MiniDecaf Stage 5: 数组

> My-laniaKeA

实验内容

Step 11: 数组

词法分析和语法分析

定义 `IndexExpr` AST 节点，进行索引运算。设声明了数组 `int a[2][3]`，那么对于索引运算表达式 `a[i][i+1]`，各成员变量的值如下所示：

类型	成员变量名称	含义	举例
`std::string`	`base_name`	数组名称	`a`
`ExprList *`	`idx_list`	每个索引表达式	`i`, `i+1`
`DimList *`	`dim`	数组每一维的大小	`2`, `3`

增加相应的词法规则。

```
"["           { return yy::parser::make_LBRACK  (loc);   }
"]"           { return yy::parser::make_RBRACK  (loc);   }
```

更新语法规则，支持数组。仅列出新增的语法规则。

```
DeclrStmt   : Type IDENTIFIER DimList SEMICOLON
                { $$ = new ast::VarDecl($2, $1, $3, POS(@2)); }
            | Type IDENTIFIER DimList ASSIGN Expr SEMICOLON
                { $$ = new ast::VarDecl($2, $1, $3, $5, POS(@2)); }
            ;
AssignExpr  : IDENTIFIER IndexList ASSIGN Expr
                { $$ = new ast::AssignExpr($1, $2, $4, POS(@1)); }
            ;
Postfix     : IDENTIFIER IndexList
                { $$ = new ast::LvalueExpr($1, $2, POS(@1)); }
            ;
IndexList   : LBRACK Expr RBRACK IndexList
                { $4->append($2);
                  $$ = $4; }
            | LBRACK Expr RBRACK
                { $$ = new ast::ExprList();
                  $$->append($2); }
            ;
DimList   : LBRACK ICONST RBRACK DimList
                { $$ = $4;
                  $$->append($2); }
```

```
            | LBRACK ICONST RBRACK
                { $$ = new ast::DimList();
                  $$->append($2); }
```

语义分析

符号表构建

在 `src/translation/translation.cpp` 中，处理 `VarDecl` 时，如果遇到数组声明，则根据 `dim` 计算数组大小，并判断是否为零长度数组。

```
void SemPass1::visit(ast::VarDecl *vdecl) {
    // ...
    if (vdecl->dim != NULL) { // is an array
        int length = 1;
        for (int d : *(vdecl->dim)){
            length *= d;
        }
        if (length == 0){
            issue(vdecl->getLocation(), new ZeroLengthedArrayError());
        }
        vdecl->type->ATTR(type) = new ArrayType(vdecl->type->ATTR(type),
length);
    }
    // ...
}
```

类型检查

在 `src/translation/type_check.cpp` 中，针对数组进行类型检查。无论读写，最终都会进入到 `SemPass2::visit(ast::VarRef *ref)` 函数处理。在前面 step 的基础上，检查是否对非数组的变量进行索引运算，并设定相应的属性值。

```
void SemPass2::visit(ast::VarRef *ref) {
    Symbol *v = scopes->lookup(ref->var, ref->getLocation());
    if (NULL == v) {
        // SymbolNotFoundError
    } else if (!v->isVariable()) {
        // NotVariableError
    } else if (!v->getType()->isArrayType()) {
        if (ref->idx_expr != NULL){
            issue(ref->getLocation(), new NotArrayError());
            goto issue_error_type;
        }
        ref->ATTR(sym) = (Variable *)v;
        ref->ATTR(type) = v->getType();
        if (((Variable *)v)->isLocalVar()) {
            ref->ATTR(lv_kind) = ast::Lvalue::SIMPLE_VAR;
        }
    } else {
        ref->ATTR(sym) = (Variable *)v;
        if (ref->idx_expr == NULL) {
            ref->ATTR(type) = v->getType();
            ref->ATTR(lv_kind) = ast::Lvalue::SIMPLE_VAR;
        }
```

```
        else {
            ref->idx_expr->accept(this);
            ref->ATTR(type) = ((ArrayType *)v->getType())->getElementType();
            ref->ATTR(lv_kind) = ast::Lvalue::ARRAY_ELE;
            ref->idx_expr->dim = ((Variable *)v)->getDimList();
            mind_assert(ref->idx_expr->dim != NULL);
        }
    }
    return;
}
```

