```
////////////////////////////////////////////////////////////////////////////////
// 15-745 S14 Assignment 3
// Group: bhumbers, psuresh
////////////////////////////////////////////////////////////////////////////////

#ifndef __CLASSICAL_DATAFLOW_LICM_H__
#define __CLASSICAL_DATAFLOW_LICM_H__

#include <deque>

#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/InstIterator.h"
#include "llvm/ADT/SmallPtrSet.h"

#include "llvm/Pass.h"
#include "llvm/Analysis/LoopInfo.h"
#include "llvm/ADT/ValueMap.h"
#include "llvm/ADT/SmallVector.h"

#include "llvm/Analysis/ValueTracking.h"

#include "dataflow.h"
#include "reaching-defs.h"

#include <iomanip>
#include <queue>
#include <map>

using namespace llvm;
using namespace std;

namespace {

/** Runs LICM on a particular function
 * Note that this borrows from LLVM's LoopPass & LPPassManager, in that we run optimizations on each loop
 * in the function. However, this FunctionPass was used so that a reaching definition analysis could be executed
 * on the whole function before the per-loop transforms. */
class LoopInvariantCodeMotion : public FunctionPass {
 public:
  static char ID;

  LoopInvariantCodeMotion();

  bool doInitialization(Module& M);
  virtual bool runOnFunction(Function& F);
  virtual void getAnalysisUsage(AnalysisUsage& AU) const;

 protected:
  deque<Loop *> LQ;

  /** Returns the set of blocks which are part of the given loop and which have at least one successor outside the loop */
  SmallPtrSet<BasicBlock*, 32> getLoopExits(Loop* L);

  /** Returns block dominance info using dataflow framework */
  DataFlowResult computeDominance(Loop* L);

  /** Computes immediate dominance info given dataflow results with basic dominance info */
  map<BasicBlock*, BasicBlock*> computeImmediateDominance(DataFlowResult dominanceResults);

  void printDominanceInfo(DataFlowResult dominanceResults, map<BasicBlock*, BasicBlock*> immDoms);

  /** Returns set of statements (instructions) in given loop which are considered loop invariant */
  set<Value*> computeLoopInvariantStatements(Loop* L, map<Value*, ReachingDefinitionInfo> reachingDefs);

  /** Returns the set of statements (instructions) in given loop which are valid candidates for movement to loop preheader according t
o LICM*/
  set<Value*> computeCodeMotionCandidateStatements(Loop* L, DataFlowResult dominanceResults, set<Value*> invariantStatements);

  /** Applies LICM to given candidates where possible (basically, if all dependencies have also been moved).
   * Returns true if any motions were applied, which modifies the loop code */
  bool applyMotionToCandidates(Loop* L, set<Value*> motionCandidates);

  /** Recurse through all subloops and all loops  into LQ. (Source: LoopPass.cpp) */
  void addLoopIntoQueue(Loop* L);
};

char LoopInvariantCodeMotion::ID = 0;
RegisterPass<LoopInvariantCodeMotion> X("cd-licm", "15-745 Loop Invariant Code Motion");

}

#endif
```

```
////////////////////////////////////////////////////////////////////////////////
// 15-745 S14 Assignment 3
// Group: bhumbers, psuresh
////////////////////////////////////////////////////////////////////////////////

#include "loop-invariant-code-motion.h"

#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/InstIterator.h"
#include "llvm/ADT/SmallPtrSet.h"

#include "llvm/Pass.h"
#include "llvm/Analysis/LoopInfo.h"
#include "llvm/ADT/ValueMap.h"
#include "llvm/ADT/SmallVector.h"

#include "llvm/Analysis/ValueTracking.h"

#include "dataflow.h"

#include <iomanip>
#include <queue>
#include <map>

using namespace llvm;
using namespace std;

namespace {

//////////////////////////////////////////////////////////////////////////////////////////////
//Dataflow analyses

//DOMINANCE
class DominanceDataFlow : public DataFlow {
  protected:
    BitVector applyMeet(std::vector<BitVector> meetInputs) {
      BitVector meetResult;

      //Meet op = intersection of inputs
      if (!meetInputs.empty()) {
        for (int i = 0; i < meetInputs.size(); i++) {
//          errs() << "  " << bitVectorToStr(meetInputs[i]) << ", ";
          if (i == 0)
            meetResult = meetInputs[i];
          else
            meetResult &= meetInputs[i];
        }
      }
//      errs() << "\n";

      return meetResult;
    }

    TransferResult applyTransfer(const BitVector& value, DenseMap<Value*, int> domainEntryToValueIdx, BasicBlock* block) {
      TransferResult transfer;
      transfer.baseValue = value;

//      errs() << "Applying transfer for block: " << block->getName() << "\n";
//      errs() << "Pre: " << bitVectorToStr(value) << "\n";

      //Transfer of dominance is simple: Just add the current block to the dominance set
      unsigned blockIdx = domainEntryToValueIdx[block];
      transfer.baseValue.set(blockIdx);

//      errs() << "Post: " << bitVectorToStr(transfer.baseValue) << "\n";

      return transfer;
    }
};

//Helper for checking dominance. Returns true if A dominates B according to given results
bool dominates(BasicBlock* A, BasicBlock* B, DataFlowResult dominanceResults) {
//  errs() << "Checking whether " << A->getName() << " dominates " << B->getName() << "...";
  DataFlowResultForBlock dominanceOfB = dominanceResults.resultsByBlock[B];
  bool aDomsB = dominanceOfB.out[dominanceResults.domainEntryToValueIdx[A]];
//  errs() << (aDomsB ? "YES" : "NO") << "\n";
  return aDomsB;
}

//////////////////////////////////////////////////////////////////////////////////////////////

LoopInvariantCodeMotion::LoopInvariantCodeMotion() : FunctionPass(ID) { }

bool LoopInvariantCodeMotion::doInitialization(Module& M) {
  return false;
```

```
}

bool LoopInvariantCodeMotion::runOnFunction(Function& F) {
  bool modified = false;

  //Get reaching definitions at each program point over whole function
  map<Value*, ReachingDefinitionInfo> reachingDefs = ReachingDefinitions().computeReachingDefinitions(F);

  //Add all loops into the processing queue. Note that addLoopIntoQueue will recursively add subloops of each
  // top-level loop in front of the parent loop, so that processing will be from most-to-least nested order.
