MiniDecaf Stage 4: 函数和全局变量

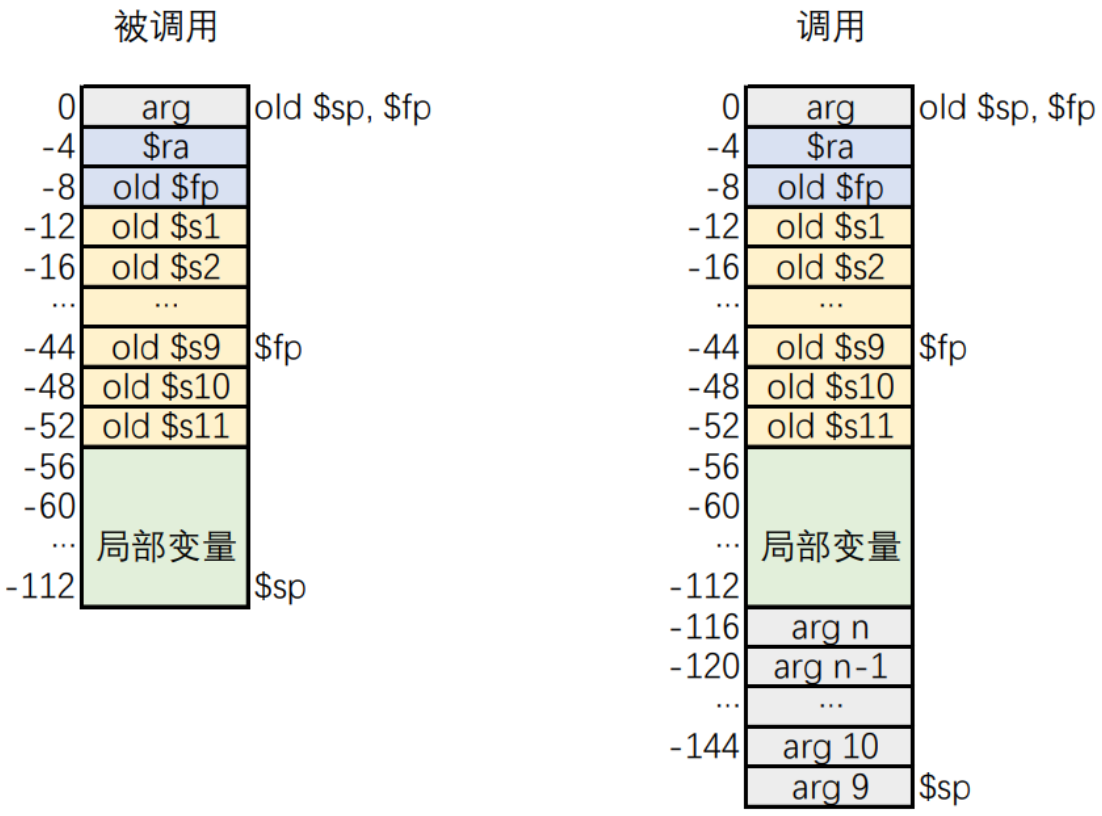
> My-laniaKeA

实验内容

Step 9: 函数

总体设计

本 step 的核心在于设计并实现运行时存储组织。我的设计如下图所示。在函数被调用时，创建栈帧，记录控制信息与 callee-saved 寄存器，并为局部变量预留空间。调用函数之前，保存 caller-saved 寄存器，并传递参数。前8个参数通过寄存器传递，其余参数压栈，更新 `$sp`。所调用函数返回时，反向进行上述操作。返回同理。



词法分析和语法分析

更新语法规则，支持函数调用。

```
FoDList :   FuncDefn
                { $$ = new ast::Program($1,POS(@1)); } |
            FoDList FuncDefn{
                {$1->func_and_globals->append($2);
                $$ = $1; }
                } |
```

```
FuncDefn : Type IDENTIFIER LPAREN FormalList RPAREN LBRACE BlockItemList RBRACE
{
            $$ = new ast::FuncDefn($2,$1,$4,$7,POS(@1));
            } |
            Type IDENTIFIER LPAREN FormalList RPAREN SEMICOLON{
                $$ = new ast::FuncDefn($2,$1,$4,new
ast::EmptyStmt(POS(@6)),POS(@1));
            };
FormalList :  /* EMPTY */
                { $$ = new ast::VarList(); }
            | ParamList
                { $$ = $1; };
ParamList   : Type IDENTIFIER
                { $$ = new ast::VarList();
                  $$->append( new ast::VarDecl($2, $1, POS(@2)) );
                }
            | ParamList COMMA Type IDENTIFIER
                { $1->append( new ast::VarDecl($4, $3, POS(@4)) );
                  $$ = $1;
                };
```

