Project 2: Local Value Numbering (LLVM)

(Submitted by Gowtham Tumati, as a part of CS 201 for Winter 2020)

Step 1:

Identified and understood the procedure behind creating a pass in LLVM, on referring the LLVM documentation and the GitHub repository that was shared as a reference.

Step 2:

To avoid confusion during the process, the GitHub repository was cloned and then, the code was added. A function is added to identify the operands and the destination of the expression. Below is the code snippet, to do so.

```
// Fetching the operands, as well as the destination.
auto* ptr = dyn_cast<User>(&inst);
Value* dest = dyn_cast<Value>(&inst);

int total_ops = ptr->getNumOperands();
if(total_ops == 1){
  operand1 = ptr->getOperand(0);
}
if(total_ops == 2){
  operand1 = ptr->getOperand(0);
  operand2 = ptr->getOperand(1);
}

bool *found = new bool(false);
valuenumber1 = addOrFindVN(operand1,found);
valuenumber2 = addOrFindVN(operand2,found);
```

The code snippet mentioned below, does the identification of operations and displays the redundant expressions within the given input block, along with the exact machine representation of the expression.

```
// Computing operations are handled in this block
if(valuenumber1 < valuenumber2){
    expr = to_string(valuenumber1) + operation +  to_string(valuenumber2);
}
else{
    expr = to_string(valuenumber2) + operation +  to_string(valuenumber1);
}
valuenumber3 = addOrFindExpr(expr,found);
if (*found){
  errs() << "Redundant Expression: \n" << inst << "\n";
  *found = false;
}
VNMap.insert(make_pair(dest, valuenumber3));
string output = to_string(valuenumber3) + "=" + to_string(valuenumber1) + operation +  to_string(valuenumber2);
outfile << output << "\n";
```

After combining all the code blocks, it can identify the redundant expressions and perform **Local Value Numbering** and export the same into a separate output file. Details of the obtained output are mentioned in the next parts of the document.

Step 3:

On adding the C function, for LVN implementation, the code was tested against the input files given by the TA.

Output of the file, as a result of running the pass on 1.ll file, and the content of the output file is mentioned below.

```
gowtham@gowthamt-vb:~/Downloads/HelloPass-LLVM-master/test$ opt -load ../Pass/build/libLLVMValueNumberingPass.so -ValueNumbering 1.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

ValueNumbering:
Function: test
   %7 = add nsw i32 %0, %1
   %8 = mul nsw i32 %0, %7
   %9 = add nsw i32 %0, 1
   %10 = add nsw i32 %9, %1
   %11 = add nsw i32 %0, %1
Redundant Expression:
  %11 = add nsw i32 %0, %1
   %12 = mul nsw i32 %0, %11
Redundant Expression:
  %12 = mul nsw i32 %0, %11
   ret void
ValueNumbering:
Function: main
```

```
3=1+2
4=1*3
6=1+5
7=6+2
3=1+2
4=1*3
```

Output of the file, as a result of running the pass on 2.ll file, and the content of the output file is mentioned below.

```
gowtham@gowthamt-vb:~/Downloads/HelloPass-LLVM-master/test$ opt -load ../Pass/build/libLLVMValueNumberingPass.so -ValueNumbering 2.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

ValueNumbering:
Function: test
   %7 = add nsw i32 1, 2
   %8 = add nsw i32 2, 1
Redundant Expression:
  %8 = add nsw i32 2, 1
   %9 = mul nsw i32 1, 2
   %10 = mul nsw i32 2, 1
Redundant Expression:
  %10 = mul nsw i32 2, 1
   %11 = mul nsw i32 %7, %7
   %12 = mul nsw i32 %10, %9
   ret void
ValueNumbering:
Function: main
```

```
3=1+2
3=2+1
4=1*2
4=2*1
5=3*3
6=4*4
```

(ADDITIONAL IMPLEMENTATION)

In addition to the above implementation, there was another approach, that partially was successful. Despite several attempts to combine both, it was not possible due to the time constraint. But, I would like to enclose details of that as well.

The written piece of code, in addition to the original code, identifies the exact expressions that are redundant in the given input block. Inserted below, is the algorithm, on whose basis, the code was based.

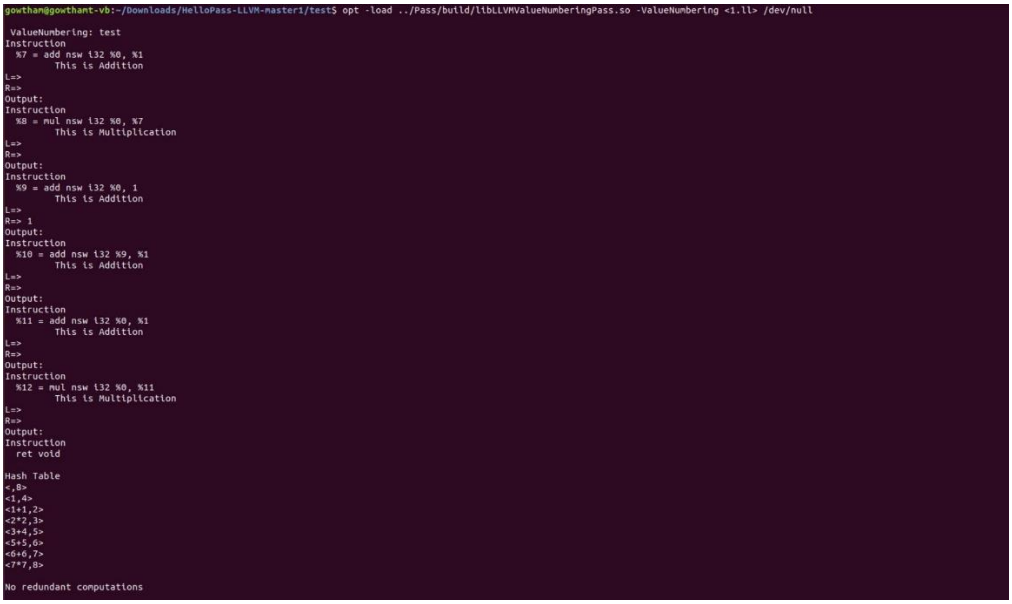
```
for each operation in form T ← L op R
   lookup hash table for value numbers of L and R: VN(L) and VN(R)
   if not found
      insert <L,VN(L)> or/and  <R,VN(R)>

   lookup hash table for a hash key VN(L) op VN(R)
   if found
      /* needs a reversed hash table */
      replace this operation with a copy operation
   else
      insert <VN(L) op VN(R),VN(VN(L) op VN(R))>
   insert <T, VN(VN(L) op VN(R))>
```