  // This helps guarantee that any loop invariant code motion will "bubble out" to the outer most loop.
  LoopInfo& LI = getAnalysis<LoopInfo>();
  for (LoopInfo::reverse_iterator I = LI.rbegin(), E = LI.rend(); I != E; ++I)
    addLoopIntoQueue(*I);

  //Apply LICM to each loop in the work queue
  while (!LQ.empty()) {
    Loop* L  = LQ.back();

    //Don’t bother with loops without a preheader
    if (L->getLoopPreheader() == NULL)
      return false;

    DataFlowResult dominanceResults = computeDominance(L);
    map<BasicBlock*, BasicBlock*> immDoms = computeImmediateDominance(dominanceResults);
    printDominanceInfo(dominanceResults, immDoms);

    set<Value*> loopInvariantStatements = computeLoopInvariantStatements(L, reachingDefs);

    set<Value*> codeMotionCandidateStatements = computeCodeMotionCandidateStatements(L, dominanceResults, loopInvariantStatements);

    bool loopModified = applyMotionToCandidates(L, codeMotionCandidateStatements);

    modified |= loopModified;

    LQ.pop_back();
  }

  return modified;
}

void LoopInvariantCodeMotion::getAnalysisUsage(AnalysisUsage& AU) const {
  AU.addRequired<LoopInfo>();
}

SmallPtrSet<BasicBlock*, 32> LoopInvariantCodeMotion::getLoopExits(Loop* L) {
  SmallVector<BasicBlock*, 32> loopSuccessors;
  L->getUniqueExitBlocks(loopSuccessors);

  SmallPtrSet<BasicBlock*, 32> loopExits;
  for (SmallVector<BasicBlock*, 32>::iterator i = loopSuccessors.begin(); i < loopSuccessors.end(); ++i) {
    //Note: As a result of the loop-simplify pass, each out-of-loop successor’s sole predecessor should be part of this loop
    loopExits.insert(*pred_begin(*i));
  }

  return loopExits;
}

DataFlowResult LoopInvariantCodeMotion::computeDominance(Loop* L) {
  //Dataflow domain = Set of all basic blocks in the loop (as well as their parents)
  std::set<BasicBlock*> blocksSet;
  std::vector<BasicBlock*> loopBlocks = L->getBlocks();
  for (std::vector<BasicBlock*>::iterator blockIter = loopBlocks.begin(); blockIter != loopBlocks.end(); ++blockIter) {
    BasicBlock* block = *blockIter;
    //Add parents
    for (pred_iterator predBlock = pred_begin(block), E = pred_end(block); predBlock != E; ++predBlock) {
      blocksSet.insert(*predBlock);
    }
    blocksSet.insert(block); //Add block
  }
  std::vector<Value*> domain;
  std::vector<BasicBlock*> blocks;
  for (std::set<BasicBlock*>::iterator it = blocksSet.begin(); it != blocksSet.end(); ++it) {
//    errs() << "Adding to domain for dominance: " << (*it)->getName() << "\n";
    blocks.push_back(*it);
    domain.push_back(*it);
  }

  int numVars = domain.size();

  //Boundary value at entry is just the entry block (entry dominates itself)
  BitVector boundaryCond(numVars, false);

  //Initial interior set is full set of blocks
  BitVector initInteriorCond(numVars, true);

  //Get dataflow values at IN and OUT points of each block
  DominanceDataFlow flow;
  return flow.run(blocks, domain, DataFlow::FORWARD, boundaryCond, initInteriorCond);
}

map<BasicBlock*, BasicBlock*> LoopInvariantCodeMotion::computeImmediateDominance(DataFlowResult dominanceResults) {
  map<BasicBlock*, BasicBlock*> immDoms;

  //We find the immediate dominators in a somewhat less-than-optimally-efficient way: basically,
  //for each block B, walk up the graph toward the root of the CFG in a BFS ordering until we see a node in dom(B)
  //There appear to be better idom algorithms, but I wasn’t sure how to make them work nicely with our dataflow framework.

  for (map<BasicBlock*, DataFlowResultForBlock>::iterator resultsIter = dominanceResults.resultsByBlock.begin();
         resultsIter != dominanceResults.resultsByBlock.end();
         ++resultsIter) {
    DataFlowResultForBlock& blockResult = resultsIter->second;
    BitVector visited(dominanceResults.resultsByBlock.size(), false);
    std::queue<BasicBlock*> work;
    work.push(resultsIter->first);
    while (!work.empty()) {
      BasicBlock* currAncestor = work.front();
      work.pop();
      int currIdx = dominanceResults.domainEntryToValueIdx[currAncestor];
      visited.set(currIdx);

//      errs() << "Checking if idom of block " << resultsIter->first->getName() << " is " << currAncestor->getName() << "\n";

      //If ancestor is contained in dom set for the results block, mark as idom and quit
      if (blockResult.in[currIdx]) {
        immDoms[resultsIter->first] = currAncestor;
        break;
      }

      for (pred_iterator predBlock = pred_begin(currAncestor), E = pred_end(currAncestor); predBlock != E; ++predBlock) {
        int predIdx = dominanceResults.domainEntryToValueIdx[*predBlock];
        if (!visited[predIdx]) {
          work.push(*predBlock);
        }
      }
    }
  }

  return immDoms;
}

void LoopInvariantCodeMotion::printDominanceInfo(DataFlowResult dominanceResults, map<BasicBlock*, BasicBlock*> immDoms) {
  //Output: Print immediate dominance information
  errs() << "Dominance domain: {";
  for (map<BasicBlock*, DataFlowResultForBlock>::iterator resultsIter = dominanceResults.resultsByBlock.begin();
         resultsIter != dominanceResults.resultsByBlock.end();
         ++resultsIter) {
    errs() << resultsIter->first->getName() << "  ";
  }
  errs() << "}\n";

  errs() << "\nImmediate Dominance Relationships: \n";

  for (map<BasicBlock*, DataFlowResultForBlock>::iterator resultsIter = dominanceResults.resultsByBlock.begin();
         resultsIter != dominanceResults.resultsByBlock.end();
         ++resultsIter) {
    char str[100];
    BasicBlock* idom = immDoms[resultsIter->first];
    if (idom) {
      sprintf(str, "%s is idom’d by %s", ((std::string)resultsIter->first->getName()).c_str(), ((std::string)idom->getName()).c_str())
;
      errs() << str << "\n";
    }
    else {
      sprintf(str, "%s has no idom", ((std::string)resultsIter->first->getName()).c_str());
      errs() << str << "\n";
    }
//      sprintf(str, "Dominators for %-20s:", ((std::string)resultsIter->first->getName()).c_str());
//      errs() << str << bitVectorToStr(resultsIter->second.in) << "\n";
