Lab4.1 实验报告

实验要求

1. 理解SSA的含义，理解为什么要使用`phi`函数来构建SSA以及如何通过`phi`函数来构建SSA；
2. 理解如何通过IR的SSA属性来对代码进行优化；
3. 理解文档给出的Men2Reg pass如何将伪SSA IR转化为SSA IR。

思考题

Mem2reg

1. - 支配性：

     > $n \in Dom(m)$，即在入口为a的流图中，当且仅当n位于a到m的每条路径上，n支配m；

   - 严格支配性：

     > 当且仅当$n \in Dom(m) - \{m\}$时，n严格支配m；

   - 直接支配性：

     > 给出流图中的某个节点n，严格支配n的结点集中与n最接近的结点为n的直接支配结点，记作IDom(n)；

   - 支配边界：

     > 给出某结点n，相对于n具有以下两种性质的结点的m集合称为n的支配边界：
     >
     > - n支配m的一个前驱；
     > - n并不严格支配m；

2. - `phi`函数是一个SSA中出现在控制流汇合点，用于记录与合并进入该汇合点所控制程序块的各条边相关联的静态单赋值形式名的函数；
   - 通过引入`phi`函数可以方便编译器具体实现SSA形式——即通过控制流编译器可以方便的使用`phi`函数得到它所需要的正确的变量值来自哪一个基本块，即合并来自不同路径的值，而不需要重新遍历一遍控制流以找到它所需要的值来自哪一次赋值。即LLVM IR可以借助这一函数：

     ```
     %retval = phi i32 [%a, %true], [%b, %false]
     ```

     来实现仅需由刚刚到达的基本块来决定使用%a还是%b，多个参数同样适用。藉由引入`phi`函数可以对代码中大量冗余`load`和`store`指令进行优化。

3. - ```
     store i32 %arg0, i32* %op1
     %op2 = load i32, i32* %op1
     ```

     此处`store`以及`load`指令被优化删去，因为使用了`phi`指令来作为为局部变量alloca出的地址空间的最新定值，然后替换掉`load`指令，然后再将`store`指令将要存入内存的rval作为最新定值以更新`phi`指令，同时可以将`store`删去完成优化：
     ```

```
%op9 = phi i32 [ %arg0, %label_entry ], [ 0, %label6 ]
```

- ```
  store i32 1, i32* @globVar
  ...
  %op8 = load i32, i32* @globVar
  ```

  此处的指令未发生变化，因为需要处理的全局名字集合是以alloca出来的局部变量来统计的，实际操作也限于对局部变量的修改，全局变量的访存正常进行；

- ```
  store i32 999, i32* %op5
  ```

  本次所实现的针对单个简单变量的SSA操作无法使编译器确定指针数组之类复杂访存define和use的具体变量，使得其并不对此作处理。事实上，加入`func(arr[5])`语句时生成的代码也同样会进行访存操作：

  ```
  %op12 = getelementptr [10 x i32], [10 x i32]* %op1, i32 0, i32 5
  %op13 = load i32, i32* %op12
  %op14 = call i32 @func(i32 %op13)
  ```

- ```
  store i32 2333, i32* %op1
  %op6 = load i32, i32* %op1
  ```

  此处`store`以及`load`指令被优化删去，因为变量b可以被常量2333直接代替，与第一种情况其实属于同类型，但此处不需要生成额外的`phi`指令，因为变量b的使用不存在多重赋值，而是直接用常量替换。

4. - 放置`phi`结点的代码：

```
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;
```

`bb_dominance_frontier_bb`是从`Mem2Reg`类中的支配树`dominators_`得到的迭代器，通过它实现对每一个基本块的DF集合（即书中插入`phi`函数的汇合点）的遍历；

```
    for(auto bb_dominance_frontier_bb : dominators_-
>get_dominance_frontier(bb)){};
```

此处使用了成员函数`get_dominance_frontier()`的返回值；
事实上插入后`phi`指令的构建并未完成，还有一个`rename`过程。此处不加赘述。

5. - 首先遍历所有的`store`指令，如果不是指针或是全局变量的操作则加入全局名字集合并进行维护：
```

```
    if (!IS_GLOBAL_VARIABLE(l_val) && !IS_GEP_INSTR(l_val)) {
        global_live_var_name.insert(l_val); //维护全局名字集合
         live_var_2blocks[l_val].insert(bb);
     }
```

- 然后利用`4`所说的插入`phi`结点，并在此过程中为`phi`指令先维护一个左值，也就是真正会被使用的值：

```
    for (auto var : global_live_var_name) {
        ...
        phi->set_lval(var);
        ...
    }
```

在`Instruction.h`中，该操作会维护`phi`指令的`l_val_`：

```
    class PhiInst : public BaseInst<PhiInst> {
        ...
        Value *l_val_;
        void set_lval(Value *l_val) { l_val_ = l_val; }
        ...
    }
```

- 然后利用`phi`指令类的`get_lval()`取出先前从全局名字集合中取出放入的值，存入全局名字初值栈`var_val_stack`中：

```
    if (instr->is_phi()) {
        auto l_val = static_cast<PhiInst *>(instr)->get_lval();
        var_val_stack[l_val].push_back(instr);
    }
```

- 最后对于`load`指令，若其`l_val`与最新定值`var_val_stack.end()`不匹配，则进行替换。以上便是选择`value`替换`load`的全过程。

代码阅读总结

1. 根据与参考材料的对比阅读，理解了对应算法伪代码的精确实现，尤其是IDom与DF集合的创建以及通过后序遍历序列来找到两个块的最邻近前驱的过程；
2. 通过Mem2Reg的阅读掌握了更多c++类中成员函数的用法，并且通过具体代码的阅读能更深刻的理解到`phi`结点的创建与使用，以及如何通过维护UD和DU链方便且清晰地处理掉已经生成的冗余代码；
3. 了解到目前所涉及的SSA模式无法对指针以及全局变量进行处理，后续可能会就此进一步优化？

实验反馈 （可选 不会评分）