中间代码生成

设计如下 TAC，为局部变量数组在栈上分配空间。

TAC	参数	含义
`ALLOC`	空间大小 `size`	分配 `size` 字节的内存，并返回内存首地址

由于全局数组的大小变化，所以需要引入新的 `Pieces` 处理全局变量的信息。首先在 `tac.hpp` 中增加 `Piece` 的成员，模仿 `FUNCTY` 实现 `GLOBAL` 类的 `Piece`。在 `trans_helper.cpp` 中通过 `genGlobalVarible` 函数来生成这些 `Piece`。

```
typedef struct GlobalObject {
    std::string name;
    int value;
    int size;
} * GlobalVar;

struct Piece {
    // kind of this Piece node
    enum {
        FUNCTY,
        GLOBAL,
    } kind;
    // data of this Piece node
    union {
        Functy functy;
        GlobalVar globalVar;
    } as;
};
```

```
/* Generates a GlobalVar object.
 */
void TransHelper::genGlobalVarible(std::string name, int value, int size) {
    ptail = ptail->next = new Piece();
    ptail->kind = Piece::GLOBAL;
    ptail->as.globalVar = new GlobalObject();
    ptail->as.globalVar->name = name;
    ptail->as.globalVar->value = value;
    ptail->as.globalVar->size = size;
}
```

因此，对于变量声明，可以按照是否为全局变量、是否为数组分成四种情况处理。不是数组的两种情况不再赘述。对于全局数组，通过 `genGlobalVarible` 生成一个 `Piece`，注意空间大小设置。对于局部数组，则先用 `genAlloc` 分配空间。

```
void Translation::visit(ast::VarDecl *decl) {
    Variable *var = decl->ATTR(sym);
     if (decl->ATTR(sym)->isGlobalVar()) {
        if(decl->init == NULL){
            tr->genGlobalVarible(decl->name, 0, decl->type->ATTR(type)-
>getSize());
        }
        else {
            assert(decl->init->getKind() == ast::ASTNode::INT_CONST);
            tr->genGlobalVarible(decl->name, ((ast::IntConst *)(decl->init))-
>value, decl->type->ATTR(type)->getSize());
        }
    }
    else {
        if (decl->type->ATTR(type)->isArrayType()) {
            Temp addr = tr->genAlloc(decl->type->ATTR(type)->getSize());
            var->attachTemp(addr);
        }
        else
            var->attachTemp(tr->getNewTempI4());

        if (NULL != decl->init){
            decl->init->accept(this);
            tr->genAssign(var->getTemp(), decl->init->ATTR(val));
        }
    }
}
```

另外需要注意的一点是偏移量的计算。对于多维数组，需要结合数组声明时每一维的大小，展开为一维数组，算出索引计算相对于数组首地址的偏移量。具体通过 `Translation::visit(ast::IndexExpr *e)` 函数实现。

```
void Translation::visit(ast::IndexExpr *e){
    mind_assert(e->idx_list->length() == e->dim->length());
    auto idx_iter = e->idx_list->begin();
    auto dim_iter  = e->dim->begin();
    int size_factor = 1;
    Temp offset = tr->genLoadImm4(0);
    for(size_t i = 0; i < e->idx_list->length(); ++idx_iter, ++dim_iter, ++i){
        (*idx_iter)->accept(this);
        Temp index = (*idx_iter)->ATTR(val);
        Temp factor = tr->genLoadImm4(size_factor);
        index = tr->genMul(index, factor);
        offset = tr->genAdd(offset, index);
        size_factor *= (*dim_iter);
    }
    Temp t = tr->genLoadImm4(4);
    offset = tr->genMul(offset, t);
    e->ATTR(val) = offset;
}
```

