Pass-LLVM-Value-Numbering

Writing a pass to find all the LVN-identifiable redundancy. Project 2 from CS201 UCR.

Setup(From HelloPass)

1. Clone this repo, see the file details below:

- Pass - root directory for build and source code
    - Transforms is top level directory for project.
    - CMakeLists.txt file is CMakeLists for the Project.
        - ValueNumbering is top level directory for the Pass
            - ValueNumbering.cpp contains code for the Pass
            - CMakeLists.txt is CMakeLists for this Pass
    - build will build Pass in this directory
- test contains Test code

2. For Mac OSX users, uncomment the following line in Pass/Transforms/ValueNumbering/CMakeLists.txt

```
SET(CMAKE_MODULE_LINKER_FLAGS "-undefined dynamic_lookup")
```

3. Move to Pass/**build**/ directory using cd command on your local system. Next, execute the following command. If it executes successfully, proceed to next step.

```
cmake -DCMAKE_BUILD_TYPE=Release ../Transforms/ValueNumbering
```

4. Next execute make and it will generate *.so files under build directory.

```
make -j4
```

5. Move to **test**/ directory and generate ***Test.ll*** file for Test.c using following command.

```
clang -Xclang -disable-O0-optnone Test.c -O0 -S -emit-llvm -o Test.ll
```

6. Next generate ***Test.bc*** file for Test.ll using following command.

```
opt Test.ll -mem2reg -S -o Test.bc
```

7. After generating test.bc, execute following command it execute the LLVM Pass.

```
opt -load ../Pass/build/libLLVMValueNumberingPass.so  -ValueNumbering < Test.bc
> /dev/null
```

Code Explanation

1. The implemented Pass extends from `FunctionPass` class and overrides `runOnFunction(Function &F)` function. And it basically finds the redundancy expression in basic block.
2. In this implement, I used two hash_table, the first one is to store the operand of each expression, the second one is to store the expression in string format.

   ```
   std::map<llvm::Value*, int> hashmap;
   std::map<std::string, int> expressionmap;
   ```

3. For hashmap which stores the operands. There is a function to judge whether this operand in the hashmap or not. And for each operand, it'll have one unique value number.

   ```
   int searchHash(Value* v, bool found){
     auto search = hashmap.find(v);
     auto temp = vn;
     // errs() << v;
     if(search != hashmap.end()){
       temp = search->second;
       found = true;
       errs() << "true\t";
     }
     else{
       found = false;
       errs() << "false\t";
       hashmap.insert(make_pair(v, vn++));
     }
     return temp;
   }
   ```

4. Also, there would be the situation that the expression is same but the operands are in different places. So I swap these two value numbers. And use the value numbers and the operation symbol to represent the expression.

```
  op_1 = searchHash(op1, found);
  op_2 = searchHash(op2, found);

  // errs() << to_string(op_1) << "\t" << to_string(op_2) << "\n";

  if(op_1 > op_2){
    swap(op_1, op_2);
  }

  string expression = to_string(op_1) + operation + to_string(op_2);
```

5. If the expression is in the expression map, it will return true, means this expression is redundant.

```
  bool searchExpression(string exp){
    auto search = expressionmap.find(exp);
    auto temp = vn_expression;
    errs() << "\n" << exp << "\n";
    if(search != expressionmap.end()){
      return true;
    }
    else{
      expressionmap.insert(make_pair(exp, vn_expression++));
      return false;
    }
  }
```

Tests

1. For the c program below:

```
void test(int a, int b, int c, int d, int e, int f) {
  c = a + b;
  d = a * c;
  e = a;
  a = a + 1;
  d = a + b;
  f = e + b;
  f = e * f;
}

int main() {
  test(1,2,3,4,5,6);
  return 0;
}
```

The result is :

```
➜ test opt -load ../Pass/build/libLLVMValueNumberingPass.so  -ValueNumbering < Test.bc > /dev/null
ValueNumbering:
  %7 = add nsw i32 %0, %1
false   false
1+2
  %8 = mul nsw i32 %0, %7
true    false
1*3
  %9 = add nsw i32 %0, 1
true    false
1+4
  %10 = add nsw i32 %9, %1
false   true
2+5
  %11 = add nsw i32 %0, %1
true    true
1+2
This Expression is Redundancy:  %11 = add nsw i32 %0, %1
  %12 = mul nsw i32 %0, %11
true    false
1*6
  ret void
ValueNumbering:
  call void @test(i32 1, i32 2, i32 3, i32 4, i32 5, i32 6)
  ret i32 0
```

2. For c program below:

```c
void test(int a, int b, int c, int d, int e, int f) {
  a = 1;
  b = 2;
  c = a + b;
  d = b + a;
  e = a * b;
  f = b * a;
  c = c * c;
  f = f * e;
}

int main() {
  test(1,2,3,4,5,6);
  return 0;
}
```

The result is:

```
➜ test opt -load ../Pass/build/libLLVMValueNumberingPass.so  -ValueNumbering < Test.bc > /dev/null
ValueNumbering:
  %7 = add nsw i32 1, 2
false   false
1+2
  %8 = add nsw i32 2, 1
true    true
1+2
This Expression is Redundancy:  %8 = add nsw i32 2, 1
  %9 = mul nsw i32 1, 2
true    true
1*2
  %10 = mul nsw i32 2, 1
true    true
1*2
This Expression is Redundancy:  %10 = mul nsw i32 2, 1
  %11 = mul nsw i32 %7, %7
false   true
3*3
  %12 = mul nsw i32 %10, %9
false   false
4*5
  ret void
ValueNumbering:
  call void @test(i32 1, i32 2, i32 3, i32 4, i32 5, i32 6)
  ret i32 0
```