实验二　基于 LLVM 的中间代码生成

姓名：邱明阳 王思源

日期：2024 年 12 月 20 日

1 实验目的

1. 熟悉 LLVM IR 语法描述。
2. 熟悉 LLVM IR 生成接口的功能与使用方法。
3. 熟悉静态与动态检查和报错的实现方式。

2 实验内容

1. 基于实验一生成的抽象语法树，生成测例的 LLVM IR。
2. 对部分语义错误进行编译时静态检查和报错。
3. 对访问 obc 数组的语句进行插入用于数组越界访问检测的 IR 代码，进行动态检查和报错。

3 实验过程

3.1 生成基本语义正确的中间代码

需要完善 SafeCIRBuiler.cpp 文件中的各函数。首先分析包含抽象语法树节点定义的 AstNode.h 文件，可以发现从语法分析树的根节点开始迭代访问子语法节点，最后总会收敛到访问 lval_node 与 number_node 节点。

因此笔者首先完善了 SafeCIRBuiler.cpp 文件中 lval_node 与 number_node 的访问函数，然后自底部向顶部完善其他节点的访问函数。而在具体某个节点的访问函数完善过程中，则是先根据 test 测例文件的代码分析此节点中可能出现的不同情况（如数组与非数组、常量或变量），再根据不同情况设计对应处理措施。

在整个完善过程中，lval_node 与 var_def_node 这两个涉及变量处理的节点给笔者的感觉是最为关键与复杂的，也是最能体现上面所说“根据不同情况设计对应处理措施”这一

理念的。所以对这两个节点将会在下面详细解释完善的思路并展示相应代码，对其他节点只会简单解释思路。

lval_node 访问函数的分析与设计过程如下：

```
void SafeCIRBuilder::visit(lval_node& node) {
    std::cout << "visit(lval_node& node) 被调用" << std::endl;
    auto name = node.name;
    std::cout << "lval name is " << name << std::endl;
    VarInfo var_info = lookup_variable(name);
    if (!var_info.is_valid()) {
        std::cerr << node.line << ":" << node.pos << ": variable '" << name
                  << "' is not declared" << std::endl;
        error_flag = true;
        return;
    }
```

由于 lval_node 涉及变量，因此要在访问函数开始时检查变量是否已声明。检查变量是否已声明需要调用 lookup_variable 函数获取变量的相关信息，此函数也需要自主完善。lookup_variable 函数从最近的作用域中查找变量并返回找到的最近声明的变量名称。

```
SafeCIRBuilder::VarInfo SafeCIRBuilder::lookup_variable(std::string name) {
    // TODO: find the nearest decalred variable `name`
    // 从最近的作用域开始遍历查找变量
    for (auto scope = scoped_variables.rbegin(); scope != scoped_variables.rend
(); ++scope) {
        int if_found = scope->variable_map.count(name);
        if (if_found) {
            return scope->variable_map[name]; // 找到变量，返回其名称
        }
    }
    return VarInfo(); // Return an invalid VarInfo if `name` not found.
}
```

回到 lval_node 访问函数，在检查变量被正确声明后，根据变量是否为数组，对其执行不同操作：

当变量为非数组时，如果期望的变量为左值，将变量的地址传递给 set_value_result 函数；如果期望的变量为右值，使用 LLVM 的 CreateLoad 函数获取变量的值并传递给 set_ value_result 函数（左值与右值的说明见下面的实验问题部分）

```
if (!var_info.is_array) {
        std::cout << "判定为非数组" << std::endl;
```

```
        // TODO: handle scalar lval
        if (IS_EXPECT_LVAL()) {
                set_value_result(var_info.val_ptr);
        }

        else {
            llvm::Value *load_val = builder.CreateLoad(var_info.val_ptr->getTyp
e()->getPointerElementType(), var_info.val_ptr, "loadtmp");
            set_value_result(load_val);
        }
    }
```

