03/25/2024 Shreya Gaur

CS502: COMPILER SYSTEMS AND PROGRAMMING LANGUAGES

PROGRAMMING ASSIGNMENT 4

Introduction

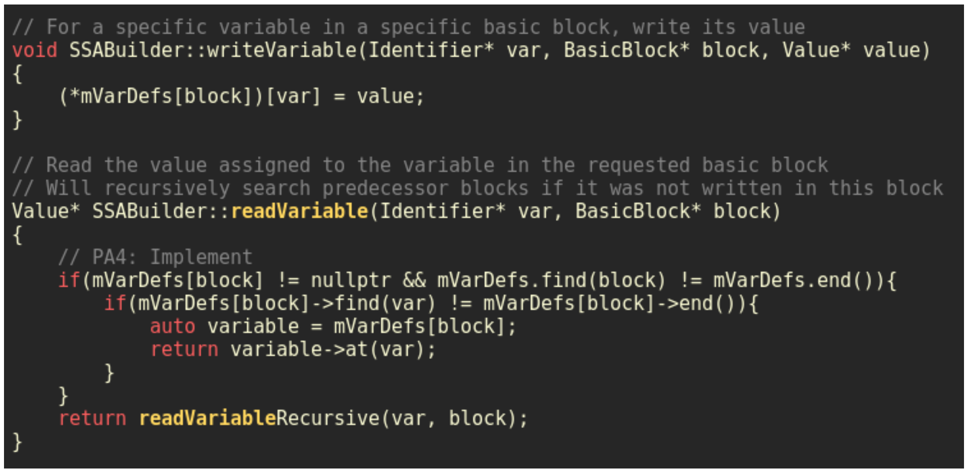
Single Static Assignment (SSA) is an intermediate representation used to simply the analysis and optimization of the high-level code. In SSA Form, each variable is defined only one and every use of the variable refers to the version of the variable assigned to it. This property makes it easier to perform certain optimizations, such as constant propagation, dead code elimination and register allocation.

The conversion of llvm IR to SSA form is the main objective of this assignment. The most popular algorithm for SSA form creation is Cytron's Algorithm, however it has several drawbacks for compilers that are entirely SSA-based. In this algorithm, the input program must be represented as a control flow graph (CFG) in non-SSA form. Hence, if the compiler wants to construct SSA from the input language (be it given by an abstract syntax tree or some bytecode format), it has to take a detour through a non-SSA CFG in order to apply Cytron et al.'s algorithm. Furthermore, to guarantee the minimality of the ϕ function placement, Cytron et al.'s algorithm relies on several other analyses and transformations. To calculate the locations where ϕ functions have to be placed, it computes a dominance tree and the iterated dominance frontiers. To avoid placing dead ϕ functions, liveness analyses or dead code elimination has to be performed. Both requiring a CFG and relying on other analyses make it inconvenient to use this algorithm in an SSA-centric compiler.

Due to the drawbacks posed by Cytron's algorithm, we instead use the algorithm described in paper "Simple and Efficient Construction of Static Single Assignment Form" by Braun et al. In this algorithm, only when a variable is used, we query its reaching definition. If it is unknown at the current location, we will search backwards through the program. We insert ϕ functions at join points in the CFG along the way, until we find the desired definition. We employ memoization to avoid repeated look-ups.

Local Value Numbering

Local value numbering is the process of renaming variables within the same basic block. We go through the sequence of instructions in basic block in program order and record each variable definitions in map mVarDefs. When we encounter a user of the variable, we query the mVarDefs map for its most recent definition. After the local value numbering is done for a basic block, we call the basic block filled. Below are the functions readVariable() and writeVariables() which perform writes to mVarDefs and reads from mVarDefs as we progress to the instructions in the basic block.

