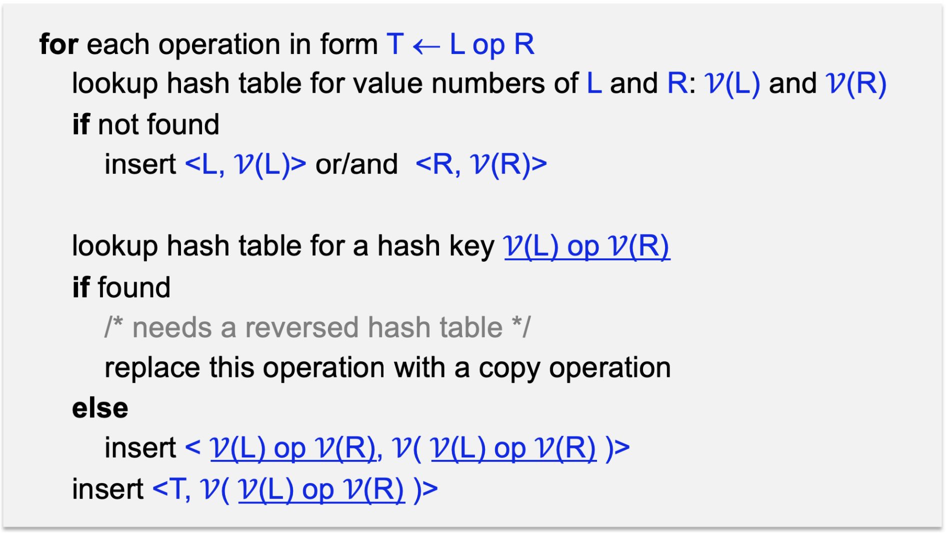
Trung Lam - 861270734
CS201 - Project 2
13 February, 2022

Project 2 - Local Value Numbering (LLVM)

Algorithm/Data Structures:
Data structure used to formulate the hash table for the algorithm in the source code is an unordered map, with a string as the key, and the value numbering integer as the value. The algorithm for parsing the store and load instruction are similar, in a way that first, parse the operands as strings and assign or replace both operands with the same value number and insert them into the hash table. For the binary operation instructions, the algorithm is very much similar to the Value Numbering lecture slides:

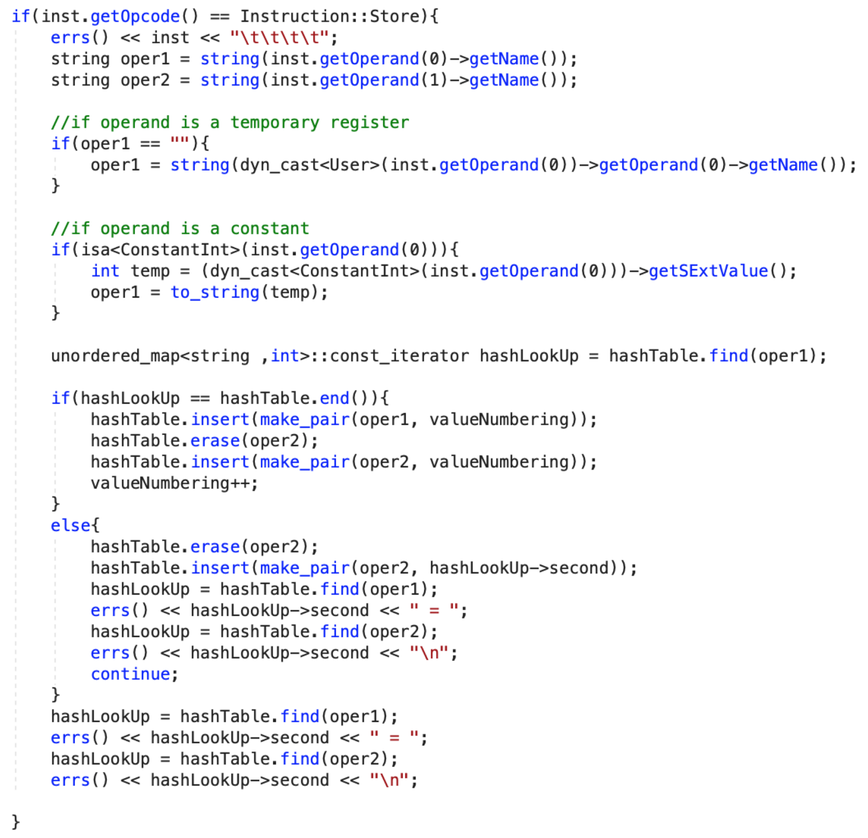


```
for each operation in form $T \leftarrow L$ op $R$
    lookup hash table for value numbers of $L$ and $R$: $\mathcal{V}(L)$ and $\mathcal{V}(R)$
    if not found
        insert $<L, \mathcal{V}(L)>$ or/and  $<R, \mathcal{V}(R)>$

    lookup hash table for a hash key $\underline{\mathcal{V}(L)\ op\ \mathcal{V}(R)}$
    if found
        /* needs a reversed hash table */
        replace this operation with a copy operation
    else
        insert $<\underline{\mathcal{V}(L)\ op\ \mathcal{V}(R)}, \mathcal{V}(\ \underline{\mathcal{V}(L)\ op\ \mathcal{V}(R)}\ )>$
    insert $<T, \mathcal{V}(\ \underline{\mathcal{V}(L)\ op\ \mathcal{V}(R)}\ )>$
```

With the exception of not being able to implement a reversed hash table, and reverse the final if-else statement such that instead of "if found or else", the algorithm is "if not found or else."