  }

  errs() << "\n";
}

std::set<Value*>  LoopInvariantCodeMotion::computeLoopInvariantStatements(Loop* L, map<Value*, ReachingDefinitionInfo> reachingDefs) {
  std::set<Value*> loopInvariantStatements;

  std::vector<BasicBlock*> loopBlocks = L->getBlocks();

  //Initialize invariant statement set
  for (std::vector<BasicBlock*>::iterator blockIter = loopBlocks.begin(); blockIter != loopBlocks.end(); ++blockIter) {
    for (BasicBlock::iterator instIter = (*blockIter)->begin(), e = (*blockIter)->end(); instIter != e; ++instIter) {
      Value* v = instIter;
      //First, check if this is an easy invariance case
      if (isa<Constant>(v) || isa<Argument>(v) || isa<GlobalValue>(v))
        loopInvariantStatements.insert(v);

      //Otherwise, check more complex conditions for typical instructions:
      //Statement A=B+C+D+... is invariant if all the reaching defs for all its operands (B, C, D, ...) are outside the loop
      //(and a few other misc safety conditions are met)
      else if (isa<Instruction>(v)) {
        Instruction* I = static_cast<Instruction*>(v);

//        errs() << "Considering invariance of: " << valueToStr(v) << "\n";

        bool mightBeLoopInvariant = (isSafeToSpeculativelyExecute(I) && !I->mayReadFromMemory() && !isa<LandingPadInst>(I));

        if (mightBeLoopInvariant) {
          bool allOperandsOnlyDefinedOutsideLoop = true;

          for (User::op_iterator opIter = I->op_begin(), e = I->op_end(); opIter != e; ++opIter) {
            Value* opVal = *opIter;
            ReachingDefinitionInfo varDefsInfo = reachingDefs[opVal];

            vector<Value*> varDefsAtStatement = varDefsInfo.defsByPoint[I];
            for (int i = 0; i < varDefsAtStatement.size(); i++) {
              if (isa<Instruction>(varDefsAtStatement[i])) {
                if (L->contains(((Instruction*)varDefsAtStatement[i])->getParent()))  {
                  allOperandsOnlyDefinedOutsideLoop = false;
                  break;
                }
              }
            }

            if (!allOperandsOnlyDefinedOutsideLoop)
              break;
          }

          if (allOperandsOnlyDefinedOutsideLoop)
            loopInvariantStatements.insert(v);
        }
      }
    }
  }

  //Iteratively update invariant statement set until convergence
  //(since invariant will grow monotonically, we detect this simply by seeing if it stops growing)
  bool converged = false;
  int invariantSetSize = loopInvariantStatements.size();
  while (!converged) {
    int prevInvariantSetSize = invariantSetSize;

    //Check through all statements in the loop, adding statement A=B+C+D+... to the invariant set if
    //all operands B,C,... have a single reaching definition at that statement AND those definitions are loop-invariant

    for (std::vector<BasicBlock*>::iterator blockIter = loopBlocks.begin(); blockIter != loopBlocks.end(); ++blockIter) {
      for (BasicBlock::iterator instIter = (*blockIter)->begin(), e = (*blockIter)->end(); instIter != e; ++instIter) {
        Value* v = instIter;

//        errs() << "Considering invariance of: " << valueToDefinitionVarStr(v) << "\n";

        //If already known to be invariant, skip checking again
        if (loopInvariantStatements.find(v) != loopInvariantStatements.end())
          continue;

        if (isa<Instruction>(v)) {
          Instruction* I = static_cast<Instruction*>(v);

          bool mightBeLoopInvariant = (isSafeToSpeculativelyExecute(I) && !I->mayReadFromMemory() && !isa<LandingPadInst>(I));

          if (mightBeLoopInvariant) {
            bool allOperandsHaveSingleLoopInvariantDef = true;

            for (User::op_iterator opIter = I->op_begin(), e = I->op_end(); opIter != e; ++opIter) {
              Value* opVal = *opIter;
              ReachingDefinitionInfo varDefsInfo = reachingDefs[opVal];

              //Check whether operand has single, loop-invariant definition.
              vector<Value*> varDefsAtStatement = varDefsInfo.defsByPoint[I];
              if (varDefsAtStatement.size() != 1 || loopInvariantStatements.count(varDefsAtStatement[0]) == 0) {
                allOperandsHaveSingleLoopInvariantDef = false;
                break;
              }
            }

            if (allOperandsHaveSingleLoopInvariantDef)
              loopInvariantStatements.insert(v);
          }
        }
      }
    }

    invariantSetSize = loopInvariantStatements.size();
    converged = (invariantSetSize == prevInvariantSetSize);
  }

  //DEBUGGING: Print out loop invariant statements
  errs() << "Loop invariant statements: {\n";
  for (std::set<Value*>::iterator liIter = loopInvariantStatements.begin(); liIter != loopInvariantStatements.end(); ++liIter) {
    errs() << valueToStr(*liIter) << "\n";
  }
  errs() << "}\n\n";

  return loopInvariantStatements;
}

set<Value*> LoopInvariantCodeMotion::computeCodeMotionCandidateStatements(Loop* L, DataFlowResult dominanceResults, set<Value*> invari
antStatements) {
  set<Value*> motionCandidates;

  //Candidate statements for LICM must meet the following:
  //1) Must be loop invariant
  //2) Must be in a block that dominates all exits of the loop
  //3) Must be in a block that dominates all blocks in the loop where the definition variable of the statement is used
  //4) Must assign to a variable that has no other assignments in the loop

  std::vector<BasicBlock*> loopBlocks = L->getBlocks();
  SmallPtrSet<BasicBlock*, 32> loopExits = getLoopExits(L);

  for (std::vector<BasicBlock*>::iterator blockIter = loopBlocks.begin(); blockIter != loopBlocks.end(); ++blockIter) {
    for (BasicBlock::iterator instIter = (*blockIter)->begin(), e = (*blockIter)->end(); instIter != e; ++instIter) {
      Instruction* I = instIter;

//      errs() << "Looking at whether to make LICM candidate: " << valueToStr(I) << "\n";

      //Check invariance
      if (invariantStatements.count(I) == 0)
        continue;

      //Check exit dominance
      bool isInExitDominatingBlock = true;