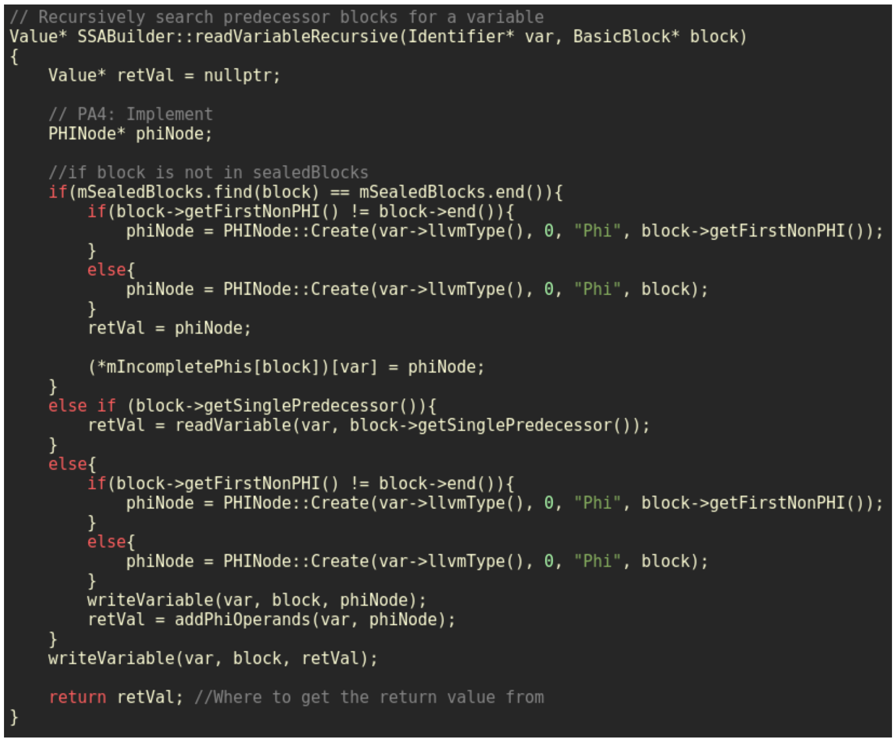


```
// For a specific variable in a specific basic block, write its value
void SSABuilder::writeVariable(Identifier* var, BasicBlock* block, Value* value)
{
    (*mVarDefs[block])[var] = value;
}

// Read the value assigned to the variable in the requested basic block
// Will recursively search predecessor blocks if it was not written in this block
Value* SSABuilder::readVariable(Identifier* var, BasicBlock* block)
{
    // PA4: Implement
    if(mVarDefs[block] != nullptr && mVarDefs.find(block) != mVarDefs.end()){
        if(mVarDefs[block]->find(var) != mVarDefs[block]->end()){
            auto variable = mVarDefs[block];
            return variable->at(var);
        }
    }
    return readVariableRecursive(var, block);
}
```

Global Value Numbering

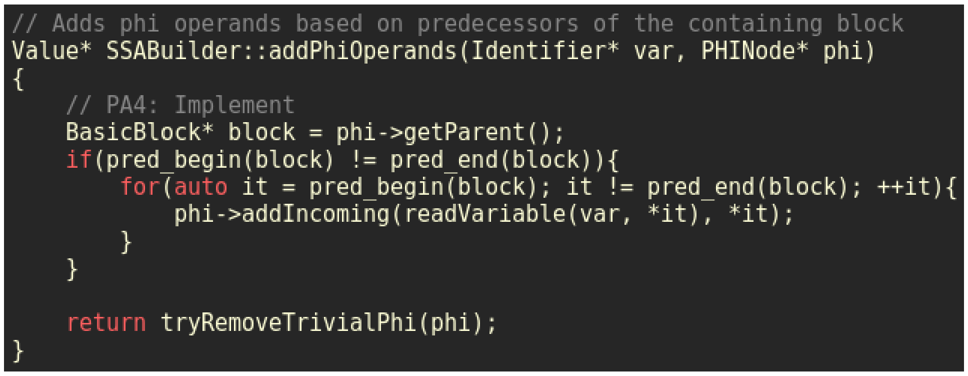
Local value numbering does not handle cases in which the variable's definition is found on the path between the root and the current basic block under consideration. Moreover, there can be multiple definitions reaching a use of the variable. To extend local value numbering to these cases, we must implement global value numbering.