Implementation:
Before iterating through each instruction of a basic block, initialize the value numbering as 1. After which, begin to process each information depending on whether they are a store, load, or a binary operation instruction.

Store:

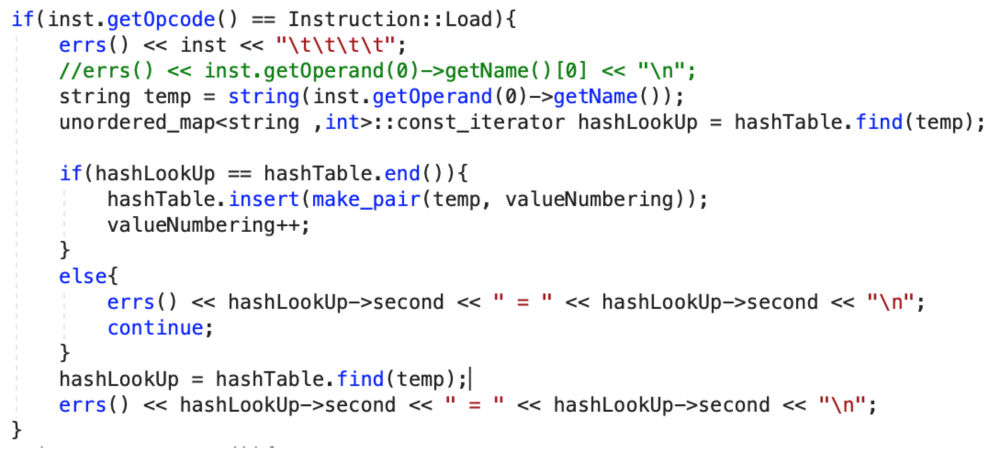


```
if(inst.getOpcode() == Instruction::Store){
    errs() << inst << "\t\t\t\t";
    string oper1 = string(inst.getOperand(0)->getName());
    string oper2 = string(inst.getOperand(1)->getName());

    //if operand is a temporary register
    if(oper1 == ""){
        oper1 = string(dyn_cast<User>(inst.getOperand(0))->getOperand(0)->getName());
    }

    //if operand is a constant
    if(isa<ConstantInt>(inst.getOperand(0))){
        int temp = (dyn_cast<ConstantInt>(inst.getOperand(0)))->getSExtValue();
        oper1 = to_string(temp);
    }

    unordered_map<string ,int>::const_iterator hashLookUp = hashTable.find(oper1);

    if(hashLookUp == hashTable.end()){
        hashTable.insert(make_pair(oper1, valueNumbering));
        hashTable.erase(oper2);
        hashTable.insert(make_pair(oper2, valueNumbering));
        valueNumbering++;
    }
    else{
        hashTable.erase(oper2);
        hashTable.insert(make_pair(oper2, hashLookUp->second));
        hashLookUp = hashTable.find(oper1);
        errs() << hashLookUp->second << " = ";
        hashLookUp = hashTable.find(oper2);
        errs() << hashLookUp->second << "\n";
        continue;
    }
    hashLookUp = hashTable.find(oper1);
    errs() << hashLookUp->second << " = ";
    hashLookUp = hashTable.find(oper2);
    errs() << hashLookUp->second << "\n";

}
```

Description:

First, retrieve the operands and assign the first operand as oper1 and the second as oper2. After which, check a few cases of the operand. Usually if an operand is a temporary register, the string would return as an empty string. So if that happens, cast the operand as a User class to retrieve the operand of whatever the register was assigned to and retrieve the name of that operand. Another case is when the operand is a constant integer. In this case, check to see if the operand is an integer. If it is, cast the operand as a ConstantInt class and use the function getSExtValue() to retrieve whatever integer the operand is and cast that integer as a string. After which, start the value numbering implementation. The store instruction usually sets the first operand equal to the second, so there is no need to check whether the second operand is in the hash table or not since it will be overwritten by whatever numbering the first operand will be. First, find oper1. If it is not found in the hash table, insert the oper1 along with a value numbering, VN. Erase the current oper2, if it exists in the hash table, and give it the same VN. After which, increment the VN. However, if oper1 is found, then the only thing to do is replace the VN of the current oper2 with the VN associated with oper1.