      for (SmallPtrSet<BasicBlock*, 32>::iterator loopExitIter = loopExits.begin(); loopExitIter != loopExits.end(); ++loopExitIter) {
        //If this block doens’t dominate this exit, it’s not an exit dominating block
        if (!dominates(*blockIter, *loopExitIter, dominanceResults)) {
          isInExitDominatingBlock = false;
          break;
        }
      }
      if (!isInExitDominatingBlock)
        continue;

      //Check whether statement dominates other uses of the assigned variable in the block
      bool dominatesAllUseBlocksInLoop = true;
      Value* assignedVar = getDefinitionVar(I);
      if (assignedVar) {
        for (Value::use_iterator useIter = assignedVar->use_begin(), e = assignedVar->use_end(); useIter != e; ++useIter) {
          if (Instruction* userInstruction = dyn_cast<Instruction>(*useIter)) {
            BasicBlock* userBlock = userInstruction->getParent();
            if (L->contains(userBlock) && !dominates(*blockIter, userBlock, dominanceResults)) {
              dominatesAllUseBlocksInLoop = false;
              break;
            }
          }
        }
      }
      if (!dominatesAllUseBlocksInLoop)
        continue;


      //Check whether assigned variable has any other assignments in loop... not a candidate if so
      bool hasNoOtherAssignmentsInLoop = true;
      if (assignedVar) {
        string assignedVarStr = valueToDefinitionVarStr(assignedVar);
        //Inefficient, but just loop over all instructions again, checking for other assignments to the same var
        for (std::vector<BasicBlock*>::iterator blockIter = loopBlocks.begin(); blockIter != loopBlocks.end(); ++blockIter) {
          for (BasicBlock::iterator otherInstIter = (*blockIter)->begin(), e = (*blockIter)->end(); otherInstIter != e; ++otherInstIte
r) {
            if (otherInstIter != instIter && valueToDefinitionVarStr(otherInstIter) == assignedVarStr) {
              hasNoOtherAssignmentsInLoop = false;
              break;
            }
          }
          if (hasNoOtherAssignmentsInLoop)
            break;
        }
      }
      if (!hasNoOtherAssignmentsInLoop)
```

```
        continue;


      //At this point, we know this statement is a good LICM candidate
      motionCandidates.insert(I);
    }
  }

  return motionCandidates;
}

bool LoopInvariantCodeMotion::applyMotionToCandidates(Loop* L, set<Value*> motionCandidates) {
  bool motionApplied = false;

  BasicBlock* preheader = L->getLoopPreheader();

  set<Instruction*> toMoveSet;

  //Algorithm: Do a DFS over the blocks of the loop and move each candidate to end of preheader if all
  //of its dependencies have also been moved to the preheader

  set<BasicBlock*> visited;

  stack<BasicBlock*> work;
  work.push(*succ_begin(preheader)); //start at loop header... the sole successor of the pre-header
  while (!work.empty()) {
    BasicBlock* block = work.top();
    work.pop();
    visited.insert(block);

    //For each instruction in the block, move to preheader if it’s a code motion candidate and conditions are met
    for (BasicBlock::iterator instIter = block->begin(), e = block->end(); instIter != e; ++instIter) {
      Instruction* I = instIter;

      if (motionCandidates.count(I) > 0) {
        motionApplied = true;
        toMoveSet.insert(I);
      }
    }

    //Add successors to search
    for (succ_iterator successorBlock = succ_begin(block), E = succ_end(block); successorBlock != E; ++successorBlock) {
      if (L->contains(*successorBlock)) {
        if (visited.count(*successorBlock) == 0)
          work.push(*successorBlock);
      }
    }
  }

  //Move all the to-move items now (a bit too tricky to do it while iterating over blocks)
  for (set<Instruction*>::iterator it = toMoveSet.begin(); it != toMoveSet.end(); ++it) {
    Instruction* instructionToMove = *it;

    //Insert as the next-to-last instruction of preheader (last needs to remain the block’s control flow branch)
    Instruction* preheaderEnd = &(preheader->back());

//    errs() << "Preheader end: " << valueToStr(preheaderEnd) << "\n";

    instructionToMove->removeFromParent();
    instructionToMove->insertBefore(preheaderEnd);
  }

  return motionApplied;
}

void LoopInvariantCodeMotion::addLoopIntoQueue(Loop* L) {
  this->LQ.push_back(L);
  for (Loop::reverse_iterator I = L->rbegin(), E = L->rend(); I != E; ++I)
    addLoopIntoQueue(*I);
}

}
```

```
////////////////////////////////////////////////////////////////////////////////
// 15-745 S14 Assignment 3
// Group: bhumbers, psuresh
////////////////////////////////////////////////////////////////////////////////

#ifndef __CLASSICAL_DATAFLOW_DATAFLOW_H__
#define __CLASSICAL_DATAFLOW_DATAFLOW_H__

#include <stdio.h>

#include "llvm/IR/Instructions.h"
#include "llvm/ADT/BitVector.h"
#include "llvm/ADT/DenseMap.h"
#include "llvm/ADT/SmallSet.h"
#include "llvm/ADT/ValueMap.h"
#include "llvm/Support/CFG.h"

#include <vector>
#include <map>

using namespace std;

namespace llvm {

/** Returns the variable that is defined by the given value (argument, instruction, etc.),
* or null if the given value is not a definition */
Value* getDefinitionVar(Value* v);

/** Util to create string representation of given BitVector */
std::string bitVectorToStr(const BitVector& bv);

/** Util to output string representation of an llvm Value */
std::string valueToStr(const Value* value);

/** Returns string representation of a set of domain elements with inclusion indicated by a bit vector
 Each element is output according to the given valFormatFunc function */
std::string setToStr(std::vector<Value*> domain, const BitVector& includedInSet, std::string (*valFormatFunc)(Value*));

/** Returns string version of definition if the Value is in fact a definition, or an empty string otherwise.
 * eg: The defining instruction "%a = add nsw i32 %b, 1" will return exactly that: "%a = add nsw i32 %b, 1"*/
std::string valueToDefinitionStr(Value* v);

/** Returns the name of a defined variable if the given Value is a definition, or an empty string otherwise.