设计 AST 节点。

节点	成员	含义
`FuncDefn`	返回类型 `return_type`，函数名 `name`，形式参数列表 `formals`，函数体内语句 `stmts`	函数
`CallExpr`	调用函数名 `func_name`，参数列表 `exprs`	函数调用

语义分析

类型检查

在 `src/translation/type_check.cpp` 的 `SemPass2::visit(ast::CallExpr *callExpr)` 函数中，检查是否调用未定义的函数。检查函数类型，调用时传入的参数数量与类型是否和函数定义时的一致。

```
void SemPass2::visit(ast::CallExpr *callExpr) {
    callExpr->ATTR(type) = BaseType::Error;
    callExpr->ATTR(sym) = NULL;

    Function *f = (Function *)scopes->lookup(callExpr->func_name, callExpr-
>getLocation());
    if (NULL == f)
        issue(callExpr->getLocation(), new SymbolNotFoundError(callExpr-
>func_name));
    else if (!f->isFunction())
        issue(callExpr->getLocation(), new NotMethodError(f));
    else {
        callExpr->ATTR(type) = f->getResultType();
        callExpr->ATTR(sym) = f;
    }
```

```
    if (callExpr->exprs->length() != f->getType()->numOfParameters())
        issue(callExpr->getLocation(), new SymbolNotFoundError(callExpr->func_name));

    // visit parameter nodes
    // ...
}
```

中间代码生成

设计如下 TAC。

TAC	参数	含义
`CALL`	函数名称 `LABEL`，函数参数总数 `arg_num`	调用函数 `LABEL`
`PASS_PARAM`	`T0`，参数顺序 `order`，参数总数 `arg_num`	将 `T0` 设置为第 `order` 个实参
`PROCESS_PARAM`	`T0`，参数顺序 `order`	将 `T0` 设置为第 `order` 个形参

在 `src/translation/translation.cpp` 中，将 `CallExpr` 节点翻译为中间代码。首先，递归地访问参数节点，得到其 VAL 属性。然后，对于每一个参数，生成一个 `PASS_PARAM` TAC。最后，生成 `CALL` TAC。

```
void Translation::visit(ast::CallExpr *e) {
    // arguments
    for (ast::ExprList::iterator it = e->exprs->begin(); it != e->exprs->end(); ++it)
        (*it)->accept(this);

    int order = 0;
    for (ast::ExprList::iterator it = e->exprs->begin(); it != e->exprs->end(); ++it){
        tr->genPassParam((*it)->ATTR(val), order, e->exprs->length());
        order++;
    }

    e->ATTR(val) = tr->genCall(e->ATTR(sym)->getEntryLabel(), e->exprs->length());
}
```

在 `FuncDefn` 节点中，增加 `PROCESS_PARAM` TAC 的代码。

```
void Translation::visit(ast::FuncDefn *f) {
    // ...
    if (!f->forward_decl){
        tr->startFunc(fun);
        // You may process params here, i.e use reg or stack to pass parameters
        for (auto it = f->formals->begin(); it != f->formals->end(); ++it) {
            auto v = (*it)->ATTR(sym);
            tr->genProcessParam(v->getTemp(), v->getOrder());
```

```
        }
        // translates statement by statement
        // ...
    }
    // ...
}
```

数据流分析

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

```
    case Tac::PASS_PARAM:
        updateDEF(t->op0.var);
        break;
    case Tac::PROCESS_PARAM:
    case Tac::CALL:
        updateLU(t->op0.var);
        break;

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

汇编代码生成

建立栈帧

在 `emitProlog` 函数中，(1)保存控制信息，(2)保存 callee-saved 寄存器，(3)修改 $fp（因为后续存放临时变量时，都是从 -12(fp) 开始的，所以 $fp 仅需 -44。

