Lab4.1 实验报告

实验要求

本次实验需要阅读并理解ssa的意义以及算法的流程，同时理解其中重要概念`phi`函数的含义和使用方法。通过阅读代码加深对于代码优化方法的理解，理解如何添加`phi`结点以及减少load和store指令从而提高程序运行效率。

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：在入口节点为 b0 的流图中，当且仅当 bi 位于从 b0 到 bj 的每条路径上时，结点 bi 支配结点 bj。

   严格支配性：当且仅当$a \in DOM(b) - b$时，a严格支配b。

   直接支配性：在严格支配n的结点集$Dom(n) - n$中与n最接近的结点称为n的直接支配结点。

   支配边界：将相对于n具有以下两个性质的节点m的集合称为n的支配边界：(1) n支配m的一个前趋；(2) n并不严格支配m。

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。

   概念：`phi`结点是用于合并来自CFG不同边的值插入的，其参数是与进入基本程序块的各条边相关联的值的静态单赋值形式名，用于把不同的静态单赋值形式名调和为一个名字。

   引入理由：在CFG中多条边到达同一个汇合点时对于同一个变量名字有许多静态单赋值形式名，表示从不同分支到达这个基本块时一个变量的不同取值，需要引入`phi`结点把这些不同的静态单赋值形式名调和为一个名字。为构造程序的静态单赋值形式，编译器必须向CFG中的汇合点处插入`phi`函数，并且必须重命名变量和临时值，使之符合支配静态单赋值形式名字空间的规则。

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。

   func 函数中的`store i32 %arg0, i32* %op1`,`%op2 = load i32, i32* %op1`,`store i32 0, i32* %op1`,`%op8 = load i32, i32* %op1`被删除了

   main 函数中的`store i32 1, i32* @globVar`,`store i32 999, i32* %op5`,`%op8 = load i32, i32* @globVar`没有改变，`store i32 2333, i32* %op1`,`%op6 = load i32, i32* %op1`被删除了

   其中`store i32 %arg0, i32* %op1`，`store i32 0, i32* %op1`，`store i32 2333, i32* %op1`被删除都是因为store的rval，也即被存入内存的值，被作为 lval 的最新定值，这样store指令就可以删除了。

   而`%op2 = load i32, i32* %op1`，`%op8 = load i32, i32* %op1`和`%op6 = load i32, i32* %op1`被删除是因为load指令被用 lval 最新的定值替代，所以load指令可以删除了。

   `store i32 1, i32* @globVar`，`%op8 = load i32, i32* @globVar`没有被删除都是因为其中lval是全局变量`@globVar`，所以不能删除。

`store i32 999, i32* %op5`没有被删除是因为`%op5 = getelementptr [10 x i32], [10 x i32]* %op0, i32 0, i32 5`，所以其中的lval `%op5` 是 `elementptr`类型的，这种store不能删除。

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）

   放置phi节点的代码：

```
// 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;
                }
            }
        }
    }
```

这段代码先是在外循环中把活跃的全局变量赋给var，然后把之前收集的var所在的block的集合付给work_list. 遍历这些block，每次把当前block赋给`bb`，

之后利用支配树所确定的支配边界，也就是`dominators_->get_dominance_frontier(bb)`返回的`dom_frontier_[bb]`，得到了`dom_frontier_[bb]`，也就是`bb`对应的block的支配边界，在支配边界集合中通过`bb_has_var_phi.find({bb_dominance_frontier_bb, var})`确定支配边界中的每个block中是否有这个var，如果有，就通过

```
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);
```

来插入phi，

然后把`bb_dominance_frontier_bb`加到`work_list`中，并把`bb_has_var_phi[{bb_dominance_frontier_bb, var}]`标记为`true`，

其中主要是使用了支配树中`dom_frontier_`，即支配边界的信息。通过支配边界来确定在哪些block中插入`phi`节点。

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）

   首先建立一个vector `std::vector<Instruction *> wait_delete;` 用于存储所有等待删除的instruction

   之前定义了`std::map<Value *, std::vector<Value *>> var_val_stack;`，用来存储元素和元素对应的值（或`phi`指令）。通过遍历block中所有instruction，把所有的`phi`指令以及`phi`指令的左值都通过`var_val_stack[l_val].push_back(instr);`进行存储，让phi指令的`l_val`对应产生它的所有phi指令的集合。

   之后再次遍历instruction，

```
if (instr->is_load()) {
    auto l_val = static_cast<LoadInst *>(instr)->get_lval();

    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);
        }
    }
}
```

   对于block中的指令instr，通过`is_load()`判断，如果指令是一个load指令，则做上面的操作。**得到load指令的`l_val`，然后判断其是否是global变量或者`GetElementPtrInst *`，如果不是，则可以进行替换，通过`var_val_stack.find(l_val)`来判断是否在var_val_stack中有这个l_val，如果有，则用`var_val_stack[l_val]`栈顶上，也就是最新的phi指令，来替换所有的这条load指令，以及用产生的值替代这条load指令的左值，完成这个功能使用的是函数调用`replace_all_use_with(var_val_stack[l_val].back())`，之后把这条load指令加入`wait_delete`中。**

   这样最后在以下代码段中，之前存入的load指令就会通过`erase_instr(instr)`被删除，也就完成了对`load`指令的替换。

```
// 清除冗余的指令
    for (auto instr : wait_delete) {
        bb->erase_instr(instr);
    }
```

代码阅读总结

本次实验中我理解了ssa的含义和把原来的伪ssa转变为ssa，以及进行代码优化的算法流程。

通过阅读有关代码，我理解了使用light IR来实现这些算法，从而达到代码优化目的的过程。

本次实验让我进一步锻炼了代码阅读能力，通过阅读类的定义来理解代码中许多类的成员和方法的引用，从而理解程序的含义。并同时对与c++编程中vector，map，set，list等数据结构的使用有了更进一步的了解。