 * eg: The defining instruction "%a = add nsw i32 %b, 1" will return "a"*/
std::string valueToDefinitionVarStr(Value* v);

/** An intermediate transfer function output entry from a block. In addition to the main value,
 * may include a list of predecessor block-specific transfer values which are appended (unioned)
 * onto the main value for the meet operator input of each predecessor (used to handle SSA phi nodes) */
struct TransferResult {
  BitVector baseValue;
  DenseMap<BasicBlock*, BitVector> predSpecificValues;
};

struct DataFlowResultForBlock {
  //Final output
  BitVector in;
  BitVector out;

  //Intermediate results
  TransferResult currTransferResult;

  DataFlowResultForBlock() {}
  DataFlowResultForBlock(BitVector in, BitVector out) {
    this->in = in;
    this->out = out;
    this->currTransferResult.baseValue = out; //tra
  }
};

struct DataFlowResult {
  /** Mapping from domain entries to linear indices into value results from dataflow */
  DenseMap<Value*, int> domainEntryToValueIdx;

  /** Mapping from basic blocks to the IN and OUT value sets for each after analysis converges */
  map<BasicBlock*, DataFlowResultForBlock> resultsByBlock;
};

/** Base interface for running dataflow analysis passes.
 * Must be subclassed with pass-specific logic in order to be used.
*/
class DataFlow {
  public:
    enum Direction {
      FORWARD,
      BACKWARD
    };
    /** Run this dataflow analysis on the given list of blocks using given parameters.*/
    DataFlowResult run(std::vector<llvm::BasicBlock*> blocks,
                          std::vector<Value*> domain,
                          Direction direction,
                          BitVector boundaryCond,
                          BitVector initInteriorCond);

    /** Prints a representation of F to raw_ostream O. */
    void ExampleFunctionPrinter(raw_ostream& O, const Function& F);

    void PrintInstructionOps(raw_ostream& O, const Instruction* I);

  protected:
    /** Meet operator behavior; specific to the subclassing data flow */
    virtual BitVector applyMeet(std::vector<BitVector> meetInputs) = 0;

    /** Transfer function behavior; specific to a subclassing data flow
     * domainEntryToValueIdx provides mapping from domain elements to the linear bitvector index for that element. */
    virtual TransferResult applyTransfer(const BitVector& value, DenseMap<Value*, int> domainEntryToValueIdx, BasicBlock* block) = 0;
};

}

#endif
```

```
////////////////////////////////////////////////////////////////////////////////
// 15-745 S14 Assignment 3
// Group: bhumbers, psuresh
////////////////////////////////////////////////////////////////////////////////

#include <set>
#include <sstream>

#include "dataflow.h"

#include "llvm/Support/raw_ostream.h"

#include "llvm/Support/CFG.h"

namespace llvm {

/* Var definition util */
Value* getDefinitionVar(Value* v) {
  // Definitions are assumed to be one of:
  // 1) Function arguments
  // 2) Store instructions (2nd argument is the variable being (re)defined)
  // 3) Instructions that start with "  %" (note the 2x spaces)
  //      Note that this is a pretty brittle and hacky way to catch what seems the most common definition type in LLVM.
  //      Unfortunately, we couldn’t figure a better way to catch all definitions otherwise, as cases like
  //      "%0" and "%1" don’t show up  when using "getName()" to identify definition instructions.
  //      There’s got to be a better way, though...

  if (isa<Argument>(v)) {
    return v;
  }
  else if (isa<StoreInst>(v)) {
    return ((StoreInst*)v)->getPointerOperand();
  }
  else if (isa<Instruction>(v)){
    std::string str = valueToStr(v);
    const int VAR_NAME_START_IDX = 2;
    if (str.length() > VAR_NAME_START_IDX && str.substr(0,VAR_NAME_START_IDX+1) == "  %")
      return v;
  }
  return 0;
}

/*********************************************************************************************
 * String output utilities */
std::string bitVectorToStr(const BitVector& bv) {
  std::string str(bv.size(), ’0’);
  for (int i = 0; i < bv.size(); i++)
    str[i] = bv[i] ? ’1’ : ’0’;
  return str;
}

std::string valueToStr(const Value* value) {
  std::string instStr; llvm::raw_string_ostream rso(instStr);
  value->print(rso);
  return instStr;
}

const int VAR_NAME_START_IDX = 2;

std::string valueToDefinitionStr(Value* v) {
  //Verify it’s a definition first
  Value* def = getDefinitionVar(v);
  if (def == 0)
    return "";

  std::string str = valueToStr(v);
  if (isa<Argument>(v)) {
    return str;
  }
  else {
      str = str.substr(VAR_NAME_START_IDX);
      return str;
  }

  return "";
}

std::string valueToDefinitionVarStr(Value* v) {
  //Similar to valueToDefinitionStr, but we return just the defined var rather than the whole definition

  Value* def = getDefinitionVar(v);
  if (def == 0)
    return "";

  if (isa<Argument>(def) || isa<StoreInst>(def)) {
    return "%" + def->getName().str();
  }
  else {
    std::string str = valueToStr(def);
    int varNameEndIdx = str.find(' ',VAR_NAME_START_IDX);
    str = str.substr(VAR_NAME_START_IDX,varNameEndIdx-VAR_NAME_START_IDX);
    return str;
  }
}

std::string setToStr(std::vector<Value*> domain, const BitVector& includedInSet, std::string (*valFormatFunc)(Value*)) {
  std::stringstream ss;
  ss << "{\n";
  int numInSet = 0;
  for (int i = 0; i < domain.size(); i++) {
    if (includedInSet[i]) {
      if (numInSet > 0) ss << " \n";
      numInSet++;
      ss << "    " << valFormatFunc(domain[i]);
    }
  }
  ss << "}";
  return ss.str();
}

/* End string output utilities *
******************************************************************************************/


DataFlowResult DataFlow::run(std::vector<llvm::BasicBlock*> blocks,
                              std::vector<Value*> domain,
                              Direction direction,
                              BitVector boundaryCond,
                              BitVector initInteriorCond) {
  map<BasicBlock*, DataFlowResultForBlock> resultsByBlock;
  bool analysisConverged = false;

  //Create mapping from domain entries to linear indices
  //(simplifies updating bitvector entries given a particular domain element)
  DenseMap<Value*, int> domainEntryToValueIdx;
  for (int i = 0; i < domain.size(); i++)
    domainEntryToValueIdx[domain[i]] = i;

  std::set<BasicBlock*> blocksSet;
  for (int i = 0; i < blocks.size(); i++) blocksSet.insert(blocks[i]);

  //Set initial val for boundary blocks, which depend on direction of analysis
  std::set<BasicBlock*> boundaryBlocks;
  switch (direction) {
    case FORWARD:
      //Post-"entry" block assumed to be the first one without a predecessor, or whose predecessors aren't in the given blocks list
      for(std::vector<BasicBlock*>::iterator blockIter = blocks.begin(), E = blocks.end(); blockIter != E; ++blockIter) {
        if (pred_begin(*blockIter) == pred_end(*blockIter)) {
          boundaryBlocks.insert(*blockIter);
        }
        else {
         bool predsNotInList = true;
         for (pred_iterator predBlock = pred_begin((*blockIter)), E = pred_end((*blockIter)); predBlock != E; ++predBlock) {
           if (blocksSet.count(*predBlock) > 0) {
            predsNotInList = false;
            break;
           }
         }
         if (predsNotInList)
           boundaryBlocks.insert(*blockIter);
        }
      }
      break;
    case BACKWARD:
      //Pre-"exit" blocks = those that have a return statement
      for(std::vector<BasicBlock*>::iterator blockIter = blocks.begin(), E = blocks.end(); blockIter != E; ++blockIter)
        if (isa<ReturnInst>((*blockIter)->getTerminator()))
          boundaryBlocks.insert((*blockIter));
      break;
  }
  for (std::set<BasicBlock*>::iterator boundaryBlock = boundaryBlocks.begin(); boundaryBlock != boundaryBlocks.end(); boundaryBlock++)
 {
    DataFlowResultForBlock boundaryResult = DataFlowResultForBlock();
    //Set either the "IN" of post-entry blocks or the "OUT" of pre-exit blocks (since entry/exit blocks don't actually exist...)
