中国科学院大学网络空间安全学院
编译原理
实验报告

姜俊彦[1], 刘镇豪[2]

目录

[1]学号：2022K8009970011，邮箱：jiangjunyan22@mails.ucas.ac.cn
[2]学号：2022K8009929027，邮箱：liuzhenhao22@mails.ucas.ac.cn

1 实验二

1.1 题目

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。

1.2 工程修改说明

笔者添加了 `Utils.h` 用于定义调试输出宏。除此以外未修改 `SafeCIRBuilder.cpp` 以外的文件。

```
// Utils.h
#define JJY_DEBUG_AST 0
#define JJY_DEBUG_IR 1
#define JJY_DEBUG_OBC_CHECK 1
#define JJY_DEBUG_SIGN "[j] "
```

1.3 实现过程及困难

1.3.1 生成测例的 LLVM IR

在编写 IR 生成相关代码中，笔者认为有以下几点需要核心注意的地方

1. **框架中几个重要函数的理解，搞清楚这些函数的作用与调用时机，可以帮助我们更方便的实现 IR 生成。**
   (a) `EXPECT_RVAL`、`EXPECT_LVAL` 和 `IS_EXPECT_LVAL`：
   这些宏用于管理表达式的值类型（左值/右值）检查：
      i. `EXPECT_RVAL(f)`：期望表达式 `f` 返回右值
      ii. `EXPECT_LVAL(f)`：期望表达式 `f` 返回左值
      iii. `IS_EXPECT_LVAL()`：检查当前是否期望返回左值

   通过维护一个值类型栈来进行表达式求值时的左值/右值检查。
   (b) `enter_scope` 和 `exit_scope`：
   这两个函数通过维护一个作用域栈来实现代码块的作用域进入和退出管理，确保变量的可见性符合语言规范。

(c) `lookup_variable`:

此函数为自主实现，从最近的作用域开始，逐层向外查找变量名，返回第一个匹配的变量信息，如果找不到则返回无效值。

(d) `declare_variable`:

在当前作用域中声明一个新变量，如果变量名已存在则返回失败，否则将变量信息存入并返回成功。

(e) `set_int_result`, `set_value_result`, `get_int_result`, `get_value_result`[1]:

这组函数实现了一个临时结果存取系统，可以存储和获取两种类型的结果（整数和 LLVM Value 指针）。

2. **对 LLVM IR 生成相关 API 的使用，笔者参照了框架中给出的 `build_example.cpp` 与 `example.c`，熟悉并掌握了以下 API 的使用**

(a) 基本块操作:

- `CreateBasicBlock()` - 创建基本块
- `SetInsertPoint()` - 设置插入点
- `GetInsertBlock()` - 获取当前插入点所在基本块

(b) 跳转指令:

- `CreateBr()` - 创建无条件跳转
- `CreateCondBr()` - 创建条件跳转

(c) 内存操作:

- `CreateAlloca()` - 创建局部变量分配
- `CreateStore()` - 创建存储操作
- `CreateLoad()` - 创建加载操作
- `CreateGEP()` - 创建获取元素指针操作

(d) 比较指令:

- `CreateICmpSLT()` - 有符号小于比较
- `CreateICmpSLE()` - 有符号小于等于比较
- `CreateICmpSGT()` - 有符号大于比较
- `CreateICmpSGE()` - 有符号大于等于比较
- `CreateICmpEQ()` - 等于比较

(e) 算术运算:

- `CreateAdd()` - 加法

[1]笔者直到写报告的时候才发现有一个没用上的 `get_result_as_value` 函数，借助此函数可以简化部分实现。

- `CreateSub()` - 减法
- `CreateMul()` - 乘法
- `CreateSDiv()` - 有符号除法
- `CreateSRem()` - 有符号取余
- `CreateNeg()` - 取负
- `CreateOr()` - 逻辑或

(f) 函数相关:

- `CreateCall()` - 创建函数调用
- `CreateRetVoid()` - 创建 void 返回

(g) 常量创建:

- `getInt32()` - 创建 32 位整型常量
- `ConstantInt::get()` - 创建整型常量
- `ConstantDataArray::getString()` - 创建字符串常量

(h) 类型相关:

- `Type::getInt32Ty()` - 获取 32 位整型类型
- `ArrayType::get()` - 获取数组类型

3. **IR 生成实现过程中的困难主要在于表达式、左值的实现**

(a) 表达式求值实现困难主要在于 `SafeCParser.g4` 的语法错误，由于错误的表达式优先级定义，抽象语法树中的一元运算符表达式的优先级过低，导致了诸如`-1+4` 的表达式被错误的解析为`-(1+4)`，而不是正确的 `(-1)+4`。笔者通过修改 `SafeCParser.g4` 中的表达式优先级定义，将一元运算符的优先级提高，解决了这个问题[1]。

(b) 左值的实现困难主要在于对左值的理解，左值是一个可以出现在赋值语句左侧的表达式，可以是变量、数组元素。

i. 对于变量左值，只需要返回变量的指针即可。

ii. 对于数组元素左值，需要返回数组元素的指针，这里需要注意数组元素的指针是一个指向数组元素的指针，而不是数组元素的值。

对应的右值实现则需要使用 `CreateLoad` 返回变量的值或数组元素的值。

(c) 变量定义的实现困难主要在于全局变量与局部变量的区分。

[1]此部分已在 Lab1 的实验报告中提及，这里不再赘述

1.3.2 编译时静态检查

笔者实现了对部分语义错误的编译时静态检查，其大多已自然融入在 IR 生成的实现逻辑中，但笔者在变量定义作用域管理中遇到了一些困难。

笔者通过查找待定义的变量出现在的作用域，判断作用域是否为当前作用域来判断变量是否重复定义。具体代码实现如下

```
// 查找变量是否在当前作用域中定义
VarInfo var_info = lookup_variable( name );
size_t scope_index;
for ( auto it = scoped_variables.rbegin( ); it != scoped_variables.rend(
↪  ); ++it ) {
  if ( it->variable_map.count( name ) ) {
    scope_index = std::distance( it, scoped_variables.rend( ) );
    break;
  }
}
if ( var_info.is_valid( ) && scoped_variables.size( ) == scope_index ) {
  std::cerr << node.line << ":" << node.pos << ": variable '" << name
            << "' is declared more than one times" << std::endl;
  error_flag = true;
  return;
}
```

1.3.3 动态检查和报错

笔者实现了对访问 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。笔者通过在数组访问前插入检查代码，判断数组访问是否越界，如果越界则输出错误信息并终止程序。

其核心逻辑如下

```
if (index < 0 || index >= length) {
  // jmp to check_fail
}
else{
 // jmp to check_success
}
```

对应的 IR 代码实现如下

```
llvm::Value *cmp_zero     = builder.CreateICmpSLT( index, zero_const );
llvm::Value *cmp_length   = builder.CreateICmpSGE( index, length_const );
llvm::Value *cmp          = builder.CreateOr( cmp_zero, cmp_length );
builder.CreateCondBr( cmp, check_fail, check_success );
```

然后只需要将 `obc_check` 函数插入到 obc 数组访问前即可。

1.4 彩蛋样例 trick.c 分析