```
// 15-745 S14 Assignment 2: dataflow.h
// 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>

namespace llvm {

/** 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 */
  DenseMap<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 function using given parameters.*/
    DataFlowResult run(Function& F,
                       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 2: dataflow.cpp
// Group: bhumbers, psuresh
////////////////////////////////////////////////////////////////////////////////

#include <set>
#include <sstream>

#include "dataflow.h"

#include "llvm/Support/raw_ostream.h"

#include "llvm/Support/CFG.h"

namespace llvm {

/*****************************************************************************************
 * 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;
}

std::string valueToDefinitionStr(Value* v) {
  std::string str = valueToStr(v);
  //Really, really brittle code: Definitions are assumed to either be arguments or to be instructions that start with "  %" (note the
2x spaces)
  //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 str;
  }
  else if (isa<Instruction>(v)){
    int varNameStartIdx = 2;
    if (str.length() > varNameStartIdx && str.substr(0,varNameStartIdx+1) == "  %") {
      str = str.substr(varNameStartIdx);
      return str;
    }
    else
      return "";
  }
  return "";
}

std::string valueToDefinitionVarStr(Value* v) {
  //Similar to valueToDefinitionStr, but we extract just the var name
  if (isa<Argument>(v)) {
    return "%" + v->getName().str();
  }
  else if (isa<Instruction>(v)){
    std::string str = valueToStr(v);
    int varNameStartIdx = 2;
    if (str.length() > varNameStartIdx && str.substr(0,varNameStartIdx+1) == "  %") {
      int varNameEndIdx = str.find(' ',varNameStartIdx);
      str = str.substr(varNameStartIdx,varNameEndIdx-varNameStartIdx);
      return str;
    }
    else
      return "";
  }
  return "";
}

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

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

```
DataFlowResult DataFlow::run(Function& F,
                             std::vector<Value*> domain,
                             Direction direction,
                             BitVector boundaryCond,
                             BitVector initInteriorCond) {
  DenseMap<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;

  //Set initial val for boundary blocks, which depend on direction of analysis
  std::set<BasicBlock*> boundaryBlocks;
  switch (direction) {
    case FORWARD:
      boundaryBlocks.insert(&F.front()); //post-"entry" block = first in list
      break;
    case BACKWARD:
      //Pre-"exit" blocks = those that have a return statement
      for(Function::iterator I = F.begin(), E = F.end(); I != E; ++I)
        if (isa<ReturnInst>(I->getTerminator()))
          boundaryBlocks.insert(I);
      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;
  }

  //Set initial vals for interior blocks (either OUTs for fwd analysis or INs for bwd analysis)
  for (Function::iterator basicBlock = F.begin(); basicBlock != F.end(); ++basicBlock) {
    if (boundaryBlocks.find((BasicBlock*)basicBlock) == boundaryBlocks.end()) {
      DataFlowResultForBlock interiorInitResult = DataFlowResultForBlock();
      BitVector* interiorInitVal = (direction == FORWARD) ? &interiorInitResult.out : &interiorInitResult.in;
      *interiorInitVal = initInteriorCond;
      interiorInitResult.currTransferResult.baseValue = initInteriorCond;
      resultsByBlock[basicBlock] = interiorInitResult;
    }
  }

  //Generate analysis "predecessor" list for each block (depending on direction of analysis)
  //Will be used to drive the meet inputs.
  DenseMap<BasicBlock*, std::vector<BasicBlock*> > analysisPredsByBlock;
  for (Function::iterator basicBlock = F.begin(); basicBlock != F.end(); ++basicBlock) {
      std::vector<BasicBlock*> analysisPreds;
      switch (direction) {
        case FORWARD:
          for (pred_iterator predBlock = pred_begin(basicBlock), E = pred_end(basicBlock); predBlock != E; ++predBlock)
            analysisPreds.push_back(*predBlock);
          break;
        case BACKWARD:
          for (succ_iterator succBlock = succ_begin(basicBlock), E = succ_end(basicBlock); succBlock != E; ++succBlock)
            analysisPreds.push_back(*succBlock);
          break;
      }

      analysisPredsByBlock[basicBlock] = 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 (Function::iterator basicBlock = F.begin(); basicBlock != F.end(); ++basicBlock) {
      DataFlowResultForBlock& blockVals = resultsByBlock[basicBlock];

      //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[basicBlock];
      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(basicB
lock);
        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);
      }
      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, basicBlock);
      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));
    }
  }
}

}
```