    BitVector* boundaryVal = (direction == FORWARD) ? &boundaryResult.in : &boundaryResult.out;
    *boundaryVal = boundaryCond;
    boundaryResult.currTransferResult.baseValue = boundaryCond;
    resultsByBlock[*boundaryBlock] = boundaryResult;

//    errs() << "Boundary block init for " << (*boundaryBlock)->getName() << ": IN = " << bitVectorToStr(resultsByBlock[*boundaryBlock
].in)
//        << "; OUT = " << bitVectorToStr(resultsByBlock[*boundaryBlock].out) << "\n";
  }
  //Set initial vals for interior blocks (either OUTs for fwd analysis or INs for bwd analysis)
  //NOTE: Since we don’t actually have a dedicated boundary block like ENTRY/EXIT, we include the "boundary"
  //blocks in the initial interior condition setup (otherwise, initial vals for "boundary" blocks is indeterminate)
  for (std::vector<BasicBlock*>::iterator blockIter = blocks.begin(); blockIter != blocks.end(); ++blockIter) {

    DataFlowResultForBlock interiorInitResult;
    if (boundaryBlocks.find((*blockIter)) != boundaryBlocks.end())
      interiorInitResult = resultsByBlock[*blockIter];
    BitVector* interiorInitVal = (direction == FORWARD) ? &interiorInitResult.out : &interiorInitResult.in;
    *interiorInitVal = initInteriorCond;
    interiorInitResult.currTransferResult.baseValue = initInteriorCond;
    resultsByBlock[*blockIter] = interiorInitResult;
//    errs() << "Interior block init for " << (*blockIter)->getName() << ": IN = " << bitVectorToStr(resultsByBlock[*blockIter].in)
//        << "; OUT = " << bitVectorToStr(resultsByBlock[*blockIter].out) << "\n";
  }

  //Generate analysis "predecessor" list for each block (depending on direction of analysis)
  //Note that we only include as predecessors those blocks which are included in the input list
  //Will be used to drive the meet inputs.
  DenseMap<BasicBlock*, std::vector<BasicBlock*> > analysisPredsByBlock;
  for (std::vector<BasicBlock*>::iterator blockIter = blocks.begin(); blockIter != blocks.end(); ++blockIter) {
      std::vector<BasicBlock*> analysisPreds;

//      errs() << "Building predecessor list for : " << (*blockIter)->getName().str() << "\n";

      switch (direction) {
        case FORWARD:
          for (pred_iterator predBlock = pred_begin((*blockIter)), E = pred_end((*blockIter)); predBlock != E; ++predBlock) {
            if (blocksSet.count(*predBlock) > 0)
              analysisPreds.push_back(*predBlock);
          }
          break;
        case BACKWARD:
          for (succ_iterator succBlock = succ_begin((*blockIter)), E = succ_end((*blockIter)); succBlock != E; ++succBlock) {
            if (blocksSet.count(*succBlock) > 0)
              analysisPreds.push_back(*succBlock);
          }
          break;
      }

      analysisPredsByBlock[(*blockIter)] = analysisPreds;
  }

  //Iterate over blocks in function until convergence of output sets for all blocks
  while (!analysisConverged) {
    analysisConverged = true; //assume converged until proven otherwise during this iteration

    //TODO: if analysis is backwards, may want instead to iterate from back-to-front of blocks list

    for (std::vector<BasicBlock*>::iterator blockIter = blocks.begin(); blockIter != blocks.end(); ++blockIter) {
      DataFlowResultForBlock& blockVals = resultsByBlock[*blockIter];

      //Store old output before applying this analysis pass to the block (depends on analysis dir)
      DataFlowResultForBlock oldBlockVals = blockVals;
      BitVector oldPassOut = (direction == FORWARD) ? blockVals.out : blockVals.in;

      //If any analysis predecessors have outputs ready, apply meet operator to generate updated input set for this block
      BitVector* passInPtr = (direction == FORWARD) ? &blockVals.in : &blockVals.out;
      std::vector<BasicBlock*> analysisPreds = analysisPredsByBlock[*blockIter];
      std::vector<BitVector> meetInputs;
      //Iterate over analysis predecessors in order to generate meet inputs for this block
      for (std::vector<BasicBlock*>::iterator analysisPred = analysisPreds.begin(); analysisPred < analysisPreds.end(); ++analysisPred
) {
        DataFlowResultForBlock& predVals = resultsByBlock[*analysisPred];

        BitVector meetInput = predVals.currTransferResult.baseValue;

        //If this pred matches a predecessor-specific value for the current block, union that value into value set
        DenseMap<BasicBlock*, BitVector>::iterator predSpecificValueEntry = predVals.currTransferResult.predSpecificValues.find(*block
Iter);
        if (predSpecificValueEntry != predVals.currTransferResult.predSpecificValues.end()) {
//            errs() << "Pred-specific meet input from " << (*analysisPred)->getName() << ": " <<bitVectorToStr(predSpecificValueEntry
->second) << "\n";
            meetInput |= predSpecificValueEntry->second;
        }

        meetInputs.push_back(meetInput);
      }
//      errs() << "Meeting inputs for block: " << (*blockIter)->getName() << "\n";
      if (!meetInputs.empty())
        *passInPtr = applyMeet(meetInputs);

      //Apply transfer function to input set in order to get output set for this iteration
      blockVals.currTransferResult = applyTransfer(*passInPtr, domainEntryToValueIdx, *blockIter);
      BitVector* passOutPtr = (direction == FORWARD) ? &blockVals.out : &blockVals.in;
      *passOutPtr = blockVals.currTransferResult.baseValue;
      //Update convergence: if the output set for this block has changed, then we’ve not converged for this iteration
      if (analysisConverged) {
        if (*passOutPtr != oldPassOut)
          analysisConverged = false;
        else if (blockVals.currTransferResult.predSpecificValues.size() != oldBlockVals.currTransferResult.predSpecificValues.size())
          analysisConverged = false;