此外，修改 `Translation::visit(ast::LvalueExpr *e)` 和 `Translation::visit(ast::AssignExpr *s)` 函数，对数组中的元素进行读写。得到全局变量的 symbol 或数组首地址后，需要加上偏移量。

```
void Translation::visit(ast::LvalueExpr *e) {
    ((ast::VarRef*)e->lvalue)->accept(this);
    ast::VarRef *ref = (ast::VarRef *)e->lvalue;
    if (ref->ATTR(sym)->isGlobalVar()){
        Temp addr = tr->genLoadSym(ref->ATTR(sym)->getName());
        if(ref->ATTR(lv_kind) == ast::Lvalue::ARRAY_ELE)
            addr = tr->genAdd(addr, ref->idx_expr->ATTR(val));
        e->ATTR(val) = tr->genLoad(addr, 0);
    }
    else {
        if (ref->ATTR(lv_kind) == ast::Lvalue::ARRAY_ELE) {
            Temp temp = tr->genAdd(ref->ATTR(sym)->getTemp(), ref->idx_expr-
>ATTR(val));
            e->ATTR(val) = tr->genLoad(temp, 0);
        }
        else
            // ...
    }
}
void Translation::visit(ast::AssignExpr *s) {
    s->left->accept(this);
    s->e->accept(this);
    ast::VarRef *ref = (ast::VarRef *)(s->left->lvalue);
    if(ref->ATTR(sym)->isGlobalVar()){
        Temp addr = tr->genLoadSym(ref->var);
        if(ref->ATTR(lv_kind) == ast::Lvalue::ARRAY_ELE)
            addr = tr->genAdd(addr, ref->idx_expr->ATTR(val));
        tr->genStore(s->e->ATTR(val), addr, 0);
    }
    else{
        if(ref->ATTR(lv_kind) == ast::Lvalue::ARRAY_ELE){
            Temp addr = tr->genAdd(ref->ATTR(sym)->getTemp(), ref->idx_expr-
>ATTR(val));
            tr->genStore(s->e->ATTR(val), addr, 0);
        }
        else
            // ...
    }
    s->ATTR(val) = s->e->ATTR(val);
}
```

数据流分析

在 `src/tac/dataflow.cpp`，为 `ALLOC` 指令设计数据流信息。

```
    case Tac::ALLOC:
        updateDEF(t->op0.var);
        break;

    case Tac::ALLOC:
        if (NULL != t_next->op0.var)
            t->LiveOut->remove(t_next->op0.var);
        break;
```

汇编代码生成

空间分配

在 `riscv_md.cpp` 中，将 `ALLOC` TAC 翻译为汇编代码。对栈顶指针 `sp` 进行修改，在栈上开辟出一块连续内存，并将这块内存的首地址返回即可。

```
void RiscvDesc::emitAllocTac(Tac *t) {
    int r0 = getRegForWrite(t->op0.var, 0, 0, t->LiveOut);
    addInstr(RiscvInstr::ADDI, _reg[RiscvReg::SP], _reg[RiscvReg::SP], NULL, -t-
>op1.size, EMPTY_STR, NULL);
    addInstr(RiscvInstr::MOVE, _reg[r0], _reg[RiscvReg::SP], NULL, 0, EMPTY_STR,
NULL);
}
```

数据段

与 step 10 的区别在于，将未初始化的全局变量则存放在 bss 段中时，需要根据数组声明设置 space。仿照 `FUNCTY` 的写法，在 `RiscvDesc::emitPieces(scope::GlobalScope *gscope, Piece *ps, std::ostream &os)` 中增加如下内容：

```
    while (NULL != ps) {
        switch (ps->kind) {
        case Piece::FUNCTY:
            emitFuncty(ps->as.functy);
            break;

        case Piece::GLOBAL:
            if (ps->as.globalVar->value == 0) {
                emit(EMPTY_STR, ((std::string)(".global ") + ps->as.globalVar-
>name).c_str(), NULL);
                emit((ps->as.globalVar->name).c_str(), NULL, NULL);
                emit(EMPTY_STR, ((std::string)(".space ") + std::to_string(ps-
>as.globalVar->size)).c_str(), NULL);
            }
            break;

        default:
            mind_assert(false); // unreachable
            break;
        }
        ps = ps->next;
    }
```

Step 12: 数组初始化与数组传参

词法分析和语法分析

在 `DeclStmt` 中增加数组初始化的内容。其中 `IntList` 为初始化列表。相应地修改`VarDecl` AST 节点，支持初始化。

```
DeclrStmt   : // ...
            | Type IDENTIFIER DimList ASSIGN LBRACE IntList RBRACE SEMICOLON
                { $$ = new ast::VarDecl($2, $1, $3, $6, POS(@2)); }
IntList     : ICONST
                { $$ = new ast::DimList();
                  $$->append($1); }
            | IntList COMMA ICONST
                { $1->append($3);
                  $$ = $1; }
```