When the definition of a given variable in a basic block is not found in the same basic block, we look for the variable’s definition in the predecessors of the current basic block. These different definitions are then combined into a single definition of the variable using a ϕ function.



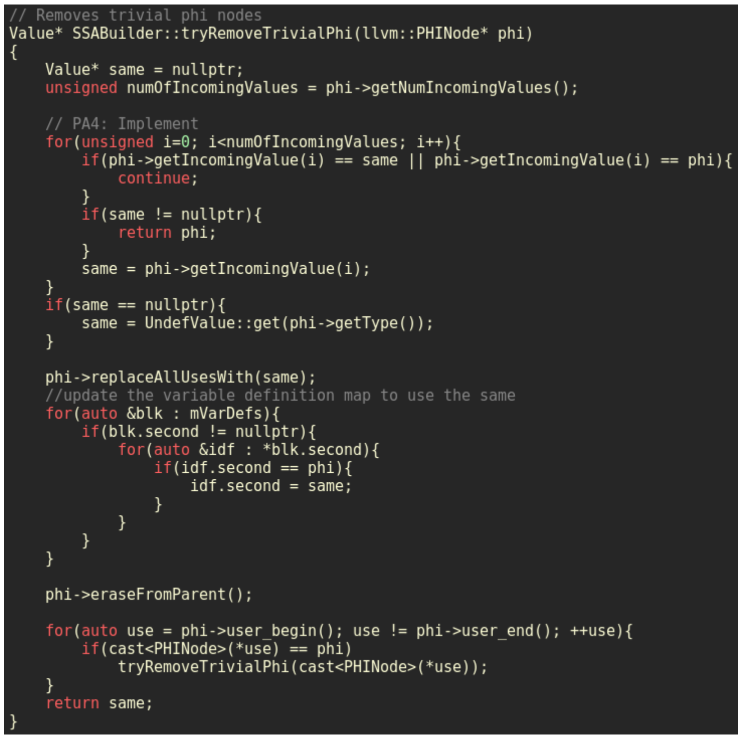
```
// Recursively search predecessor blocks for a variable
Value* SSABuilder::readVariableRecursive(Identifier* var, BasicBlock* block)
{
    Value* retVal = nullptr;

    // PA4: Implement
    PHINode* phiNode;

    //if block is not in sealedBlocks
    if(mSealedBlocks.find(block) == mSealedBlocks.end()){
        if(block->getFirstNonPHI() != block->end()){
            phiNode = PHINode::Create(var->llvmType(), 0, "Phi", block->getFirstNonPHI());
        }
        else{
            phiNode = PHINode::Create(var->llvmType(), 0, "Phi", block);
        }
        retVal = phiNode;

        (*mIncompletePhis[block])[var] = phiNode;
    }
    else if (block->getSinglePredecessor()){
        retVal = readVariable(var, block->getSinglePredecessor());
    }
    else{
        if(block->getFirstNonPHI() != block->end()){
            phiNode = PHINode::Create(var->llvmType(), 0, "Phi", block->getFirstNonPHI());
        }
        else{
            phiNode = PHINode::Create(var->llvmType(), 0, "Phi", block);
        }
        writeVariable(var, block, phiNode);
        retVal = addPhiOperands(var, phiNode);
    }
    writeVariable(var, block, retVal);

    return retVal; //Where to get the return value from
}
```