        //(should really check whether contents of pred-specific values changed as well, but
        // that doesn’t happen when the pred-specific values are just a result of phi-nodes)
      }
    }
  }

  DataFlowResult result;
  result.domainEntryToValueIdx = domainEntryToValueIdx;
  result.resultsByBlock = resultsByBlock;
  return result;
}

void DataFlow::PrintInstructionOps(raw_ostream& O, const Instruction* I) {
  O << "\nOps: {";
  if (I != NULL) {
    for (Instruction::const_op_iterator OI = I->op_begin(), OE = I->op_end();
        OI != OE; ++OI) {
      const Value* v = OI->get();
      v->print(O);
      O << ";";
    }
  }
  O << "}\n";
}

void DataFlow::ExampleFunctionPrinter(raw_ostream& O, const Function& F) {
  for (Function::const_iterator FI = F.begin(), FE = F.end(); FI != FE; ++FI) {
    const BasicBlock* block = FI;
    O << block->getName() << ":\n";
    const Value* blockValue = block;
    PrintInstructionOps(O, NULL);
    for (BasicBlock::const_iterator BI = block->begin(), BE = block->end();
        BI != BE; ++BI) {
      BI->print(O);
      PrintInstructionOps(O, &(*BI));
    }
  }
}

}
```

```
////////////////////////////////////////////////////////////////////////////////
// 15-745 S14 Assignment 3
// Group: bhumbers, psuresh
////////////////////////////////////////////////////////////////////////////////

#ifndef __CLASSICAL_DATAFLOW_REACHING_DEFS_H__
#define __CLASSICAL_DATAFLOW_REACHING_DEFS_H__

#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/InstIterator.h"

#include "dataflow.h"

#include <map>

using namespace llvm;
using namespace std;

namespace llvm {

struct ReachingDefinitionInfo {
  //The variable for which the definitions apply
  Value* variable;

  //Mapping from program points (just above instruction key) to definitions that reach that point (values) for this variable
  map<Instruction*, vector<Value*> > defsByPoint;

  ReachingDefinitionInfo() {
    variable = 0;
  }
};

/** A modified version of our reaching definitions function from A2 which
 * now returns a mapping from variables to reaching definitions.
 * Includes fixes for more correct handling of definitions both with and without SSA form. */
class ReachingDefinitions {

 public:

  /** For the given function, returns lookup to reaching definitions for each variable*/
  map<Value*, ReachingDefinitionInfo> computeReachingDefinitions(Function& F);
};

}

#endif
```

```
////////////////////////////////////////////////////////////////////////////////
// 15-745 S14 Assignment 3
// Group: bhumbers, psuresh
////////////////////////////////////////////////////////////////////////////////

#include "reaching-defs.h"

namespace llvm {

//////////////////////////////////////////////////////////////////////////////////////////////
//Dataflow analysis
class ReachingDefinitionsDataFlow : public DataFlow {

  protected:
    BitVector applyMeet(std::vector<BitVector> meetInputs) {
      BitVector meetResult;

      //Meet op = union of inputs
      if (!meetInputs.empty()) {
        for (int i = 0; i < meetInputs.size(); i++) {
//          if (i > 0) errs() << ", ";
//          errs() << bitVectorToStr(meetInputs[i]);
          if (i == 0)
            meetResult = meetInputs[i];
          else
            meetResult |= meetInputs[i];
        }
      }
//      errs() << "\n";

      return meetResult;
    }

    TransferResult applyTransfer(const BitVector& value, DenseMap<Value*, int> domainEntryToValueIdx, BasicBlock* block) {
      TransferResult transfer;

//      errs() << "Applying transfer for block: " << block->getName() << "\n";
//      errs() << "Pre: " << bitVectorToStr(value) << "\n";

      //First, calculate the set of downwards exposed definition generations and the set of killed definitions in this block
      int domainSize = domainEntryToValueIdx.size();
      BitVector genSet(domainSize);
      BitVector killSet(domainSize);
      for (BasicBlock::iterator instruction = block->begin(); instruction != block->end(); ++instruction) {
        DenseMap<Value*, int>::const_iterator currDefIter = domainEntryToValueIdx.find(&*instruction);
        if (currDefIter != domainEntryToValueIdx.end()) {
          //Kill prior definitions for the same variable (including those in this block’s gen set)
          for (DenseMap<Value*, int>::const_iterator prevDefIter = domainEntryToValueIdx.begin();
               prevDefIter != domainEntryToValueIdx.end();
               ++prevDefIter) {
            std::string prevDefStr = valueToDefinitionVarStr(prevDefIter->first);
            std::string currDefStr = valueToDefinitionVarStr(currDefIter->first);
            if (prevDefStr == currDefStr) {
              killSet.set(prevDefIter->second);
              genSet.reset(prevDefIter->second);
            }
          }

          //Add this new definition to gen set (note that we might later remove it if another def in this block kills it)
          genSet.set((*currDefIter).second);
        }
      }

      //Then, apply transfer function: Y = GenSet \union (X - KillSet)
      transfer.baseValue = killSet;
      transfer.baseValue.flip();
      transfer.baseValue &= value;
      transfer.baseValue |= genSet;

//      errs() << "Post: " << bitVectorToStr(transfer.baseValue) << "\n";

      return transfer;
    }
};
//////////////////////////////////////////////////////////////////////////////////////////////

map<Value*, ReachingDefinitionInfo> ReachingDefinitions::computeReachingDefinitions(Function& F) {
  map<Value*, ReachingDefinitionInfo> reachingDefs;

  //NOTE: Unfortunately, we don’t have enought time to handle SSA aliasing correctly

  //Set domain = definitions in the function
  std::vector<Value*> domain;
  for (Function::arg_iterator arg = F.arg_begin(); arg != F.arg_end(); ++arg)