在 `ParamList` 中支持数组参数。其中 `IntList` 为初始化列表。相应地修改 `VarDecl` AST 节点，支持初始化。

```
ParamList   : // ...
            | Type IDENTIFIER DimList
                { $$ = new ast::VarList();
                  $$->append( new ast::VarDecl($2, $1, $3, POS(@2)) );
                }
            | Type IDENTIFIER LBRACK RBRACK DimList
                { $$ = new ast::VarList();
                  $$->append( new ast::VarDecl($2, $1, new ast::DimList(),
POS(@2)) );
                }
            | ParamList COMMA Type IDENTIFIER DimList
                { $1->append( new ast::VarDecl($4, $3, $5, POS(@4)) );
                  $$ = $1;
                }
            | ParamList COMMA Type IDENTIFIER LBRACK RBRACK DimList
                { $1->append( new ast::VarDecl($4, $3, new ast::DimList(),
POS(@4)) );
                  $$ = $1;
                }
```

语义分析

符号表构建

在 `src/translation/build_sym.cpp` 中，修改 `SemPass1::visit(ast::VarDecl *vdecl)`，特别处理已经初始化的全局数组的情况。在 `Varaible` 类中增加成员变量 `ast::DimList *global_arr_init` 来记录全局数组的初始化列表，并增加成员函数 `setGlobalArrInit` 和 `getGlobalArrInit` 对该初始化列表进行读写。

```
void SemPass1::visit(ast::VarDecl *vdecl) {
    // ...
    // 4. Special processing for global variables
    if (v->isGlobalVar()){
```

```
        if (t->isBaseType()) { // global variable
            if (NULL != vdecl->init){
                ast::IntConst *init_value = dynamic_cast<ast::IntConst *>(vdecl->init);
                if (NULL != init_value)
                    v->setGlobalInit(init_value->value);
                else
                    issue(vdecl->getLocation(), new BadGlobalInitError(vdecl->name));
            }
            else
                v->setGlobalInit(0);
        }
        else if (t->isArrayType()) { // global array
            if (NULL != vdecl->arr_init)
                v->setGlobalArrInit(vdecl->arr_init);
        }
        else
            mind_assert(false);
    }
    // ...
}
```

中间代码生成

在 `src/translation/translation.cpp` 中，处理 `VarDecl`。对于全局数组，`genGlobalVarible`（重载，初始化值为列表）；对于局部数组，得到数组首地址后，先生成 `MEMSET` TAC 将数组对应的内存区域置为0，再生成一系列 `STORE` TAC，将初始值存入对应的内存单元。

```
void Translation::visit(ast::VarDecl *decl) {
    Variable *var = decl->ATTR(sym);
     if (decl->ATTR(sym)->isGlobalVar()) {
        if (decl->type->ATTR(type)->isBaseType()) {
            if(decl->init == NULL){
                tr->genGlobalVarible(decl->name, 0, decl->type->ATTR(type)->getSize());
            }
            else {
                assert(decl->init->getKind() == ast::ASTNode::INT_CONST);
                tr->genGlobalVarible(decl->name, ((ast::IntConst *)(decl->init))->value, decl->type->ATTR(type)->getSize());
            }
        }
        else if (decl->type->ATTR(type)->isArrayType()) {
            tr->genGlobalVarible(decl->name, decl->arr_init, decl->type->ATTR(type)->getSize());
        }
        else
            mind_assert(false);
    }
    else {
        if (decl->type->ATTR(type)->isArrayType()) {
            Temp addr = tr->genAlloc(decl->type->ATTR(type)->getSize());
            var->attachTemp(addr);
            if (NULL != decl->arr_init){
```

```
                int length = 1;
                for (auto iter = decl->dim->begin(); iter != decl->dim->end();
++iter)
                    length *= (*iter);
                Temp value = tr->genLoadImm4(0);
                tr->genMemset(addr, value, length);
                int offset = 0;
                for (auto iter = decl->arr_init->begin(); iter != decl-
>arr_init->end(); ++iter) {
                    Temp init_value = tr->genLoadImm4((*iter));
                    tr->genStore(init_value, addr, offset);
                    offset += 4;
                }
            }
        }
        else {
            var->attachTemp(tr->getNewTempI4());
        }

        if (NULL != decl->init){
            decl->init->accept(this);
            tr->genAssign(var->getTemp(), decl->init->ATTR(val));
        }
    }
}
```