The readVariableRecursive() is used to handle the recursive reading of the variable in the preceding basic blocks. The creation of the ϕ function also happens in the readVariableRecursive() when an incomplete CFG is detected or there are multiple predecessors of the same basic block. The addPhiOperands() function is used to add the operands in these newly created ϕ functions.



```
// Adds phi operands based on predecessors of the containing block
Value* SSABuilder::addPhiOperands(Identifier* var, PHINode* phi)
{
    // PA4: Implement
    BasicBlock* block = phi->getParent();
    if(pred_begin(block) != pred_end(block)){
        for(auto it = pred_begin(block); it != pred_end(block); ++it){
            phi->addIncoming(readVariable(var, *it), *it);
        }
    }

    return tryRemoveTrivialPhi(phi);
}
```

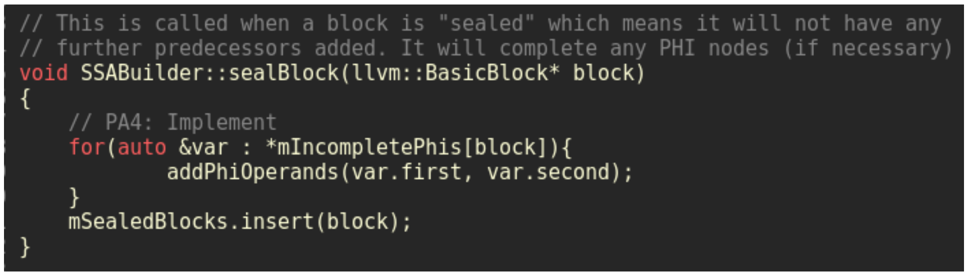
The algorithm can create trivial ϕ functions. A ϕ function is considered trivial if and only if it references itself and just another variable. The uses of such a ϕ function are also considered trivial. There trivial ϕ functions are removed from the parent basic block. A special is of the ϕ function that only references itself. In such a case the operand of the ϕ function is replaced with undefined value. The removal of these trivial ϕ functions is done by the function tryRemoveTrivialPhi().



```
// Removes trivial phi nodes
Value* SSABuilder::tryRemoveTrivialPhi(llvm::PHINode* phi)
{
    Value* same = nullptr;
    unsigned numOfIncomingValues = phi->getNumIncomingValues();

    // PA4: Implement
    for(unsigned i=0; i<numOfIncomingValues; i++){
        if(phi->getIncomingValue(i) == same || phi->getIncomingValue(i) == phi){
            continue;
        }
        if(same != nullptr){
            return phi;
        }
        same = phi->getIncomingValue(i);
    }
    if(same == nullptr){
        same = UndefValue::get(phi->getType());
    }

    phi->replaceAllUsesWith(same);
    //update the variable definition map to use the same
    for(auto &blk : mVarDefs){
        if(blk.second != nullptr){
            for(auto &idf : *blk.second){
                if(idf.second == phi){
                    idf.second = same;
                }
            }
        }
    }

    phi->eraseFromParent();

    for(auto use = phi->user_begin(); use != phi->user_end(); ++use){
        if(cast<PHINode>(*use) == phi)
            tryRemoveTrivialPhi(cast<PHINode>(*use));
    }
    return same;
}
```

