Lab4.1 实验报告

实验要求

- 了解lab3中产生的IR可能存在冗余的现象
- 阅读和理解 SSA 和 Mem2Reg Pass
- 学习使用 Mem2Reg Pass
- 阅读相关的代码实现，为后续优化的实现做准备

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：到达 m 的每条路径都经过 n，则称 n 支配 m，记作$n \in Dom(m)$
    - 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：在所有严格支配 m 的节点中，距离 m 最近的那个节点，记作$IDom(m)$
    - 支配边界：满足 n 支配 m 的一个前驱，n 并不严格支配 m 的 m 点的集合，记作$DF(n)$
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
    - 在前面的块中，可能出现同一个变量不同的定值，`phi` 节点用于选择进入接下来的块后使用来自哪一条路径的定值
    - 在部分情况中，来自一些路径的值可能在实际运行的过程中起不到任何作用，引入 `phi` 节点能让我们发现这样的值，并在编译的过程中优化
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
    - 可以观察到在函数中，去除了对函数参数的保存与读取，直接使用参数的值。存储之后又读取的过程显然是冗余的
    - 在进行函数调用时，参数的值只有一处用到，直接省去定义该变量的步骤，在调用时直接使用常数。为只使用一次的变量做分配内存和赋值以及读取操作是冗余的
    - 函数返回时，删除了额外的寄存器，直接使用存储返回值的寄存器，这样的额外存储也是冗余的
    - 引入了 `phi` 节点，省去了部分变量 `load` 和 `store` 的过程，而将其留给 `phi` 节点处理，这正是引入 `phi` 节点的原因
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)
    - 函数 `void Mem2Reg::generate_phi()` 用于放置 `phi` 节点
    - 通过使用类的私有成员:

      ```
      1    std::unique_ptr<Dominators> dominators_;
      ```

      调用函数

      ```
      1    std::set<BasicBlock *> &get_dominance_frontier(BasicBlock *bb);
      ```

获得节点的前驱，并遍历进行相关操作

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）
    - 维护一个栈

    ```
    std::map<Value *, std::vector<Value *>> var_val_stack; // 全局变量初值提前存入栈中
    ```

    - 在遇到一个新的 `value` 后，将其入栈

    ```
    var_val_stack[l_val].push_back(instr);
    ```

    - 发现冗余 `load` ，进行替换

    ```
    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);
    }
    ```

代码阅读总结

- 认识到代码的冗余，亲身体验了代码优化的过程
- 初步了解了部分代码优化的实现
- 学习了一些 `c++` 的 `stl` 操作

实验反馈 （可选 不会评分）

- 阅读理解都好难...