```
// 15-745 S14 Assignment 2: liveness.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 "llvm/ADT/SmallPtrSet.h"

#include "dataflow.h"

using namespace llvm;

namespace {

////////////////////////////////////////////////////////////////////////////////////////
//Dataflow analysis
class LivenessDataFlow : 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 set of locally exposed uses and set of defined variables in this block
      int domainSize = domainEntryToValueIdx.size();
      BitVector defSet(domainSize);
      BitVector useSet(domainSize);
      for (BasicBlock::iterator instruction = block->begin(); instruction != block->end(); ++instruction) {
        //Locally exposed uses
        //Phi node handling: Add operands to predecessor-specific value set
        if (PHINode* phiNode = dyn_cast<PHINode>(&*instruction)) {
          for (int incomingIdx = 0; incomingIdx < phiNode->getNumIncomingValues(); incomingIdx++) {
            Value* val = phiNode->getIncomingValue(incomingIdx);
            if (isa<Instruction>(val) || isa<Argument>(val)) {
              int valIdx = domainEntryToValueIdx[val];

              BasicBlock* incomingBlock = phiNode->getIncomingBlock(incomingIdx);
              if (transfer.predSpecificValues.find(incomingBlock) == transfer.predSpecificValues.end())
                transfer.predSpecificValues[incomingBlock] = BitVector(domainSize);
              transfer.predSpecificValues[incomingBlock].set(valIdx);
            }
          }
        }
        //Non-phi node handling: Add operands to general use set
        else {
          User::op_iterator operand, opEnd;
          for (operand = instruction->op_begin(), opEnd = instruction->op_end(); operand != opEnd; ++operand) {
            Value* val = *operand;
            if (isa<Instruction>(val) || isa<Argument>(val)) {
              int valIdx = domainEntryToValueIdx[val];

              //Only locally exposed use if not defined earlier in this block
              if (!defSet[valIdx])
                useSet.set(valIdx);
            }
          }
        }

        //Definitions
        DenseMap<Value*, int>::const_iterator iter = domainEntryToValueIdx.find(instruction);
        if (iter != domainEntryToValueIdx.end())
          defSet.set((*iter).second);
      }

      //Then, apply liveness transfer function: Y = UseSet \union (X - DefSet)
      transfer.baseValue = defSet;
      transfer.baseValue.flip();
      transfer.baseValue &= value;
      transfer.baseValue |= useSet;
```

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

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

  Liveness() : FunctionPass(ID) { }

  virtual bool runOnFunction(Function& F) {
    //Set domain = variables 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 has a nonempty LHS variable name, then it defines a variable for our domain
      if (!valueToDefinitionVarStr(&*instruction).empty())
        domain.push_back(&*instruction);
    }

    int numVars = domain.size();

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

    //Get dataflow values at IN and OUT points of each block
    LivenessDataFlow flow;
    DataFlowResult dataFlowResult = flow.run(F, domain, DataFlow::BACKWARD, boundaryCond, initInteriorCond);

    //Then, extend those values into the interior points of each block, outputting the result along the way
    errs() << "\n********** LIVENESS OUTPUT FOR FUNCTION: " << F.getName() << " **********\n";
    errs() << "Domain of values: " << 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 liveness at program points within each block
    for (Function::iterator basicBlock = F.begin(); basicBlock != F.end(); ++basicBlock) {
      DataFlowResultForBlock blockLivenessVals = 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 liveness at end of block
      BitVector livenessVals = blockLivenessVals.out;

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

      //Output live variables at the OUT point of this block (not strictly needed, but useful to see)
      blockOutputLines.push_back("Liveness: " + setToStr(domain, livenessVals, valueToDefinitionVarStr));

      //Iterate backward through instructions of the block, updating and outputting liveness of vars as we go
      for (BasicBlock::reverse_iterator instruction = basicBlock->rbegin(); instruction != basicBlock->rend(); ++instruction) {
        //Output the instruction contents
        blockOutputLines.push_back(valueToStr(&*instruction));

        //Special treatment for phi functions: Kill LHS, but don't output liveness here (not a "real" instruction)
        PHINode* phiInst = dyn_cast<PHINode>(&*instruction);
        if (phiInst) {
          DenseMap<Value*, int>::const_iterator defIter = dataFlowResult.domainEntryToValueIdx.find(phiInst);
          if (defIter != dataFlowResult.domainEntryToValueIdx.end())
            livenessVals.reset((*defIter).second);
        }
        else {
          //Add vars to live set when used as operands
          for (Instruction::const_op_iterator operand = instruction->op_begin(), opEnd = instruction->op_end(); operand != opEnd; ++op
erand) {
            Value* val = *operand;
            if (isa<Instruction>(val) || isa<Argument>(val)) {
              int valIdx = dataFlowResult.domainEntryToValueIdx[val];
              livenessVals.set(valIdx);
            }
          }

          //Remove a var from live set at its definition (this is its unique definition in SSA form)
          DenseMap<Value*, int>::const_iterator defIter = dataFlowResult.domainEntryToValueIdx.find(&*instruction);
          if (defIter != dataFlowResult.domainEntryToValueIdx.end())
            livenessVals.reset((*defIter).second);
```

```
          //Output the set of live variables at program point just before instruction
          blockOutputLines.push_back("Liveness: " + setToStr(domain, livenessVals, valueToDefinitionVarStr));
        }
      }
      //Print out in reverse order (since we iterated backward over instructions)
      for (std::vector<std::string>::reverse_iterator i = blockOutputLines.rbegin(); i < blockOutputLines.rend(); ++i)
        errs() << *i << "\n";
    }
    errs() << "****************** END LIVENESS OUTPUT FOR FUNCTION: " << F.getName() << " ******************\n\n";

    //flow.ExampleFunctionPrinter(errs(), F);

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

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

 private:
};



char Liveness::ID = 0;
RegisterPass<Liveness> X("cd-liveness", "15745 Liveness");

}
```