```
void RiscvDesc::emitProlog(Label entry_label, int frame_size) {
    std::ostringstream oss;
    // ...
    // saves old context
    emit(EMPTY_STR, "sw    ra, -4(sp)", NULL); // saves old frame pointer
    emit(EMPTY_STR, "sw    fp, -8(sp)", NULL); // saves return address
    // establishes new stack frame (new context)
    emit(EMPTY_STR, "mv    fp, sp", NULL);
    // oss << "addi  sp, sp, -" << (frame_size + 2 * WORD_SIZE); // 2 WORD's for
old $fp and $ra
    oss << "addi  sp, sp, -" << (frame_size + 13 * WORD_SIZE); // 2 WORD's for
old $fp and $ra, 11 WORD's for old $s1 - $s11
    emit(EMPTY_STR, oss.str().c_str(), NULL);
    oss.str("");

    // save calle-saved registers
    emit(EMPTY_STR, NULL, "save callee-saved registers"); // marks the function
entry label
    oss.str("");
    int cnt = -12;
```

```
    for (int i = RiscvReg::S1; i <= RiscvReg::S11; i++){
        if (_reg[i]->saver == CALLEE){
            oss << "sw  " << _reg[i]->name << ", " << cnt << "(fp)";
            emit(EMPTY_STR, oss.str().c_str(), NULL);
            oss.str("");
            cnt -= 4;
        }
    }
    emit(EMPTY_STR, "addi    fp, fp, -44", NULL);
}
```

传递实参

在 `emitPassParamTac` 函数中，根据参数的顺序，处理传递实参的过程。小于 8 时，通过寄存器传参。大于等于 8 时，压栈传参。通过 `PASS_PARAM` TAC 的 `arg_num` 计算压栈时候在栈上的偏移量。

```
void RiscvDesc::emitPassParamTac(Tac *t){
    int order = t->op0.order;   // parameter order
    if (order < 8){ // pass via register
        passParamReg(t, order);
    }
    else{ // pass via stack
        int total_arg_num = t->op0.total_arg_num;
        if (order == 8){
            int new_stack_size = WORD_SIZE * (total_arg_num - 8);
            addInstr(RiscvInstr::ADDI, _reg[RiscvReg::SP], _reg[RiscvReg::SP],
                     NULL, -new_stack_size, EMPTY_STR, NULL);
        }
        Set<Temp>* liveness = t->LiveOut->clone();
        liveness->add(t->op0.var);
        int r0 = getRegForRead(t->op0.var, 0, liveness);
        addInstr(RiscvInstr::SW, _reg[r0],  _reg[RiscvReg::SP], NULL,
                 WORD_SIZE * (order-8), EMPTY_STR, NULL);
    }
}
```

处理形参

与传递实参类似，根据参数顺序进行相应的处理。

```
void RiscvDesc::emitProcessParamTac(Tac *t){
    int order = t->op0.order;   // parameter order
    if (order < 8){ // pass via register
        int r0 = getRegForRead(t->op0.var, 0, t->LiveOut);
        addInstr(RiscvInstr::MOVE, _reg[r0], _reg[RiscvReg::A0 + order], NULL,
0, EMPTY_STR, NULL);
        _reg[r0]->dirty = true;
    }
    else{ // pass via stack
        int r0 = getRegForWrite(t->op0.var, RiscvReg::FP, 0, t->LiveOut);
        addInstr(RiscvInstr::LW, _reg[r0],  _reg[RiscvReg::FP], NULL,
                WORD_SIZE * (order-8) + 44, EMPTY_STR, NULL);
    }
}
```

函数调用

函数调用的处理分为保存、调用、返回值、恢复四个步骤。在保存与恢复中，通过 `_old_reg` 数组记录被 spill 到栈中的寄存器。

```
void RiscvDesc::emitCallTac(Tac *t){
    // 1. save caller-saved registers and temp variables
    for (int i = 0; i < RiscvReg::TOTAL_NUM; ++i) {
        _old_reg[i]->dirty = false;
        _old_reg[i]->var = NULL;

