lab3 实验报告

张艺耀 PB20111630

1. 实验要求

使用访问者模式来实现 IR 的自动生成

2. 实验难点

1. 首先要确定几个全局变量，以存储当前变量的值、地址和当前函数。没有全局变量的定义导致前期实验无从下手。
2. 五种情况下的类型转换

- 赋值时
- 返回值类型和函数签名中的返回类型不一致时
- 函数调用时实参和函数签名中的形参类型不一致时
- 二元运算的两个参数类型不一致时
- 下标计算时

3. 实验设计

3.1全局变量的设计

```
Value *addr;
Value *val;
std::vector<Type *> param_types;
bool pre_enter_scope = false;
int bb = 0;

// function that is being built
Function *cur_fun = nullptr;

// types
Type *VOID_T;
Type *INT1_T;
Type *INT32_T;
Type *INT32PTR_T;
```

```
15  Type *FLOAT_T;
16  Type *FLOATPTR_T;
```

其中`addr`存储当前变量的**地址**；`val`存储当前变量的**值**；`param_types`存储函数参数类型；`pre_enter_scope`表示是否在之前已经进入一个scope；`bb`是基本块的块号。

`cur_fun`用来指示当前分析的函数。

`Type* VOID_T`等是助教提供的全局变量，
在`CminusfBuilder::visit(ASTProgram &*node*)`函数中定义为模块的类型。

3. 2遇到的难点以及解决方案

1. **难点**： 循环判断块的BUG:

```
1  if (ret_val->get_type()->is_integer_type()) {
2          cmp = builder->create_icmp_ne(ret_val,
   CONST_INT(0));
3      } else {
4          cmp = builder->create_fcmp_ne(ret_val,
   CONST_FP(0.));
5      }
```

要对函数返回值的类型进行检验并调用不同的`builder`函数。

2. **难点**： 简单表达式注意遍历顺序:

一开始使用val直接赋值`l_val`和`r_val`而未遍历。

```
1     if(node.additive_expression_r != nullptr) {
    //simple-expression -> additive-expression relop
   additive-expression
2         node.additive_expression_l->accept(*this);
3         auto l_val = val;
4         node.additive_expression_r->accept(*this);
5         auto r_val = val;
```

3. ASTVar的编写:

个人认为是本实验最难的部分。

因为`var`可以是一个整型变量、浮点变量，或者一个取了下标的数组变量，于是要分别编写非数组和数组类型的代码。因为数组的下标值为整型，作为数组下标值的表达式计算结果可能需要类型转换变成整型值，且一个负的下标会导致程序终止，还需要调用框架中的内置函数`neg_idx_except`。若是数组类型则先要判断下标是否大于 0，如果小于 0，则调用所给的neg_idx_except 函数，且调用完后需要根据这一基本块的返回值类型返回相应的值，（对 void、int、float 型进行相应的处理），而不能直接返回调用 neg_idx_except 函数的返回值（void*）

```
//error
builder->set_insert_point(errBB);
auto err_func = scope.find("neg_idx_except");
builder->create_call(err_func, {});
```

赋值语义为：先找到`var`代表的变量地址（如果是数组，需要先对下标表达式求值），然后对右侧的表达式进行求值，求值结果将在转换成变量类型后存储在先前找到的地址中。同时，存储在`var`中的值将作为赋值表达式的求值结果。

```
builder->set_insert_point(correctBB);

        if(array_var->get_type()->get_pointer_element_type()->is_array_type()) {
            addr = builder->create_gep(array_var, {CONST_INT(0), index});
        } else {
            auto array_load = builder->create_load(array_var);
            addr = builder->create_gep(array_load, {index});
        }

        val = builder->create_load(addr);
```

4. ASTCall中的分支：

```
if(fun_type->get_param_type(i)->is_pointer_type())
{
  // is pointer
```

```
    auto load = builder->create_load(arg_addr);
    if(load->get_type()->is_pointer_type()) {
      arguments.push_back(load);
    } else {
      auto ptr = builder->create_gep(arg_addr,
{CONST_INT(0), CONST_INT(0)});
      arguments.push_back(ptr);
    }
  } else {
    // int float
    if(fun_type->get_param_type(i) != arg_val-
>get_type()) {
      if(fun_type->get_param_type(i)-
>is_float_type()) {
        arg_val = builder->create_sitofp(val,
FLOAT_T);
      }
      else if(fun_type->get_param_type(i)-
>is_integer_type()) {
        arg_val = builder->create_fptosi(val,
INT32_T);
      }
    }
    arguments.push_back(arg_val);
  }
```

需要对int float 类型、 指针的类型进行讨论（指向指针的指针或指向数组的指针）要注意前后逻辑关系和类型转换。

H2 实验总结

1. 深入理解了访问者模式和cminusf的语法以及C++的类、成员函数、访问控制、重载等特性。
2. 加深了对IR指令的印象，熟悉了对应的C++ API。
3. 更加得心应手地使用 GDB。因为之前只在OS课程中浅薄地了解了 GDB 调试的用法，本次实验助教贴心地提供了GDB的使用文档，在阅读之后，我花了10分钟完全了解了如何使用GDB调试代码解决一些肉眼难以观察出来的BUG，并借此快速解决了一些BUG。

H2 实验反馈 （可选 不计入评分）

1. 希望可以提供多一点样例，只有一个ASTProgram的样例完全无从下手。
2. 助教可以提醒一下要注意看之前的代码。