Lab4.1 实验报告

实验要求

1. 了解一些代码优化中SSA的相关知识。以及增加对流图和 `phi` 指令的理解，为下一步实验做准备
2. 学习助教的代码，加深对访问者模式的理解，学习如何把优化融入到代码生成中

思考题

Mem2reg

1. 请简述概念：支配性、严格支配性、直接支配性、支配边界。

   支配性：是指如果从流图的入口结点到n，每一条路径都会经过某个结点d，d就支配n

   严格支配性：当且仅当$a \in Dom(n) - n$时，即a是支配n的结点且a不是n，那么a就严格支配n

   直接支配性，在$Dom(n) - n$中，最大的结点直接支配n，即最靠近n的一个结点直接支配n

   支配边界：一个节点n的支配边界可以理解成它支配能力的极限。即流图中，从结点n出发，既有路径可以通过，也有路径不会通过的第一个汇合点，就是n的支配边界。

2. `phi` 节点是SSA的关键特征，请简述 `phi` 节点的概念，以及引入 `phi` 节点的理由。

   `phi` 结点是一种将不同分支中重定值的变量进行合并的操作。

   引入的理由：

   1. 将不同的分支中某个被重定值的变量进行合并，保证了代码的正确性
   2. 可以作为**隐式运行时变量定值**和**显式编译时变量定值**的中间转换, 既解决了编译时的静态单赋值转化问题, 又满足了运行时代码逻辑的正确性.
   3. 编译器可以计算可达定义，接下来编译器可以重命名每个使用的变量和临时值

3. 观察下面给出的 `cminus` 程序对应的 LLVM IR，与**开启** `Mem2Reg` 生成的LLVM IR对比，每条 `load` ， `store` 指令发生了变化吗？变化或者没变化的原因是什么？请分类解释。

```
int globVar;
int func(int x){
    if(x > 0){
        x = 0;
    }
    return x;
}
int main(void){
    int arr[10];
    int b;
    globVar = 1;
    arr[5] = 999;
    b = 2333;
    func(b);
    func(globVar);
    return 0;
}
```

before `Mem2Reg` :

```
@globVar = global i32 zeroinitializer
declare void @neg_idx_except()
define i32 @func(i32 %arg0) {
```

```
label_entry:
  %op1 = alloca i32
  store i32 %arg0, i32* %op1
  %op2 = load i32, i32* %op1
  %op3 = icmp sgt i32 %op2, 0
  %op4 = zext i1 %op3 to i32
  %op5 = icmp ne i32 %op4, 0
  br i1 %op5, label %label6, label %label7
label6:                                           ; preds =
%label_entry
  store i32 0, i32* %op1
  br label %label7
label7:                                           ; preds =
%label_entry, %label6
  %op8 = load i32, i32* %op1
  ret i32 %op8
}
define i32 @main() {
label_entry:
  %op0 = alloca [10 x i32]
  %op1 = alloca i32
  store i32 1, i32* @globVar
  %op2 = icmp slt i32 5, 0
  br i1 %op2, label %label3, label %label4
label3:                                           ; preds =
%label_entry
  call void @neg_idx_except()
  ret i32 0
label4:                                           ; preds =
%label_entry
  %op5 = getelementptr [10 x i32], [10 x i32]* %op0, i32 0, i32 5
  store i32 999, i32* %op5
  store i32 2333, i32* %op1
  %op6 = load i32, i32* %op1
  %op7 = call i32 @func(i32 %op6)
  %op8 = load i32, i32* @globVar
  %op9 = call i32 @func(i32 %op8)
  ret i32 0
}
```

After `Mem2Reg` :

```
@globVar = global i32 zeroinitializer
declare void @neg_idx_except()
define i32 @func(i32 %arg0) {
label_entry:
  %op3 = icmp sgt i32 %arg0, 0
  %op4 = zext i1 %op3 to i32
  %op5 = icmp ne i32 %op4, 0
  br i1 %op5, label %label6, label %label7
label6:                                           ; preds =
%label_entry
  br label %label7
label7:                                           ; preds =
%label_entry, %label6
  %op9 = phi i32 [ %arg0, %label_entry ], [ 0, %label6 ]
  ret i32 %op9
}
define i32 @main() {
label_entry:
```

```
  %op0 = alloca [10 x i32]
  store i32 1, i32* @globVar
  %op2 = icmp slt i32 5, 0
  br i1 %op2, label %label3, label %label4
label3:                                           ; preds =
%label_entry
  call void @neg_idx_except()
  ret i32 0
label4:                                           ; preds =
%label_entry
  %op5 = getelementptr [10 x i32], [10 x i32]* %op0, i32 0, i32 5
  store i32 999, i32* %op5
  %op7 = call i32 @func(i32 2333)
  %op8 = load i32, i32* @globVar
  %op9 = call i32 @func(i32 %op8)
  ret i32 0
}
```