当变量为数组时，则需要考虑更多情况：数组索引为空时，后续处理方式与非数组变量相同；数组索引不为空时，则需要访问索引并获取索引值来计算所要操作的数组元素地址，然后仍是按相同的左右值处理方式将对应的值传递给 set_value_result 函数。

```
else {
        std::cout << "判定为数组" << std::endl;
        std::cout << "数组大小: " << var_info.array_length << std::endl;
        // TODO: handle array lval and call obc_check to insert obc check code
for obc array.
        llvm::Value* array_ptr = var_info.val_ptr;
        if (node.array_index == nullptr) {
            if (IS_EXPECT_LVAL()) {
                    set_value_result(array_ptr);
            } else {
                llvm::Value *load_val = builder.CreateLoad(array_ptr->getType()
->getPointerElementType(), array_ptr, "loadtmp");
                set_value_result(load_val);
            }
            return;
        } else {
            if (!array_ptr) {
                std::cerr << "Error: array_ptr is null" << std::endl;
                return;
            }
            var_info.val_ptr->getType();
            std::cout << "准备访问索引" << std::endl;
            EXPECT_RVAL(node.array_index->accept(*this));
            std::cout << "访问索引结束" << std::endl;
            llvm::Value *index_value;
            if (!get_result_as_value(&index_value)) {
                std::cerr << node.line << ":" << node.pos
```

```
                        << ": array index must be a constant or a vailable"
                        << std::endl;
                return;
            }
            std::cout << "索引值获取完成" << std::endl;
            if (var_info.is_obc){
                obc_check(index_value, var_info.array_length, node.line, node.p
os, name);
            }

            std::cout << "array_ptr type: " << std::endl;
            if (!array_ptr) {
                std::cerr << "Error: array_ptr is null" << std::endl;
                return;
            }
            llvm::Value* element_ptr = builder.CreateGEP(array_ptr->getType()->
getPointerElementType(), array_ptr, {builder.getInt32(0), index_value}, "elemen
tptr");
            std::cout << "元素地址计算完成" << std::endl;
            if (IS_EXPECT_LVAL())
                // 如果期望左值，返回数组元素的地址
                set_value_result(element_ptr);
            else {
                // 如果期望右值，加载数组元素的值并返回加载后的值
                llvm::Value* load_val = builder.CreateLoad(element_ptr->getType
()->getPointerElementType(), element_ptr, "loadtmp");
                set_value_result(load_val);
            }
        }
        return;
```

可以看到，整个的 lval_node 访问函数的设计过程中要考虑变量是否定义、变量是否为数组、数组索引是否为空、上层节点期望左值还是右值等多种不同情况。

var_def_node 访问函数的分析与设计过程如下（展示代码省略了部分参数定义）：

```
if (cur_scope == FLAG::GLOBAL_SCOPE) {  // global define
        llvm::GlobalVariable* global_variable;
        llvm::Constant *init_value;
```

由于全局变量与局部变量的声明方法不同，所以 var_def_node 访问函数一开始就要区分这两种情况。

确定为全局变量后，还要对是否为数组进行区分，下面展示的是非数组全局变量。当有初始化值时，将初始化值 res_const 转换为 LLVM 常量整数，并赋值给 init_value，如果没有初始化值，则将 0 转换为 LLVM 常量整数，并赋值给 init_value。

```
if (!array_length) {  // not an array
    if (!node.initializers.empty()) {
        EXPECT_RVAL(node.initializers[0]->accept(*this));
        if (!get_int_result(res_const)) {
            std::cerr << node.line << ":" << node.pos
                << ": initializer must be a constant."
                << std::endl;
            return;
        } else {
            // init_value = llvm::cast<llvm::Constant>(builder.getInt32(res_con
st));
            init_value = llvm::ConstantInt::get(context, llvm::APInt(32, res_co
nst));
        }
    } else {
        // init_value = llvm::cast<llvm::Constant>(builder.getInt32(0));
        init_value = llvm::ConstantInt::get(context, llvm::APInt(32, 0));
    }
```

然后使用 llvm::GlobalVariable 构造函数创建一个新的全局变量，并使用 declare_ variable 函数将全局变量添加到符号表中。

```
    global_variable = new llvm::GlobalVariable(
        *module,
        llvm::Type::getInt32Ty(context),
        is_const,
        llvm::GlobalValue::ExternalLinkage,
        init_value, name);
    if (!declare_variable(name, global_variable, is_const, false, is_obc, 0)){
        std::cerr << node.line << ":" << node.pos << ":"
            << " variable '" << name << "' is declared more than one times"
            << std::endl;
        error_flag = true;
        return;
    }
```