    domain.push_back(arg);
  for (inst_iterator instruction = inst_begin(F), e = inst_end(F); instruction != e; ++instruction) {
    //If instruction is nonempty when converted to a definition string, then it’s a definition and belongs in our domain
    if (!valueToDefinitionStr(&*instruction).empty())
      domain.push_back(&*instruction);
  }

  //Initialize keys for reaching definition lookup (consists of the defined variables in our domain(
  for (int i = 0; i < domain.size(); i++) {
    Value* definedVar = getDefinitionVar(domain[i]);
    reachingDefs[definedVar] = ReachingDefinitionInfo();
  }

  int numVars = domain.size();

  //Set the initial boundary dataflow value to be the set of input argument definitions for this function
  BitVector boundaryCond(numVars, false);
  for (int i = 0; i < domain.size(); i++)
    if (isa<Argument>(domain[i]))
      boundaryCond.set(i);

  //Set interior initial dataflow values to be empty sets
  BitVector initInteriorCond(numVars, false);

  //Get dataflow values at IN and OUT points of each block
  ReachingDefinitionsDataFlow flow;
  vector<BasicBlock*> blocks;
  for (Function::iterator blockIter = F.begin(); blockIter != F.end(); ++blockIter)
    blocks.push_back(blockIter);
  DataFlowResult dataFlowResult = flow.run(blocks, domain, DataFlow::FORWARD, boundaryCond, initInteriorCond);

  //Then, extend those values into the interior points of each block, outputting the result along the way
//  errs() << "\n****************** REACHING DEFINITIONS OUTPUT FOR FUNCTION: " << F.getName() << " *****************\n";
//  errs() << "Domain of values: " << setToStr(domain, BitVector(domain.size(), true), valueToDefinitionStr) << "\n";
//  errs() << "Variables: "   << setToStr(domain, BitVector(domain.size(), true), valueToDefinitionVarStr) << "\n";

  //Print function header (in hacky way... look for "definition" keyword in full printed function, then print rest of that line only)
  std::string funcStr = valueToStr(&F);
  int funcHeaderStartIdx = funcStr.find("define");
  int funcHeaderEndIdx = funcStr.find('{', funcHeaderStartIdx + 1);
//  errs() << funcStr.substr(funcHeaderStartIdx, funcHeaderEndIdx-funcHeaderStartIdx) << "\n";

  //Now, use dataflow results to output reaching definitions at program points within each block
  for (Function::iterator basicBlock = F.begin(); basicBlock != F.end(); ++basicBlock) {
    DataFlowResultForBlock blockReachingDefVals = dataFlowResult.resultsByBlock[basicBlock];

    //Print just the header line of the block (in a hacky way... blocks start w/ newline, so look for first occurrence of newline beyo
nd first char
    std::string basicBlockStr = valueToStr(basicBlock);
//    errs() << basicBlockStr.substr(0, basicBlockStr.find(':', 1) + 1) << "\n";

    //Initialize reaching definitions at the start of the block
    BitVector reachingDefVals = blockReachingDefVals.in;

    std::vector<std::string> blockOutputLines;

    //Output reaching definitions at the IN point of this block (not strictly needed, but useful to see)
    blockOutputLines.push_back("\nReaching Defs: " + setToStr(domain, reachingDefVals, valueToDefinitionStr) + "\n");

    //Iterate forward through instructions of the block, updating and outputting reaching defs
    for (BasicBlock::iterator instruction = basicBlock->begin(); instruction != basicBlock->end(); ++instruction) {
      //In the output data, mark all the reaching defs just before this instruction
      for (int i = 0; i < domain.size(); i++) {
        if (reachingDefVals[i]) {
          Value* definition = domain[i];
          Value* definedVar = getDefinitionVar(definition);
          ReachingDefinitionInfo& defsInfoForVar = reachingDefs[definedVar];
          if (defsInfoForVar.defsByPoint.find(instruction) == defsInfoForVar.defsByPoint.end())
            defsInfoForVar.defsByPoint[instruction] = vector<Value*>();
          defsInfoForVar.defsByPoint[instruction].push_back(definition);
        }
      }

      //REACHING DEF UPDATE FOR INSTRUCTION
      {
        DenseMap<Value*, int>::const_iterator defIter;

        std::string currDefStr = valueToDefinitionVarStr(instruction);

        //Kill (unset) all existing defs for this variable
        //(is there a better way to do this than string comparison of the defined var names?)
        for (defIter = dataFlowResult.domainEntryToValueIdx.begin(); defIter != dataFlowResult.domainEntryToValueIdx.end(); ++defIter)
 {
          std::string prevDefStr = valueToDefinitionVarStr(defIter->first);
          if (prevDefStr == currDefStr)
            reachingDefVals.reset(defIter->second);
        }

        //Add this definition to the reaching set
        defIter = dataFlowResult.domainEntryToValueIdx.find(&*instruction);
        if (defIter != dataFlowResult.domainEntryToValueIdx.end())
          reachingDefVals.set((*defIter).second);
      }

      //Add debugging output lines for the reaching defs
      {
        //Output the instruction contents
        blockOutputLines.push_back(valueToStr(&*instruction));

        //Output the set of reaching definitions at program point just past instruction
        //(but only if not a phi node... those aren’t "real" instructions)
        if (!isa<PHINode>(instruction)) {
          blockOutputLines.push_back("\nReaching Defs: " + setToStr(domain, reachingDefVals, valueToDefinitionStr) + "\n");
        }
      }
    }

    //CONSOLE OUTPUT (for debugging)
//    for (std::vector<std::string>::iterator i = blockOutputLines.begin(); i < blockOutputLines.end(); ++i)
//      errs() << *i << "\n";
  }
//  errs() << "****************** END REACHING DEFINITION OUTPUT FOR FUNCTION: " << F.getName() << " ******************\n\n";

  return reachingDefs;
}

}
```