Handling Incomplete Control Flow Graphs

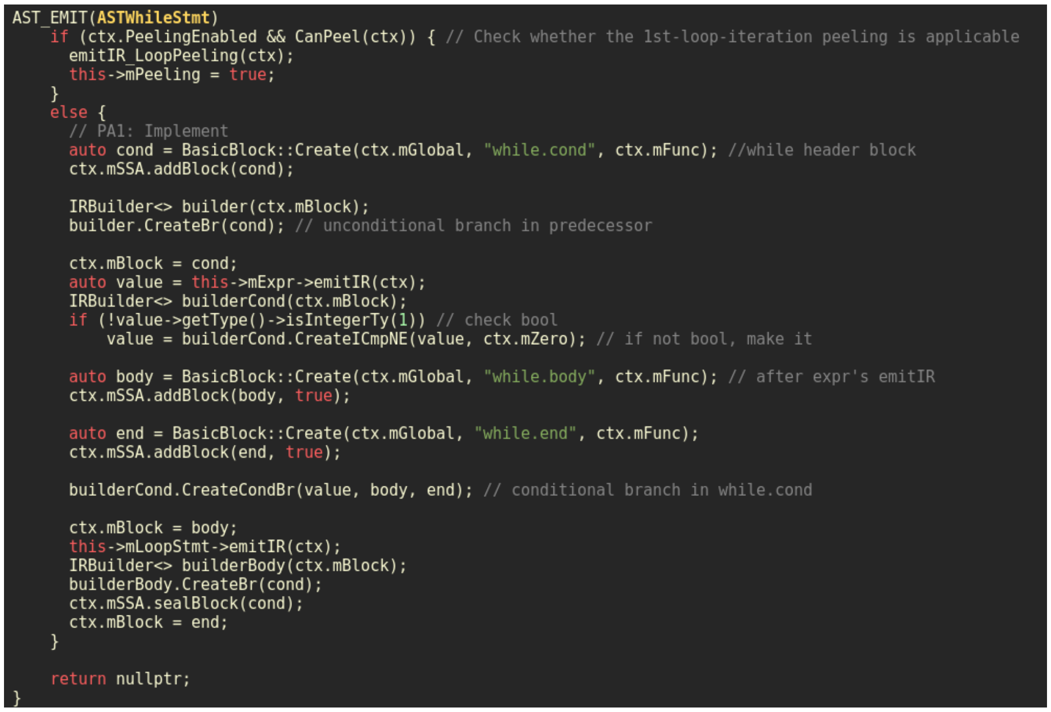
A sealBlock() function is used to handle the case of incomplete CFGs. Such a case happens when during the pass, all the predecessors of a given basic block have not been visited. A good example is the while.body basic block in a while loop. The while.body basic block has a backward edge to the start of the loop in a CFG due to the unconditional jump at the end of the while loop body. We cannot consider the while.body basic block a predecessor of the while.cond basic block until the while.body is visited. Such a case is handled using the sealBlock() function which calls addOperandPhi when all the predecessors of the basic block have been visited. The sealBlock() function helps complete the incomplete ϕ functions in the given basic block.



```
// This is called when a block is "sealed" which means it will not have any
// further predecessors added. It will complete any PHI nodes (if necessary)
void SSABuilder::sealBlock(llvm::BasicBlock* block)
{
    // PA4: Implement
    for(auto &var : *mIncompletePhis[block]){
            addPhiOperands(var.first, var.second);
    }
    mSealedBlocks.insert(block);
}
```