以下代码展示的是数组全局变量的处理流程：

首先要获取数组长度，然后根据数组长度使用 llvm::ArrayType::get 方法创建一个数组类型。当有初始化值时，遍历每个数组元素，处理每个初始化值，如果初始化元素小于声明的数组元素数量，使用零值填充剩余元素，最后将这些结果并赋值给 init_value；当没有初始化值时，使用 llvm::ConstantAggregateZero::get 方法创建一个全零初始化的数组并赋值给 init_value。

```
} else {              // is an array
    int array_length;
    EXPECT_RVAL(node.array_length->accept(*this));
    if (!get_int_result(array_length)) {
        std::cerr << node.line << ":" << node.pos
            << ": array length must be a constant"
            << std::endl;
        return;
    } else {
        if (array_length < 0){
            std::cerr << node.line << ":" << node.pos << ":"
                << " size of array '" << name << "' is not positive"
                << std::endl;
            error_flag = true;
            return;
        }
        llvm::ArrayType *array_type = llvm::ArrayType::get(llvm::Type::getInt32
Ty(context), array_length);

        if (!node.initializers.empty()) {
            std::vector<llvm::Constant*> elements;
            for (auto init : node.initializers) {
                EXPECT_RVAL(init->accept(*this));
                if (!get_int_result(res_const)) {
                    std::cerr << node.line << ":" << node.pos << ":"
                        << "initializer must be a constant"
                        << std::endl;
                    return;
                }
                elements.push_back(llvm::ConstantInt::get(context, llvm::APInt(
32, res_const)));
            }
            if (elements.size() > array_length) {
                std::cerr << node.line << ":" << node.pos << ":"
```

```
                    << " excess elements in the initializer of array '" << name
<<"'"
                    << std::endl;
                error_flag = true;
                return;
            }
            for (size_t i = elements.size(); i < array_length; ++i) {
                elements.push_back(llvm::ConstantInt::get(context, llvm::APInt(
32, 0)));
            }
            init_value = llvm::ConstantArray::get(array_type, elements);
        } else {
            init_value = llvm::ConstantAggregateZero::get(array_type);
        }
```

同样使用 llvm::GlobalVariable 构造函数创建一个新的全局变量，并使用 declare_ variable 函数将全局变量添加到符号表中。但与上面不同的是，这里 llvm 的 Type 类型相应地使用 ArrayType 表示这是一个数组。

```
                global_variable = new llvm::GlobalVariable(
                    *module,
                    llvm::ArrayType::get(llvm::Type::getInt32Ty(context),
                    array_length),
                    is_const,
                    llvm::GlobalValue::ExternalLinkage,
                    init_value, name);

                if(!declare_variable(name, global_variable, is_const, true,
                    is_obc, array_length)){
                    std::cerr << node.line << ":" << node.pos << ":"
                    <<"variable'"<< name <<"' is declared more than one times"
                    << std::endl;
                    error_flag = true;
                    return;
                }
            }
        }
    } else {  // local define
        llvm::Value *local_variable;
        llvm::Value *init_value;

        if (!array_length) {  // not an array
            // DONE: create and declare local scalar
```

```
        llvm::AllocaInst *local_variable = builder.CreateAlloca(llvm::Type::
                                        getInt32Ty(context), nullptr, name);

        if (!node.initializers.empty()) {
            EXPECT_RVAL(node.initializers[0]->accept(*this));
            if (!get_result_as_value(&init_value)) {
                std::cerr << node.line << ":" << node.pos
                          <<":initializer must be a constant or a variable."
                           << std::endl;
                return;
            }
        } else {
            init_value = llvm::ConstantInt::get(context,llvm::APInt(32,0));
        }
        builder.CreateStore(init_value, local_variable);
        if(!declare_variable(name,local_variable,is_const,false,is_obc,0)){
            std::cerr << node.line << ":" << node.pos << ":"
            << " variable '" << name << "' is declared more than one times"
            << std::endl;
            error_flag = true;
            return;
        }
```