        Temp v = _reg[i]->var;
        if (_reg[i]->saver == CALLER &&
            (NULL != v) && t->LiveOut->contains(v)){
            spillReg(i, t->LiveOut);
            _old_reg[i]->dirty = true;
            _old_reg[i]->var = v;
        }
    }

    // 2. function call
    addInstr(RiscvInstr::CALL, NULL, NULL, NULL, 0, std::string("_") + t-
>op1.label->str_form, NULL);

    // 3. save return value to dest register
    int r0 = getRegForWrite(t->op0.var, 0, 0, t->LiveOut);
    addInstr(RiscvInstr::MOVE, _reg[r0], _reg[RiscvReg::A0], NULL, 0, EMPTY_STR,
"return value");
    _reg[RiscvReg::A0]->var = NULL;
    _reg[r0]->var = t->op0.var;

    // 4. recover caller-saved registers and temp variables
    t->LiveOut->add(t->op0.var);
     for (int i = 0; i < RiscvReg::TOTAL_NUM; ++i) {
       if (_reg[i]->saver == CALLER &&
            (NULL != _old_reg[i]->var) && _old_reg[i]->dirty && t->LiveOut-
>contains(_old_reg[i]->var)){
            recoverReg(i, t->LiveOut);
        }
    }
}
```

栈帧销毁

在 `prepareSingleChain` 函数中，对于 `BY_RETURN` 的情形，(1) 恢复 $fp，(2) 恢复 callee-saved 寄存器，(3) 恢复 $sp 等。

```
RiscvInstr *RiscvDesc::prepareSingleChain(BasicBlock *b, FlowGraph *g) {
    // ...
    switch (b->end_kind) {
    case BasicBlock::BY_RETURN:{
        r0 = getRegForRead(b->var, 0, b->LiveOut);
        spillDirtyRegs(b->LiveOut); // just to deattach all temporary variables
        addInstr(RiscvInstr::MOVE, _reg[RiscvReg::A0], _reg[r0], NULL, 0,
                 EMPTY_STR, NULL);
```

```
        addInstr(RiscvInstr::ADDI, _reg[RiscvReg::FP], _reg[RiscvReg::FP], NULL,
                 44, EMPTY_STR, NULL); // addi fp, fp, 44
        // recover callee-saved registers
        int cnt = -12;
        for (int i = RiscvReg::S1; i <= RiscvReg::S11; i++){
            if (_reg[i]->saver == CALLEE){
                addInstr(RiscvInstr::LW, _reg[i], _reg[RiscvReg::FP], NULL,
                         cnt, EMPTY_STR,NULL);
                cnt -= 4;
            }
        }
        addInstr(RiscvInstr::MOVE, _reg[RiscvReg::SP], _reg[RiscvReg::FP], NULL,
                 0, EMPTY_STR, NULL);
        addInstr(RiscvInstr::LW, _reg[RiscvReg::RA], _reg[RiscvReg::FP], NULL,
                 -4, EMPTY_STR, NULL);
        addInstr(RiscvInstr::LW, _reg[RiscvReg::FP], _reg[RiscvReg::FP], NULL,
                 -8, EMPTY_STR, NULL);
        addInstr(RiscvInstr::RET, NULL, NULL, NULL, 0, EMPTY_STR, NULL);
        }
        break;
    // ...
    }
    _tail = NULL;
    return leading.next;
}
```

Step 10: 全局变量

词法分析和语法分析

增加语法规则，支持 `VarDecl` 节点。

```
FoDList :   FuncDefn
                { $$ = new ast::Program($1,POS(@1)); } |
            DeclrStmt
                { $$ = new ast::Program($1,POS(@1)); } |
            FoDList FuncDefn{
                {$1->func_and_globals->append($2);
                $$ = $1; }
                } |
            FoDList DeclrStmt{
                {$1->func_and_globals->append($2);
                $$ = $1; }
                }
```

语义分析

符号表构建

在 `src/translation/build_sym.cpp` 中，修改 `SemPass1::visit(ast::VarDecl *vdecl)`，特别处理全局变量的情况。如果有对全局变量进行初始化，则设置其初始化值。通过 `<dynamic_cast>` 来保证全局变量只能被整型常量初始化。

