Lab4.1 实验报告

张艺耀 PB20111630

实验要求

首先阅读材料及相关代码，了解SSA IR的基本概念和SSA的格式细节等。接下来对照附件伪代码阅读Mem2Reg Pass的代码，学习、理解其代码实现。最后完成思考题。

思考题

Mem2reg

1. **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$m \in DOM(n) - n$时，m严格支配n。

   直接支配性：给出流图中的一个结点n，严格支配n的结点集是$DOM(n) - n$。该集合中与n最接近的结点称为n的直接支配结点，记作$IDOM(n)$。流图的入口结点没有直接支配结点。

   支配边界：对于CFG中的任一节点n中的一个定义，仅在满足以下两个条件的汇合点才需要插入对应的ϕ函数：（1）n支配m的一个前驱。（2）n并不严格支配m。将相对于n具有这种性质的结点m的集合称为n的支配边界，记作$DF(n)$。非正式地，$DF(n)$包含：在离开n的每条CFG路径上，从结点n可达但不支配的第一个结点。

2. phi结点：**引入理由**：为了将一个过程转换为静态单赋值（每个变量仅被赋值一次）形式，编译器必须在具有多个前驱的每个程序块起始处，为当前过程中定义或使用的每个名字y插入一个phi函数，如$y \leftarrow \phi(y, y)$。且必须用下标重命名变量使之符合支配静态单赋值形式名字空间的规则：（1）每个定义名字唯一。（2）每个使用处都引用了一个定义。**概念**：phi函数合并来自不同路径的值，它能够根据不同的路径选择不同的值。phi函数的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。在控制流进入一个基本程序块时，该程序块中的所有phi函数都将并发执行。当前控制流进入基本程序块时经由的CFG边对应的参数即为phi函数的值。

3. 开启`Mem2Reg`前的`func`函数：

```
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
```

开启`Mem2Reg`后的`func`函数：

```
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
```

可以看到`store i32 %arg0, i32* %op1`和`%op2 = load i32, i32* %op1` `store i32 0, i32* %op1` `%op8 = load i32, i32* %op1`被删去了，原因是重复使用x的冗余代码被删除，第一个load和store指令可以直接使用%arg代替，最后使用对来自不同路径的返回值x插入一个**phi函数**作为%op9，这样可以避免两次load x。

开启`Mem2Reg`前的`main`函数：

```
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
}
```

开启`Mem2Reg`后的`main`函数：

```
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
}
```

可以看到`store i32 1, i32* @globVar`这句指令未被删去，原因是`globVar`是一个全局变量，它存在于整个程序的生命周期中，需要显式赋值。`store i32 999, i32* %op5`未被删去，原因是`arr`是一个数组，程序指明了将值存储在此数组中的某个地址。而`store i32 2333, i32* %op1`被删去，因为b仅在函数入口被使用一次，故可以直接使用常量。

4. 放置`phi`结点的代码在`Mem2Reg.cpp`中的`void Mem2Reg::generate_phi()`函数中，代码如下：

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

在外层循环遍历活跃在多个`block`的全局名字集合`global_live_var_name`并把它们所属的bb块集合加入到`work_list`中。第二层循环遍历`work_list`，对于每个其中的block，通过`dominators_->get_dominance_frontier(bb)`得到此结点的支配边界（使用支配树`dominators_`的信息），遍历这些支配边界，如未在`bb_has_var_phi`找到相应条目则插入phi结点。

5. 在`void Mem2Reg::re_name(BasicBlock *bb)`函数中，首先将`phi`指令作为`l_val`的最新定值，`l_val`即是为局部变量alloca出的地址空间，代码如下：

```
for (auto &instr1 : bb->get_instructions()) {
        auto instr = &instr1;
        if (instr->is_phi()) {
            auto l_val = static_cast<PhiInst *>(instr)->get_lval();
            var_val_stack[l_val].push_back(instr);
        }
    }
```

替换`load`指令的代码如下：

```
for (auto &instr1 : bb->get_instructions()) {
        auto instr = &instr1;
        // 步骤四：用 lval 最新的定值替代对应的load指令
        if (instr->is_load()) {
            auto l_val = static_cast<LoadInst *>(instr)->get_lval();
```

```
6
7               if (!IS_GLOBAL_VARIABLE(l_val) &&
    !IS_GEP_INSTR(l_val)) {
8                   if (var_val_stack.find(l_val) !=
    var_val_stack.end()) {
9                       // 此处指令替换会维护 UD 链与 DU
    链
10                      instr-
    >replace_all_use_with(var_val_stack[l_val].back())
    ;
11                      wait_delete.push_back(instr);
12                  }
13              }
14          }
15          ...
16  }
```

变量最新的值由`l_val`指向，变量维护在`var_val_stack`中。如果此变量最新的值不是全局变量地址或`getelementptr`指令，且在`var_val_stack`中存在时，进行指令替换。

代码阅读总结

了解了构造半剪枝静态单赋值形式的具体实现和支配树的建立、如何求支配边界等过程。对面向对象编程有了更深入的理解。

实验反馈 （可选 不会评分）

无