对于局部变量定义来说，对是否为数组以及是否有初始化的区分处理流程与全局变量定义的思路相同，不再赘述，唯一有较大不同的不能直接使用 llvm::GlobalVariable 这样的单个构造函数完成变量定义，而是要先使用 llvm::AllocaInst 在当前函数的栈上分配空间给局部变量，再使用 builder.CreateStore 函数将初始化结果存入栈上空间。

可以看到，整个的 var_def_node 访问函数的设计过程中要考虑是否为全局变量、变量是否为数组、数组初始化是否为空、初始化元素是否小于声明的数组元素数量等不同情况。

通过对 lval_node 访问函数与 var_def_node 访问函数实现过程的详细分析，我们能得到完善各节点访问函数的大致方向：找到可能出现的所有情况，并根据不同情况设计对应处理流程。剩下的函数中，cond_node、binop_expr_node、unaryop_expr_node 访问函数就是要根据不同的 op 值进行对应的运算操作并存储运算值；if_stmt_node、while_stmt_node 访问函数就是要根据条件表达式成立与否从而将控制流分支到对应的的基本块。分析出每种情况需要的处理流程后，代码实现其实很容易。

总的来说，笔者进行这部分实验的核心思路就是从纵向和横向两个维度梳理清楚中间代码的生成流程。纵向指的是从语法树的层面自上而下地分析各节点地迭代关系，理解父子节点之间的数据传递关系（如期望左值还是右值），最后按自下而上顺序完善节点访问函数。横向是指在完善某个节点访问函数的过程中，分析可能出现的不同情况，并对每种情况都设计出对应的处理流程。

3.2 语义错误静态检查和报错

需要完善 SafeCIRBuiler.cpp 文件中的各函数，使程序编译时对部分语义错误进行静态检查和报错。这一部分任务实际上已经在上一部分中一并完成。因为笔者将语义错误也包含在访问函数可能出现的情况中一并分析，进一步设计出对应的报错信息作为处理流程。下面展示 tests 测例中要求的语义错误检查如何在函数中实现：

3.2.1 变量重复声明

在 var_def_node 访问函数中，用 declare_variable 函数返回值判断是否有重复声明情况。如果存在同名变量，返回 false，进入相应的 if 分支处理流程，打印变量声明失败报错信息。

```
if (!declare_variable(name, global_variable, is_const, false, is_obc, 0)){
    std::cerr << node.line << ":" << node.pos << ":"
        << " variable '" << name << "' is declared more than one times"
        << std::endl;
    error_flag = true;
    return;
}
```

3.2.2 初始化元素数量超过数组长度

在 var_def_node 访问函数中的数组声明流程中，用初始化列表中元素数量与声明的数组长度进行比较。如果初始化元素数量超过数组长度，条件语句返回 false，进入相应的 if 分支处理流程，打印变量声明失败报错信息。

```
if (elements.size() > array_length) {
    std::cerr << node.line << ":" << node.pos << ":"
        << " excess elements in the initializer of array '"
        << name << "'" << std::endl;
```

```
    error_flag = true;
    return;
}
```

3.2.3 数组大小非正数

在 var_def_node 访问函数中的数组声明流程中，用声明的数组长度与 0 进行比较。如果数组长度小于 0，条件语句返回 false，进入相应的 if 分支处理流程，打印变量声明失败报错信息。

```
if (array_length < 0){
    std::cerr << node.line << ":" << node.pos << ":"
        << " size of array '" << name
        << "' is not positive" << std::endl;
    error_flag = true;
    return;
}
```

3.2.4 修改只读变量

在 assign_stmt_node 访问函数访问 target 与 value 子节点前，对 target 变量（所要赋值的变量）的 is_const 值进行检测。如果 is_const 为 true，说明目标变量为常量，不能进行赋值操作。进入相应的 if 分支处理流程，打印变量声明失败报错信息。

```
if (var_info.is_const) {
    std::cerr << node.line << ":" << node.pos
        << ": assignment of read-only variable '"
        << name << "'" << std::endl;
    error_flag = true;
    return;
}
```