数据流分析

为上述 `MEMSET` 补充数据流分析。根据指令特点，分析LiveUse, Define集合，计算LiveOut集合。

```
    case Tac::MEMSET:
        updateLU(t->op0.var);
        updateLU(t->op1.var);
        break;

    case Tac::MEMSET:
        t->LiveOut->add(t_next->op0.var);
        t->LiveOut->add(t_next->op1.var);
        break;
```

汇编代码生成

对于 `MEMSET` TAC，采取简单的实现方法，对每个位置产生一条赋值语句。在 `riscv_md.cpp` 中添加相应函数。

```
void RiscvDesc::emitMemsetTac(Tac *t) {
    int addr = getRegForWrite(t->op0.var, 0, 0, t->LiveOut);
    int val = getRegForRead(t->op1.var, 0, t->LiveOut);
    for (int offset = 0; offset < t->op1.size; offset = offset + 4) {
        addInstr(RiscvInstr::SW, _reg[val], _reg[addr], NULL, offset, EMPTY_STR,
NULL);
    }
}
```

修改已初始化的全局变量部分，即 `.data` 段。将数组初始化的值设为 `.word`，其余为0，即 `.zero`。在 `RiscvDesc::emitPieces(scope::GlobalScope *gscope, Piece *ps, std::ostream &os)` 中增加如下内容：

```
    while (NULL != ps) {
        switch (ps->kind) {
        case Piece::FUNCTY:
            break;

        case Piece::GLOBAL:
            if (ps->as.globalVar->size == 4 && ps->as.globalVar->value) {
                emit(EMPTY_STR, ((std::string)(".global ") + ps->as.globalVar-
>name).c_str(), NULL);
                emit((ps->as.globalVar->name).c_str(), NULL, NULL);
                emit(EMPTY_STR, ((std::string)(".word ") + std::to_string(ps-
>as.globalVar->value)).c_str(), NULL);
            }
            else if (NULL != ps->as.globalVar->arr_value) {
                emit(EMPTY_STR, ((std::string)(".global ") + ps->as.globalVar-
>name).c_str(), NULL);
                emit((ps->as.globalVar->name).c_str(), NULL, NULL);
                for (auto it = ps->as.globalVar->arr_value->begin(); it != ps-
>as.globalVar->arr_value->end(); ++it) {
                    emit(EMPTY_STR, ((std::string)(".word ") +
std::to_string((*it))).c_str(), NULL);
                }
                int remain = ps->as.globalVar->size - 4 * ps->as.globalVar-
>arr_value->length();
                emit(EMPTY_STR, ((std::string)(".zero ") +
std::to_string(remain)).c_str(), NULL);
            }
            break;

        default:
            mind_assert(false); // unreachable
            break;
        }
        ps = ps->next;
    }
```

对于数组传参，处理相对简单。如果参数是数组，则传递数组首地址即可。在 `Translation::visit(ast::LvalueExpr *e)` 函数中判断一下是否为 `BaseType` 即可。

```
    if (ref->ATTR(type)->isBaseType()) {
        if (ref->ATTR(lv_kind) == ast::Lvalue::ARRAY_ELE)
            addr = tr->genAdd(addr, ref->idx_expr->ATTR(val));
        e->ATTR(val) = tr->genLoad(addr, 0);
    }
    else { // array as parameter
        e->ATTR(val) = addr;
    }
```

思考题

1. 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？

   在进入函数时统一给局部变量分配内存，记录指向 `a` 的指针，在离开函数时统一释放这些内存。每次控制流经过该声明时，对 `n` 求值，然后在栈顶为数组分配空间，并修改指向 `a` 的指针的值。在离开作用域时，释放栈顶的空间。

2. 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？

   传递的参数实际上是数组的首地址。得到数组元素的地址，只需要知道其相对于首地址的偏移量。计算第 k 维下标对应的偏移量，只需要用到第 k+1, k+2, ... 维的大小，其中 $k \geq 1$。因此，第 1 维的大小是无用的，可以省略。

代码借鉴

Step 11 中间代码生成，对 `LvalueExpr` 和 `AssignExpr` 的处理：https://github.com/Bbeholder/mycompiler/blob/master/src/translation/translation.cpp