15-745: Optimizing Compilers
Assignment 2 Writeup
Ben Humberston, bhumbers
Prashanth Suresh, psuresh

Implementation Summary

Generic Data Flow Framework:
See DataFlow.h and .cpp. Rather than use function pointers passed into a dataflow function, we define the generic dataflow as an abstract interface. Any pass which wishes to use the framework must subclass *DataFlow* and implement the *applyMeet()* and *applyTransfer()* functions with the logic for the pass. Calling *run()* then executes the dataflow analysis and returns the values for each block's IN and OUT.

Values (sets) are represented by BitVector, and we decided that using a vector of *Value* pointers as the domain type was sufficiently generic to cover most passes. The results of calling *run()* includes the mapping from input domain elements to positions in the value bit vectors (this is equivalent to the order of entries in the given domain, but it seemed prudent to make this mapping explicit).

Liveness:
See liveness.c. The generic dataflow framework is used to compute the liveness of each variable at the IN and OUT of all blocks (see "*LivenessDataFlow*" implementation in liveness.cpp for meet operator and transfer function). Then, these liveness values are expanded to the interior of each block by iterating backward over instructions and updating liveness instruction-by-instruction.

In order to make liveness results sensible, we have special handling for ϕ nodes. If not otherwise alive, the liveness of any operands used in ϕ instructions is only propagated backward along the edge to the predecessor block associated with that operand.

Reaching Definitions:
The implementation of this pass is similar in many respects to the liveness pass, except that analysis moves forward rather than backward and the transfer function is modified to use the correct gen and kill sets for reaching definitions. There is also no special ϕ node handling necessary in this pass. It should be noted, then, that all definitions reach the exit block as a result of the SSA form.

Code Listing

- DataFlow.h & .cpp
- liveness.cpp
- reaching_definitions.cpp
- /tests/*.c *(tests)*

Test Listing

Run "./ClassicalDataflow/RUN_ ME.sh" in a bash shell in order to 1) build the passes & test cases and 2) run all test cases. Test output will be shown in the console.

All tests are executed using both the liveness and reaching definition passes.

- sum.c
    - The example specified in the assignment writeup. As shown in the sample output below, our tests did not produce exactly the same IR as given in the writeup, but the pass results are largely comparable.
- basic.c
    - Tests absolute basic cases as well as nested conditionals and function calls (though this last was hard to verify due to tail recursion optimizations from LLVM).
- loc_exp_usage.c
    - Tests whether locally exposed uses are correctly processed

Test Output

(only showing sum.c test output as an example)

Liveness

```
****************** LIVENESS OUTPUT FOR FUNCTION: sum ****************
Domain of values: {%a | %b | %cmp4 | %res.06 | %i.05 | %mul | %inc | %exitcond | %res.0.lcssa}
define i32 @sum(i32 %a, i32 %b) #0

entry:
Liveness: {%a | %b}
  %cmp4 = icmp slt i32 %a, %b
Liveness: {%a | %b | %cmp4}
  br i1 %cmp4, label %for.body, label %for.end
Liveness: {%a | %b}

for.body:
  %res.06 = phi i32 [ %mul, %for.body ], [ 1, %entry ]
  %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ]
Liveness: {%b | %res.06 | %i.05}
  %mul = mul nsw i32 %res.06, %i.05
Liveness: {%b | %i.05 | %mul}
  %inc = add nsw i32 %i.05, 1
Liveness: {%b | %mul | %inc}
  %exitcond = icmp eq i32 %inc, %b
Liveness: {%b | %mul | %inc | %exitcond}
  br i1 %exitcond, label %for.end, label %for.body
Liveness: {%b | %mul | %inc}

for.end:
  %res.0.lcssa = phi i32 [ 1, %entry ], [ %mul, %for.body ]
Liveness: {%res.0.lcssa}
  ret i32 %res.0.lcssa
Liveness: {}
****************** END LIVENESS OUTPUT FOR FUNCTION: sum ******************
```