从上述代码可以看出，静态语义错误检查的实现比较简单，只需分析出错误在各函数流程中可能出现的位置并进行相应的 if 条件判断即可。

3.3 obc 数组越界访问检测

需要完善 SafeCIRBuiler.cpp 文件中的 obc_check 函数，使程序在生成 IR 的运行过程中，在访问 obc 数组的指令之前，插入 IR 代码对 index 进行检查。

首先获取当前插入点所在的基本块所属的函数，并创建两个基本块：check_fail_bb（检查失败块）和 check_success_bb（检查成功块）

```
llvm::Function* function = builder.GetInsertBlock()->getParent();
llvm::BasicBlock* current_bb = builder.GetInsertBlock();
llvm::BasicBlock* check_fail_bb = llvm::BasicBlock::Create(context, "check_fail
", function);
llvm::BasicBlock* check_success_bb = llvm::BasicBlock::Create(context, "check_s
uccess", function);
```

然后检查数组索引是否越界，并根据越界与否进行相依的条件分支处理，如果越界就进入 check_fail_bb，反之进入 check_success_bb。

```
// 比较 index < 0
llvm::Value* isNegative = builder.CreateICmpSLT(index, llvm::ConstantInt::get(l
lvm::Type::getInt32Ty(context), 0), "isNegative");
// 比较 index >= array_length
llvm::Value* isOutOfRange = builder.CreateICmpSGE(index, llvm::ConstantInt::get
(llvm::Type::getInt32Ty(context), array_length), "isOutOfRange");
// 将两个比较结果进行 OR 操作
llvm::Value* cmp = builder.CreateOr(isNegative, isOutOfRange, "cmp");
// 根据 cmp 结果进行条件分支
builder.CreateCondBr(cmp, check_fail_bb, check_success_bb);
```

最后分别在 check_fail_bb 与 check_success_bb 块中插入对应的处理代码。check_fail_bb 块中按照创建错误处理函数 obc_check_error 的参数、获取参数变量的指针、将参数值存储到参数变量、调用 obc_check_error 错误处理函数的流程处理；check_success_bb 块不需要任何处理操作。

```
builder.SetInsertPoint(check_fail_bb);
llvm::Value *arg0_val = llvm::ConstantInt::get(llvm::Type::getInt32Ty(context),
 node_line);
llvm::Value *arg1_val = llvm::ConstantInt::get(llvm::Type::getInt32Ty(context),
 node_pos);
llvm::Value *arg2_val = llvm::ConstantDataArray::getString(context, name);
llvm::Value *arg0_ptr = lookup_variable("arg0").val_ptr;
llvm::Value *arg1_ptr = lookup_variable("arg1").val_ptr;
llvm::Value *arg2_ptr = lookup_variable("arg2").val_ptr;
llvm::Function *check_err = functions["obc_check_error"];
builder.CreateStore(arg0_val, arg0_ptr);
builder.CreateStore(arg1_val, arg1_ptr);
builder.CreateStore(arg2_val, arg2_ptr);
```

```
builder.CreateCall(check_err, {});
builder.CreateRetVoid();
builder.SetInsertPoint(check_success_bb);
return;
```

这种插入 IR 代码对数组索引进行检查的方式允许代码在运行时检查数组索引是否越界，而不是在编译时进行检查，这样做可以处理在程序执行过程中动态生成的索引值。像 tests 测例中的 a[input_var]部分，input_var 是一个动态生成的索引值，它的值在编译时是未知的，只有在运行时通过 input() 函数获取用户输入后才能确定。

3.4 样例测试

完成代码后，运行 check.py 进行测试，如图所示，20 个样例均通过。

```
armin@armin-virtual-machine:~/work/github/2024-fall-CPP/Compiler-lab2/Lab2$ sudo python3 check.py
[sudo] armin 的密码：
array_init_1.c pass.
array_init_2.c pass.
array_init_3.c pass.
array_obc.c pass.
array_unobc_overflow.c pass.
calc.c pass.
global_func_1.c pass.
global_func_2.c pass.
global_var.c pass.
global_var_array.c pass.
global_var_array_const_obc_1.c pass.
global_var_array_const_obc_2.c pass.
global_var_array_init_1.c pass.
global_var_array_init_2.c pass.
global_var_const.c pass.
if.c pass.
recursive_func.c pass.
var_def_1.c pass.
var_def_2.c pass.
while.c pass.
20 / 20 Passed.
armin@armin-virtual-machine:~/work/github/2024-fall-CPP/Compiler-lab2/Lab2$
```