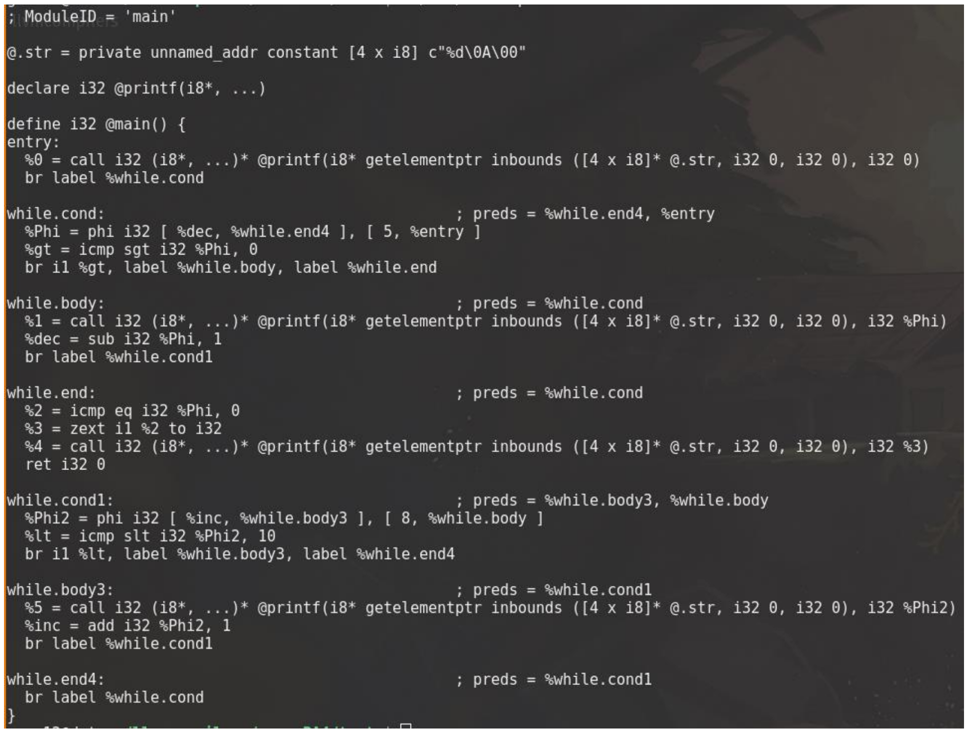
Integration with ASTEmit and Testing

The above functions are integrated with the ASTEmit.cpp file. As the different AST Nodes are emitted, the SSABuilder class functions are called to include ϕ functions and complete them. Important part of this integration is the additions of addBlock() and sealBlock() functions in the case of ASTIfStmt and ASTWhileStmt. The same can be seen in the code below:



```
AST_EMIT(ASTWhileStmt)
    if (ctx.PeelingEnabled && CanPeel(ctx)) { // Check whether the 1st-loop-iteration peeling is applicable
      emitIR_LoopPeeling(ctx);
      this->mPeeling = true;
    }
    else {
      // PA1: Implement
      auto cond = BasicBlock::Create(ctx.mGlobal, "while.cond", ctx.mFunc); //while header block
      ctx.mSSA.addBlock(cond);

      IRBuilder<> builder(ctx.mBlock);
      builder.CreateBr(cond); // unconditional branch in predecessor

      ctx.mBlock = cond;
      auto value = this->mExpr->emitIR(ctx);
      IRBuilder<> builderCond(ctx.mBlock);
      if (!value->getType()->isIntegerTy(1)) // check bool
          value = builderCond.CreateICmpNE(value, ctx.mZero); // if not bool, make it

      auto body = BasicBlock::Create(ctx.mGlobal, "while.body", ctx.mFunc); // after expr's emitIR
      ctx.mSSA.addBlock(body, true);

      auto end = BasicBlock::Create(ctx.mGlobal, "while.end", ctx.mFunc);
      ctx.mSSA.addBlock(end, true);

      builderCond.CreateCondBr(value, body, end); // conditional branch in while.cond

      ctx.mBlock = body;
      this->mLoopStmt->emitIR(ctx);
      IRBuilder<> builderBody(ctx.mBlock);
      builderBody.CreateBr(cond);
      ctx.mSSA.sealBlock(cond);
      ctx.mBlock = end;
    }

    return nullptr;
}
```

After adding all the code, the following are the results of emit07.usc testcase:

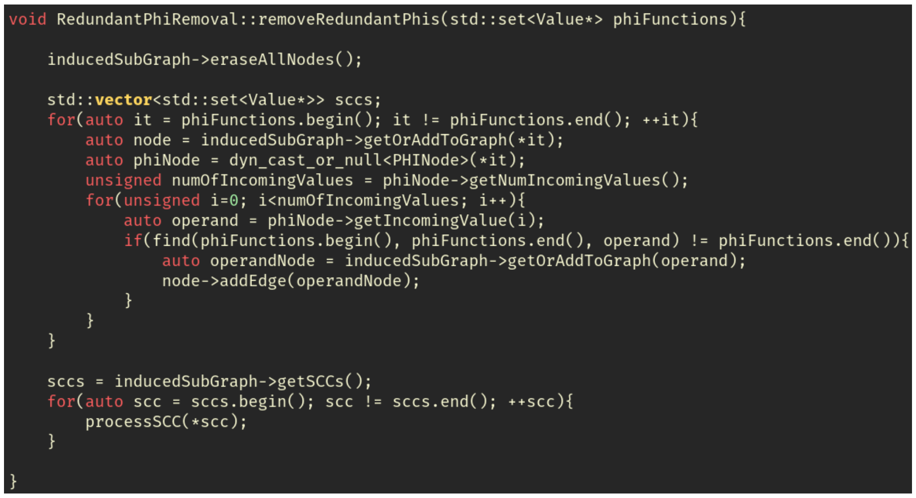


```
; ModuleID = 'main'

@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00"

declare i32 @printf(i8*, ...)

define i32 @main() {
entry:
  %0 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str, i32 0, i32 0), i32 0)
  br label %while.cond

while.cond:                                       ; preds = %while.end4, %entry
  %Phi = phi i32 [ %dec, %while.end4 ], [ 5, %entry ]
  %gt = icmp sgt i32 %Phi, 0
  br i1 %gt, label %while.body, label %while.end

while.body:                                       ; preds = %while.cond
  %1 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str, i32 0, i32 0), i32 %Phi)
  %dec = sub i32 %Phi, 1
  br label %while.cond1

while.end:                                        ; preds = %while.cond
  %2 = icmp eq i32 %Phi, 0
  %3 = zext i1 %2 to i32
  %4 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str, i32 0, i32 0), i32 %3)
  ret i32 0

while.cond1:                                      ; preds = %while.body3, %while.body
  %Phi2 = phi i32 [ %inc, %while.body3 ], [ 8, %while.body ]
  %lt = icmp slt i32 %Phi2, 10
  br i1 %lt, label %while.body3, label %while.end4

while.body3:                                      ; preds = %while.cond1
  %5 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str, i32 0, i32 0), i32 %Phi2)
  %inc = add i32 %Phi2, 1
  br label %while.cond1

while.end4:                                       ; preds = %while.cond1
  br label %while.cond
}
```

As it can be seen from the output above, the phi functions are introduced at the correct places in the IR. The code also passes all testSSA.py test cases.