With the help of the algorithm, I was successful in identifying the redundant expressions and displaying them directly in the terminal itself, without the necessity of writing them to an output file. Attached below is the implemented code snippet, and output snapshots when the code is tested against the two sample inputs provided.

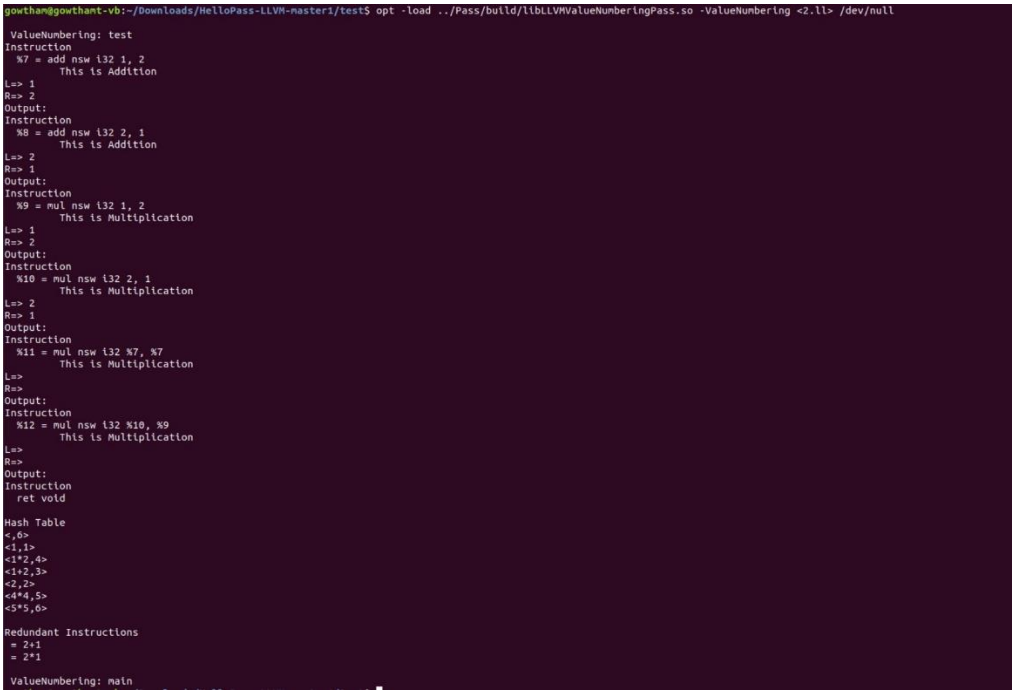
```
// Search hashtable for values numbers L and R and add non occuring VN
int x, y;
if (VNMap.find(L) == VNMap.end()) {
  VNMap[L] = ValueNumber+1;
  ValueNumber+=1;
}
x = VNMap[L];
if (VNMap.find(R) == VNMap.end()) {
  VNMap[R] = ValueNumber+1;
  ValueNumber+=1;
}
y = VNMap[R];
string exp;
string expression;
string rev_expression;
if (inst.getOpcode() == Instruction::Add) {
  exp = to_string(x)+"+"+to_string(y);
  expression = L+"+"+R;
  rev_expression = to_string(y)+"+"+to_string(x);
} else if (inst.getOpcode() == Instruction::Sub) {
  exp = to_string(x)+"-"+to_string(y);
  expression = L+"-"+R;
  rev_expression = to_string(y)+"-"+to_string(x);
} else if (inst.getOpcode() == Instruction::Mul) {
  exp = to_string(x)+"*"+to_string(y);
  expression = L+"*"+R;
  rev_expression = to_string(y)+"*"+to_string(x);
} else if (inst.getOpcode() == Instruction::SDiv) {
  exp = to_string(x)+"/"+to_string(y);
  expression = L+"/"+R;
  rev_expression = to_string(y)+"/"+to_string(x);
} else {
  exp = to_string(x);
}

if (VNMap.find(exp) != VNMap.end()) {
  redundantExpr.push_back(outputVal+" = "+expression);
  VNMap[outputVal] = VNMap[exp];
} else if(VNMap.find(rev_expression) != VNMap.end()) {
  redundantExpr.push_back(outputVal+" = "+expression);
  VNMap[outputVal] = VNMap[rev_expression];
} else {
  VNMap[exp] = ValueNumber+1;
  VNMap[outputVal] = ValueNumber+1;
  ValueNumber+=1;
}
```

Output of the code implementation:

For Testcase 1, no redundant expressions were identified.



For Testcase 2, there are two redundant expressions identified.



-THE END-