在 `func` 中，所有的load/store指令都消失了，因为在return前被插入了 `phi` 指令，可以直接用最新的左值去返回，这就无需load/store指令了

在 `main` 中，少了2条对b定值的语句，因为引用前已经被定值了，这个值直接就可用了。而全局变量的指令利用改变，因为mem2reg中不对全局变量处理

4. 指出放置phi节点的代码，并解释是如何使用支配树的信息的。（需要给出代码中的成员变量或成员函数名称）

```cpp
// 步骤二：从支配树获取支配边界信息，并在对应位置插入 phi 指令
std::map<std::pair<BasicBlock *, Value *>, bool> bb_has_var_phi; // bb has phi for var
for (auto var : global_live_var_name) {
    std::vector<BasicBlock *> work_list;
    work_list.assign(live_var_2blocks[var].begin(), live_var_2blocks[var].end());
    for (int i = 0; i < work_list.size(); i++) {
        auto bb = work_list[i];
        for (auto bb_dominance_frontier_bb : dominators_->get_dominance_frontier(bb)) {
            if (bb_has_var_phi.find({bb_dominance_frontier_bb, var}) == bb_has_var_phi.end()) {
                // generate phi for bb_dominance_frontier_bb & add bb_dominance_frontier_bb to work list
                auto phi =
                    PhiInst::create_phi(var->get_type()->get_pointer_element_type(), bb_dominance_frontier_bb);
                phi->set_lval(var);
                bb_dominance_frontier_bb->add_instr_begin(phi);
                work_list.push_back(bb_dominance_frontier_bb);
                bb_has_var_phi[{bb_dominance_frontier_bb, var}] = true;
            }
        }
    }
}
```

1. 先获取每一个块的支配边界，通过 `dominators_->get_dominance_frontier(bb)` ，查找对应的支配边界
2. 然后在每一个支配边界中查找是否有这个变量的 `phi` 指令(通过 `bb_has_var_phi`)，如果没有，就插入一条phi指令

5. 算法是如何选择 `value` (变量最新的值)来替换 `load` 指令的？（描述清楚对应变量与维护该变量的位置）

主要是在 `void Mem2Reg::re_name(BasicBlock *bb)` 中

1. 获取每一条 `phi` 指令，作为最新的左值，把它 `push` 到一个(变量，值栈) `var_val_stack` 上
2. load指令用对应 `phi` 指令的左值替换，store指令替换对应 `phi` 指令的左值
3. 需要修改ud链和du链

```
if (!IS_GLOBAL_VARIABLE(l_val) && !IS_GEP_INSTR(l_val)) {
    if (var_val_stack.find(l_val) != var_val_stack.end()) {
        // 此处指令替换会维护 UD 链与 DU 链
        instr->replace_all_use_with(var_val_stack[l_val].back());
        wait_delete.push_back(instr);
    }
}
```

4. 替换完后，需要把栈上的值pop出去，避免影响以后使用

```
for (auto &instr1 : bb->get_instructions()) {
    auto instr = &instr1;

    if (instr->is_store()) {
        auto l_val = static_cast<StoreInst *>(instr)->get_lval();
        if (!IS_GLOBAL_VARIABLE(l_val) && !IS_GEP_INSTR(l_val)) {
            var_val_stack[l_val].pop_back();
        }
    } else if (instr->is_phi()) {
        auto l_val = static_cast<PhiInst *>(instr)->get_lval();
        if (var_val_stack.find(l_val) != var_val_stack.end()) {
            var_val_stack[l_val].pop_back();
        }
    }
}
```

代码阅读总结

1. 对实验代码的结构和算法有了一定的理解。
2. 把课上学的支配性相关的知识串联了起来，不再是简单的伪代码

实验反馈 （可选 不会评分）

对本次实验的建议