Lab4 实验报告

PB20111633
阿非提

实验要求

- 为下一次实验预备，掌握基础知识
  - SSA 格式的 IR
  - 优化 Pass 的概念
- 理解实验框架代码的实现

思考题

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：结点 a 支配结点 n 当且仅当从开始（源、入口节点）结点到结点n中的所有路径均要经过结点 a。易知每个结点均支配其自身。

   严格支配性：结点 a 严格支配结点 n 当且仅当结点 a 支配结点 n 且 $a \neq n$。

   直接支配性：结点 a 是结点 n 的直接支配结点当且仅当结点 a 严格支配结点 n 且不严格支配任何严格支配结点 n 的其他结点，即严格支配结点 n 的结点集中离结点 n 最近的结点。开始（源、入口节点）结点没有直接支配结点。

   支配边界：结点 a 属于结点 n 的支配边界，当且仅当结点 n 支配结点 a 的前趋且结点 n 不严格支配结点 a。

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。

   ϕ 结点（函数）：在汇合点，根据当前控制流进入的方向选择同名变量的静态单赋值形式名。其值为当前控制流进入基本块时经由的CFG边对应的参数。

   理由：将代码转化成SSA形式时，在控制流汇合点（基本块）中同名变量的值可能根据控制流进入的方向而不同，需要根据控制流进入的方向选择变量的一个静态单赋值形式名的值。

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。

   `func`中：

label_entry中:

store i32 %arg0, i32* %op1
删除，此时 %op1 存储的不是全局变量也不是通过 GEP 指令得到的地址，可标记 %op1 的最新值为 %arg0

%op2 = load i32, i32* %op1
删除，此时 %op1 存在最新值 %arg0，可使用 %op1 替代所有 %op2

label6中:

store i32 0, i32* %op1
删除，此时 %op1 存储的不是全局变量，可标记 %op1 的最新值为 0

label7中:

%op8 = load i32, i32* %op1
指令被 Phi 指令替换，因为 %op1 当前的值根据跳转到label7的基本块的不同可为 %arg0（从label_entry跳转）或 0 （从label6跳转）

main中:
label_entry中:

store i32 1, i32* @globVar
没变，因为 @globVar 为全局变量

label4中:

store i32 999, i32* %op5
没变，因为 %op5 是通过 GEP 指令计算出的地址。

store i32 2333, i32* %op1
删除，此时 %op1 存储的不是全局变量也不是通过 GEP 指令得到的地址，可标记 %op1 的最新值为 2333

%op6 = load i32, i32* %op1
删除，此时 %op1 存在最新值 2333，可使用 2333 替代所有 %op6

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结点的函数为void Mem2Reg::generate_phi()

该函数首先通过遍历函数中的所有 store 指令找出所有 alloca 出的局部变量，并记录他们出现在基本块。再通过对每个上述找出的变量，对所有出现该变量的基本块（bb），通过支配树找出他们的支配边界（dominators_->get_dominance_frontier(bb)），对支配边界中的所有基本块（bb_dominance_frontier_bb）当且仅当还没插入 phi 结点时插入 phi 结点。此时 phi 指令的参数并不完整，phi 可能（存在多个前驱时）会在 void Mem2Reg::re_name(BasicBlock *bb) 补全更多参数。

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）

   算法通过
   std::map<Value *, std::vector<Value *>> var_val_stack 维护每一变量应该取的值（value），即调用 var_val_stack[val].back() 返回 val 的最新值。

   load 指令在 void Mem2Reg::re_name(BasicBlock *bb) 中被替换。该函数首先将在 bb 基本块中出现的所有 phi 指令维护 var_val_stack，使得 phi 指令所对应的变量的最新值为该 phi 指令返回的值。之后算法会按顺序遍历该基本快中所有的指令找出 load 和 store 指令（当且仅当访问对象不为全局变量或数组的指令），对 load 指令，根据 var_val_stack 和 load 指令所读区的内存地址 l_val，如 var_val_stack 中存在对 l_val 所代表的变量的值，则用 l_val 在 var_val_stack 中的最新值（var_val_stack[l_val].back()）替换所有（此时会维护 UD 链与 DU 链）使用该load指令返回值的地方。而对 store 指令，维护 store 指令指向的存储地址 l_val， 所对应的变量的最新值（var_val_stack[l_val].push_back(r_val);）。

代码阅读总结

通过阅读代码，以及附件中所提供的算法为代码，我理解了 Dominators.cpp 所生成的支配树在 Mem2Reg.cpp 应用，认识到算法是如何消除冗余的 store 和 load 指令，并对 SSA 的功能以及它实现有了深刻的理解。