```
void SemPass1::visit(ast::VarDecl *vdecl) {
    Type *t = NULL;
    vdecl->type->accept(this);
    t = vdecl->type->ATTR(type);
    // Add a new symbol to a scope
    // 1. Create a new `Variable` symbol
    Variable *v = new Variable(vdecl->name, t, vdecl->getLocation());
    // 2. Check for conflict in `scopes`, which is a global variable refering to
    // a scope stack
    Symbol *sym = scopes->lookup(vdecl->name, vdecl->getLocation(), false);
    if (NULL != sym)
        issue(vdecl->getLocation(), new DeclConflictError(vdecl->name, sym));
    // 3. Declare the symbol in `scopes`
    else
        scopes->declare(v);
    // 4. Special processing for global variables
    if (v->isGlobalVar()){
        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);
        }
    }
    // 5. Tag the symbol to `vdecl->ATTR(sym)`
    vdecl->ATTR(sym) = v;
    // 6. Visit initialization expression if existed
    if (NULL != vdecl->init)
        vdecl->init->accept(this);
}
```

中间代码生成

在 `src/translation/translation.cpp` 中，处理 `AssignExpr`，`LvalueExpr`。当左值为全局变量时，先通过 `LOAD_SYMBOL` TAC 得到地址，再通过 `LOAD`，`STORE` 进行读写操作。

```
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);
        tr->genStore(s->e->ATTR(val), addr, 0);
    }
    else{
        tr->genAssign(s->left->ATTR(val), s->e->ATTR(val));
    }
    s->ATTR(val) = s->e->ATTR(val);
}
```

```
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());
        e->ATTR(val) = tr->genLoad(addr, 0);
    }
    else {
        e->ATTR(val) = ref->ATTR(sym)->getTemp();
    }
}
```

数据流分析

为上述3个 TAC 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::LOAD_SYMBOL:
        updateDEF(t->op0.var);
        break;
    case Tac::LOAD:
        updateLU(t->op1.var);
        updateDEF(t->op0.var);
        break;
    case Tac::STORE:
        updateLU(t->op0.var);
        updateLU(t->op1.var);
        break;

    case Tac::LOAD_SYMBOL:
        if (NULL != t_next->op0.var)
            t->LiveOut->remove(t_next->op0.var);
        break;
    case Tac::LOAD:
        if (NULL != t_next->op0.var)
            t->LiveOut->remove(t_next->op0.var);
        t->LiveOut->add(t_next->op1.var);
        break;
    case Tac::STORE:
        t->LiveOut->add(t_next->op0.var);
        t->LiveOut->add(t_next->op1.var);
```

```
        break;
```

汇编代码生成

直接使用 RISC-V 指令翻译 TAC 即可。

TAC	RISC-V
`LOAD_SYMBOL`	LA reg[`op0.var`], `op1.name`
`LOAD`	LW reg[`op0.var`], `op1.offset`(reg[`op1.var`])
`STORE`	SW reg[`op0.var`], `op1.offset`(reg[`op1.var`])

思考题

1. MiniDecaf 的函数调用时参数求值的顺序是未定义行为。试写出一段 MiniDecaf 代码，使得不同的参数求值顺序会导致不同的返回结果。

```
int sub(int x, int y){
    return x - y;
}

int main()
{
    int i = 2022;
    return sub(i, (i=i-1));
}
```

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？

   对于活跃期很短的临时变量，应当由易失性寄存器来保存，以减少频繁出栈入栈的开销，提高性能。

   这样可以防止被调用者更改返回地址，返回到错误/无权限的地址发起攻击。

3. 写出 `la v0, a` 这一 RiscV 伪指令可能会被转换成哪些 RiscV 指令的组合（说出两种可能即可）。

```
auipc   rd, offsetHi
lw      rd, offsetLo(rd)
```

```
auipc   rd, offsetHi
addi    rd, rd, offsetLo
lw      rd, 0(rd)
```

代码借鉴

Step 10 中间代码生成：https://github.com/Bbeholder/mycompiler/blob/master/src/translation/translation.cpp