Load:

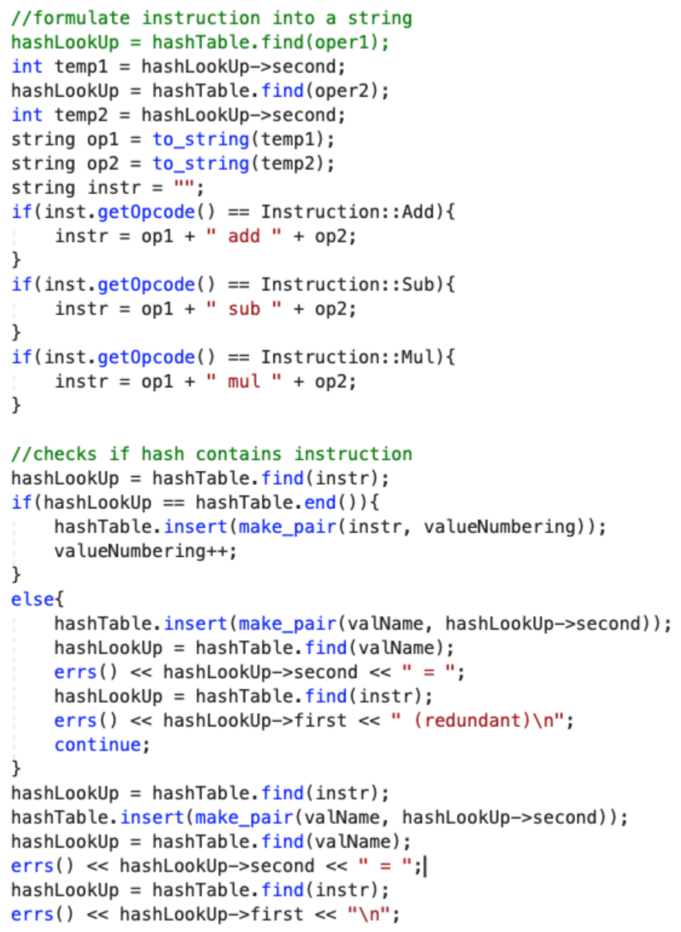


```
if(inst.getOpcode() == Instruction::Load){
    errs() << inst << "\t\t\t\t";
    //errs() << inst.getOperand(0)->getName()[0] << "\n";
    string temp = string(inst.getOperand(0)->getName());
    unordered_map<string ,int>::const_iterator hashLookUp = hashTable.find(temp);

    if(hashLookUp == hashTable.end()){
        hashTable.insert(make_pair(temp, valueNumbering));
        valueNumbering++;
    }
    else{
        errs() << hashLookUp->second << " = " << hashLookUp->second << "\n";
        continue;
    }
    hashLookUp = hashTable.find(temp);
    errs() << hashLookUp->second << " = " << hashLookUp->second << "\n";
}
```

Description:

Load is mostly loading the values into a temporary register. So the only thing to do is check to see whether the value being loaded exists in the hash table. Parse the operand into a string and check if that operand exists in the hash table. If it does, then do nothing. Otherwise, insert the value along with a VN.

Binary:



```
//formulate instruction into a string
hashLookUp = hashTable.find(oper1);
int temp1 = hashLookUp->second;
hashLookUp = hashTable.find(oper2);
int temp2 = hashLookUp->second;
string op1 = to_string(temp1);
string op2 = to_string(temp2);
string instr = "";
if(inst.getOpcode() == Instruction::Add){
    instr = op1 + " add " + op2;
}
if(inst.getOpcode() == Instruction::Sub){
    instr = op1 + " sub " + op2;
}
if(inst.getOpcode() == Instruction::Mul){
    instr = op1 + " mul " + op2;
}

//checks if hash contains instruction
hashLookUp = hashTable.find(instr);
if(hashLookUp == hashTable.end()){
    hashTable.insert(make_pair(instr, valueNumbering));
    valueNumbering++;
}
else{
    hashTable.insert(make_pair(valName, hashLookUp->second));
    hashLookUp = hashTable.find(valName);
    errs() << hashLookUp->second << " = ";
    hashLookUp = hashTable.find(instr);
    errs() << hashLookUp->first << " (redundant)\n";
    continue;
}
hashLookUp = hashTable.find(instr);
hashTable.insert(make_pair(valName, hashLookUp->second));
hashLookUp = hashTable.find(valName);
errs() << hashLookUp->second << " = ";
hashLookUp = hashTable.find(instr);
errs() << hashLookUp->first << "\n";
```

Description:

The bulk of code before the sample piece above is extremely similar to the implementation of previous instructions: parse operands into strings and check if those operands are in the hash table. After retrieving the operands as strings, formulate the strings to the format of $operand1\ op\ operand2$. With the op dependent on whether the instruction is Add, Mul, or Sub. After which, run the algorithm. If the formulated instr is not found in the hash table, then insert instr along with a value number. After that, increment it. Else, or if it is found, insert T or valName along the VN associated with the formulate instr to the value numbering.