```
// 15-745 S14 Assignment 2: reaching-definitions.cpp
// Group: bhumbers, psuresh
////////////////////////////////////////////////////////////////////////////////

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

#include "dataflow.h"

using namespace llvm;

namespace {

////////////////////////////////////////////////////////////////////////////////////////////
//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)
            meetResult = meetInputs[i];
          else
            meetResult |= meetInputs[i];
        }
      }

      return meetResult;
    }

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

      //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 variable (including those in this block's gen set)
          for (DenseMap<Value*, int>::const_iterator prevDefIter = domainEntryToValueIdx.begin();
               prevDefIter != domainEntryToValueIdx.end();
               ++prevDefIter) {
            if (prevDefIter->first->getName() == currDefIter->first->getName()) {
              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;

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

class ReachingDefinitions : public FunctionPass {
 public:
  static char ID;

  ReachingDefinitions() : FunctionPass(ID) { }

  virtual bool runOnFunction(Function& F) {
    //Set domain = definitions in the function
    //(since we're using SSA form, this is just the same as the set of variables in liveness analysis)
    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 has a nonempty definition variable, then it defines a variable for our domain
      if (!valueToDefinitionVarStr(&*instruction).empty())
```

```
        domain.push_back(&*instruction);
    }

    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;
    DataFlowResult dataFlowResult = flow.run(F, 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";

    //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 be
yond 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("Reaching Defs: " + setToStr(domain, reachingDefVals, valueToDefinitionStr));

      //Iterate forward through instructions of the block, updating and outputting reaching defs
      for (BasicBlock::iterator instruction = basicBlock->begin(); instruction != basicBlock->end(); ++instruction) {
        //Output the instruction contents
        blockOutputLines.push_back(valueToStr(&*instruction));

        DenseMap<Value*, int>::const_iterator defIter;

        //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)
 {
          if (defIter->first->getName() == instruction->getName())
            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);

        //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("Reaching Defs: " + setToStr(domain, reachingDefVals, valueToDefinitionStr));
      }

      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";

    // Did not modify the incoming Function.
    return false;
  }

  virtual void getAnalysisUsage(AnalysisUsage& AU) const {
    AU.setPreservesCFG();
  }

 private:
```

```
};

char ReachingDefinitions::ID = 0;
RegisterPass<ReachingDefinitions> X("cd-reaching-definitions",
    "15745 ReachingDefinitions");

}
```