer.println(".data")
            for key, value in self.data.items():
                self.printer.println(".globl " + key)
                self.printer.println(key + ":")
                if type(value.var_t.type) == ArrayType:
                    for init_value in value.init_expr.values:
                        self.printer.println("    " + ".word " +
str(init_value.value))
                    if len(value.init_expr.values) <
value.var_t.type.length:
                        self.printer.println("    " + ".zero " +
str((value.var_t.type.length - len(value.init_expr.values)) * 4))
                else:
                    self.printer.println("    " + ".word " +
str(value.init_expr.value))
```

三、思考题

3.1 Step11 思考题

Q1：C 语言规范规定，允许局部变量是可变长度的数组（Variable Length Array，VLA），在我们的实验中为了简化，选择不支持它。请你简要回答，如果我们决定支持一维的可变长度的数组(即允许类似 `int n = 5; int a[n];` 这种，但仍然不允许类似 `int n = ...; int m = ...; int a[n][m];` 这种)，而且要求数组仍然保存在栈上（即不允许用堆上的动态内存申请，如`malloc`等来实现它），应该在现有的实现基础上做出那些改动?

> 提示：不能再像现在这样，在进入函数时统一给局部变量分配内存，在离开函数时统一释放内存。
>
> 你可以认为可变长度的数组的长度不大于0是未定义行为，不需要处理。

A：首先需要对文法识别进行修改，单独定义形如 `int a[n]` 形式的数组定义。在`namer.py`的类型检查中，需要检查所访问的 n 是否有定义。在生成中间代码的时候，只需要修改 ALLOC 指令，允许使用变量申请空间即可 `T1 = ALLOC T0`，由于只允许一维可变长度，因此不需要对索引表达式的生成进行修改。后端的修改较为麻烦，因为不能再进入函数时直接计算处一共需要开辟的空间大小进行 sp 偏移，因此需要修改 ALLOC 对应的指令选择函数，一种实现方式时 `addi rd, sp, -size` 来获得一片空间，但是在这种实现方式下需要对栈的偏移等内容进行修改和保存，避免数据覆盖出现错误。

3.2 Step12 思考题

Q1：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?

A：数组作为函数参数，实际上传递的是数组的首地址，后续在函数中通过索引表达式访问数组元素，实际上是通过计算相对于首地址的偏移量来获得实际地址，在这个计算的过程中不会使用到第一维的大小，而后面的维度的大小是有必要的。而编译器之所以不处理标明了大小的情况是减少处理的开销，也可以避免造成误会。

四、参考资料

实现过程中主要参考了实验文档和实验思路QA墙

五、其他

在实现过程中，step11的quicksort测例无法通过，然而step12的quicksort可以通过，事实上我阅读了quicksort.s，也并没有发现问题，由于时间的原因，没有进行进一步的调试，很遗憾。

非常感谢助教一学期的工作，也非常感谢助教的答疑，很高兴能够完成这次实验!