Redundant Phi Removal

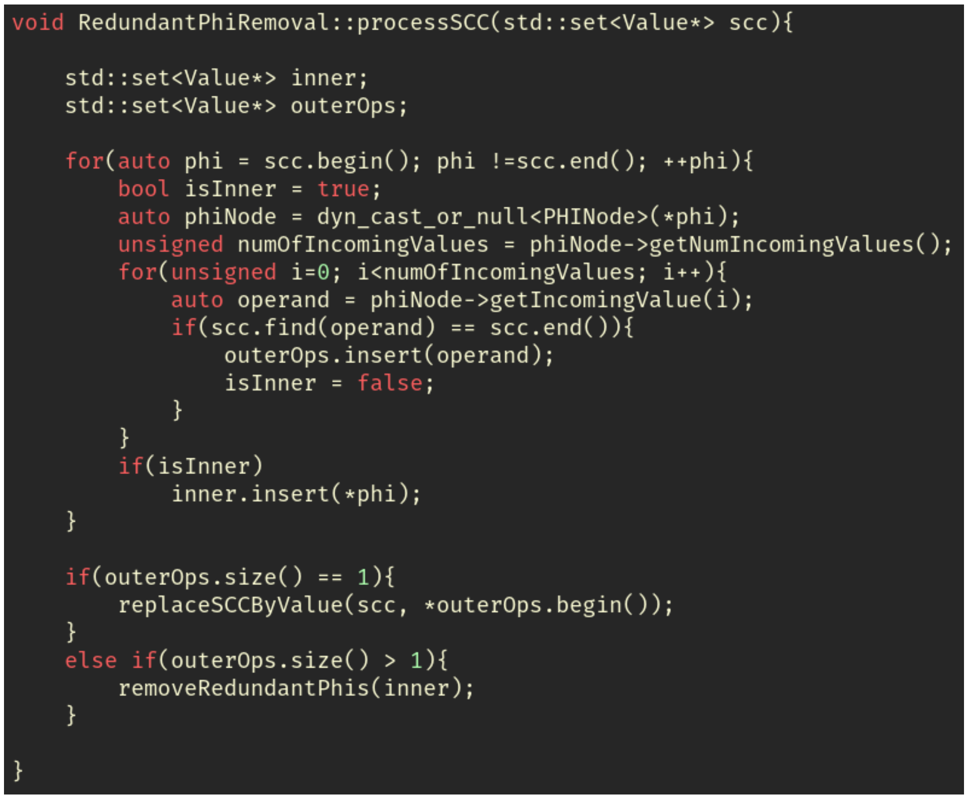
The given ϕ functions are redundant if the ϕ functions only refer to each other or one other value. A set of redundant ϕ functions P always contains Strongly connected components that are also redundant. The same idea is exploited in order to find redundant ϕ functions in the code sequence. The removeRedundantPhi() function below is used to leverage to do the same.



```
void RedundantPhiRemoval::removeRedundantPhis(std::set<Value*> phiFunctions){

    inducedSubGraph->eraseAllNodes();

    std::vector<std::set<Value*>> sccs;
    for(auto it = phiFunctions.begin(); it != phiFunctions.end(); ++it){
        auto node = inducedSubGraph->getOrAddToGraph(*it);
        auto phiNode = dyn_cast_or_null<PHINode>(*it);
        unsigned numOfIncomingValues = phiNode->getNumIncomingValues();
        for(unsigned i=0; i<numOfIncomingValues; i++){
            auto operand = phiNode->getIncomingValue(i);
            if(find(phiFunctions.begin(), phiFunctions.end(), operand) != phiFunctions.end()){
                auto operandNode = inducedSubGraph->getOrAddToGraph(operand);
                node->addEdge(operandNode);
            }
        }
    }

    sccs = inducedSubGraph->getSCCs();
    for(auto scc = sccs.begin(); scc != sccs.end(); ++scc){
        processSCC(*scc);
    }

}
```

The function first created a data flow graph filled using all the phi functions present in the phiFunctions. The edges are attached from phi function and its operands. The strongly connected components of the graph are put into vector and sccs.

Function processSCC() is called on every element of the sccs. ProcessSCC() function is used to get set of phi nodes which contain operands outside the SCCs and which do not, in sets outerOps and inner respectively. The redundant ϕ are replaced using replaceSCCByValue() when there is only one element outerOps.



```cpp
void RedundantPhiRemoval::processSCC(std::set<Value*> scc){

    std::set<Value*> inner;
    std::set<Value*> outerOps;

    for(auto phi = scc.begin(); phi !=scc.end(); ++phi){
        bool isInner = true;
        auto phiNode = dyn_cast_or_null<PHINode>(*phi);
        unsigned numOfIncomingValues = phiNode->getNumIncomingValues();
        for(unsigned i=0; i<numOfIncomingValues; i++){
            auto operand = phiNode->getIncomingValue(i);
            if(scc.find(operand) == scc.end()){
                outerOps.insert(operand);
                isInner = false;
            }
        }
        if(isInner)
            inner.insert(*phi);
    }

    if(outerOps.size() == 1){
        replaceSCCByValue(scc, *outerOps.begin());
    }
    else if(outerOps.size() > 1){
        removeRedundantPhis(inner);
    }

}
```

Conclusion

The test cases from both testSSA.py and testPhi.py are passing completely.