图 1：运行结果

4 实验问题

在进行实验时，笔者遇到的最大困难就是如何理解左值与右值的功能区别。

SafeCIRBuiler.cpp 文件开头便定义了三个与左、右值有关的宏定义，笔者最初没有理解这些宏定义的作用，经过询问 compilot 理解了 EXPECT_RVAL 和 EXPECT_LVAL 宏是在调用函数前后分别使用 push 与 pop 在栈中设置和恢复期望的返回值类型，从而使

调用函数可以调用 IS_EXPECT_LVAL 宏压出栈顶元素检查当前函数被期望返回的值类型是否为左值。

```
// For lval and rval identification.
// Expect the function f to return an rval.
#define EXPECT_RVAL(f) val_type_stack.push(FLAG::RVAL); \
    f; \
    val_type_stack.pop();
// Expect the function f to return an lval.
#define EXPECT_LVAL(f) val_type_stack.push(FLAG::LVAL); \
    f; \
    val_type_stack.pop();
// Is current expected to return an lval.
#define IS_EXPECT_LVAL() val_type_stack.top() == FLAG::LVAL
```

区分左值与右值对于生成正确的赋值代码非常重要。在 assign_stmt_node 访问函数中笔者使用以下代码完成赋值：

```
// 将赋值表达式的右值存储到目标变量的左值
builder.CreateStore(value_rval, target_lval);
```

value_rval 参数保存着 value 的值，而 target_lval 参数保存的是 target 的目标内存位置。那么一个函数在访问 value 与 target 节点时该如何区分应该保存值还是内存地址呢？这就需要定义左值与右值来解决。左值表示内存地址，可以出现在赋值操作符左侧；右值表示临时值或常量，不能出现在赋值操作符左侧。

由此总结出这几个宏的使用方式：每次调用新的访问函数时，设置期望新函数的返回值类型，新函数调用 IS_EXPECT_ LVAL，根据当前函数被期望返回的值类型，进行不同的流程处理：

以下代码是笔者 assign_stmt_node 访问函数中使用宏设置 target 与 value 返回值类型的实例。用 EXPECT_RVAL 设置 value 返回具体的值；用 EXPECT_LVAL 设置 target 返回物理地址。

```
EXPECT_LVAL(node.target->accept(*this));
llvm::Value* target_lval;
get_result_as_value(&target_lval);
EXPECT_RVAL(node.value->accept(*this));
llvm::Value* value_rval;
get_result_as_value(&value_rval);
```

以下代码是笔者 lval_node 访问函数中数组处理流程中根据数组元素左右值不同进行不同操作的实例。用 IS_EXPECT_LVAL()判断被期待的类型，如果是左值直接返回数组元素的地址，右值就使用 builder.CreateLoad 函数加载数组元素的值并返回加载后的值。

```
if (IS_EXPECT_LVAL()) {
    // 如果期望左值，返回数组元素的地址
    set_value_result(element_ptr);
} else {
    // 如果期望右值，加载数组元素的值并返回加载后的值
    llvm::Value* load_val = builder.CreateLoad(element_ptr->getType()->getPoint
erElementType(), element_ptr, "loadtmp");
    set_value_result(load_val);
```

5 实验总结

本次实验，笔者基于实验一生成的抽象语法树，完善了 SafeCIRBuiler.cpp 文件已有的框架，实现了基于 LLVM 的中间代码生成。还对部分语义错误实现了编译时静态检查和报错。对访问 obc 数组的语句插入用于数组越界访问检测的 IR 代码进行动态检查和报错。

通过本次实验，笔者学习了 LLVM IR 语法描述，并初步掌握了 LLVM IR 生成接口的功能与使用方法。同时还实践了 LLVM IR 静态与动态检查和报错的实现方式。这对笔者以后的学习和科研工作都有很大的帮助。