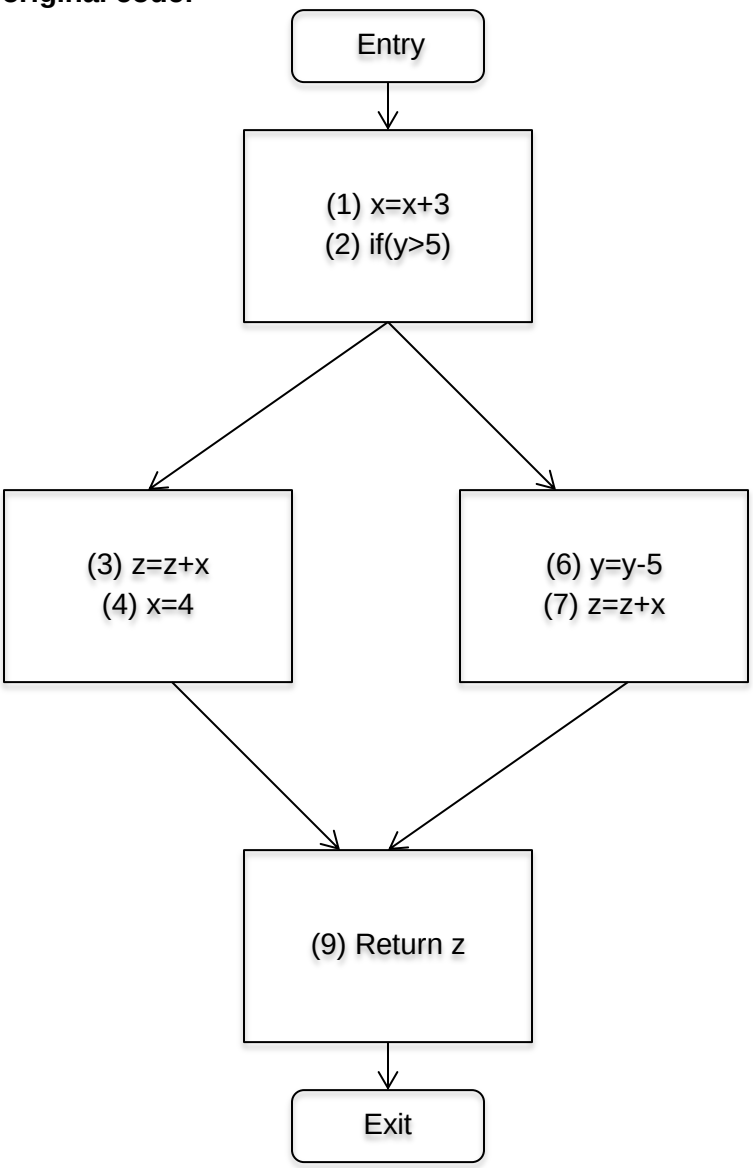
Reaching Definitions:

```
****************** REACHING DEFINITIONS OUTPUT FOR FUNCTION: sum *****************
Domain of values: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b | %res.06 = phi i32 [ %mul, %for.body ], [ 1,
%entry ] | %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ] | %mul = mul nsw i32 %res.06, %i.05 | %inc = add
nsw i32 %i.05, 1 | %exitcond = icmp eq i32 %inc, %b | %res.0.lcssa = phi i32 [ 1, %entry ], [ %mul, %for.body ]}
define i32 @sum(i32 %a, i32 %b) #0

entry:
Reaching Defs: {i32 %a | i32 %b}
  %cmp4 = icmp slt i32 %a, %b
Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b}
  br i1 %cmp4, label %for.body, label %for.end
Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b}

for.body:
Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b | %res.06 = phi i32 [ %mul, %for.body ], [ 1,
%entry ] | %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ] | %mul = mul nsw i32 %res.06, %i.05 | %inc = add
nsw i32 %i.05, 1 | %exitcond = icmp eq i32 %inc, %b}

  %res.06 = phi i32 [ %mul, %for.body ], [ 1, %entry ]
  %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ]
  %mul = mul nsw i32 %res.06, %i.05

Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b | %res.06 = phi i32 [ %mul, %for.body ], [ 1,
%entry ] | %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ] | %mul = mul nsw i32 %res.06, %i.05 | %inc = add
nsw i32 %i.05, 1 | %exitcond = icmp eq i32 %inc, %b}

  %inc = add nsw i32 %i.05, 1

Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b | %res.06 = phi i32 [ %mul, %for.body ], [ 1,
%entry ] | %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ] | %mul = mul nsw i32 %res.06, %i.05 | %inc = add
nsw i32 %i.05, 1 | %exitcond = icmp eq i32 %inc, %b}

  %exitcond = icmp eq i32 %inc, %b

Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b | %res.06 = phi i32 [ %mul, %for.body ], [ 1,
%entry ] | %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ] | %mul = mul nsw i32 %res.06, %i.05 | %inc = add
nsw i32 %i.05, 1 | %exitcond = icmp eq i32 %inc, %b}

  br i1 %exitcond, label %for.end, label %for.body

Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b | %res.06 = phi i32 [ %mul, %for.body ], [ 1,
%entry ] | %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ] | %mul = mul nsw i32 %res.06, %i.05 | %inc = add
nsw i32 %i.05, 1 | %exitcond = icmp eq i32 %inc, %b}

for.end:
Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b | %res.06 = phi i32 [ %mul, %for.body ], [ 1,
%entry ] | %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ] | %mul = mul nsw i32 %res.06, %i.05 | %inc = add
nsw i32 %i.05, 1 | %exitcond = icmp eq i32 %inc, %b}

  %res.0.lcssa = phi i32 [ 1, %entry ], [ %mul, %for.body ]
  ret i32 %res.0.lcssa

Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b | %res.06 = phi i32 [ %mul, %for.body ], [ 1,
%entry ] | %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ] | %mul = mul nsw i32 %res.06, %i.05 | %inc = add
nsw i32 %i.05, 1 | %exitcond = icmp eq i32 %inc, %b | %res.0.lcssa = phi i32 [ 1, %entry ], [ %mul, %for.body ]}
****************** END REACHING DEFINITION OUTPUT FOR FUNCTION: sum ******************
```

Q3

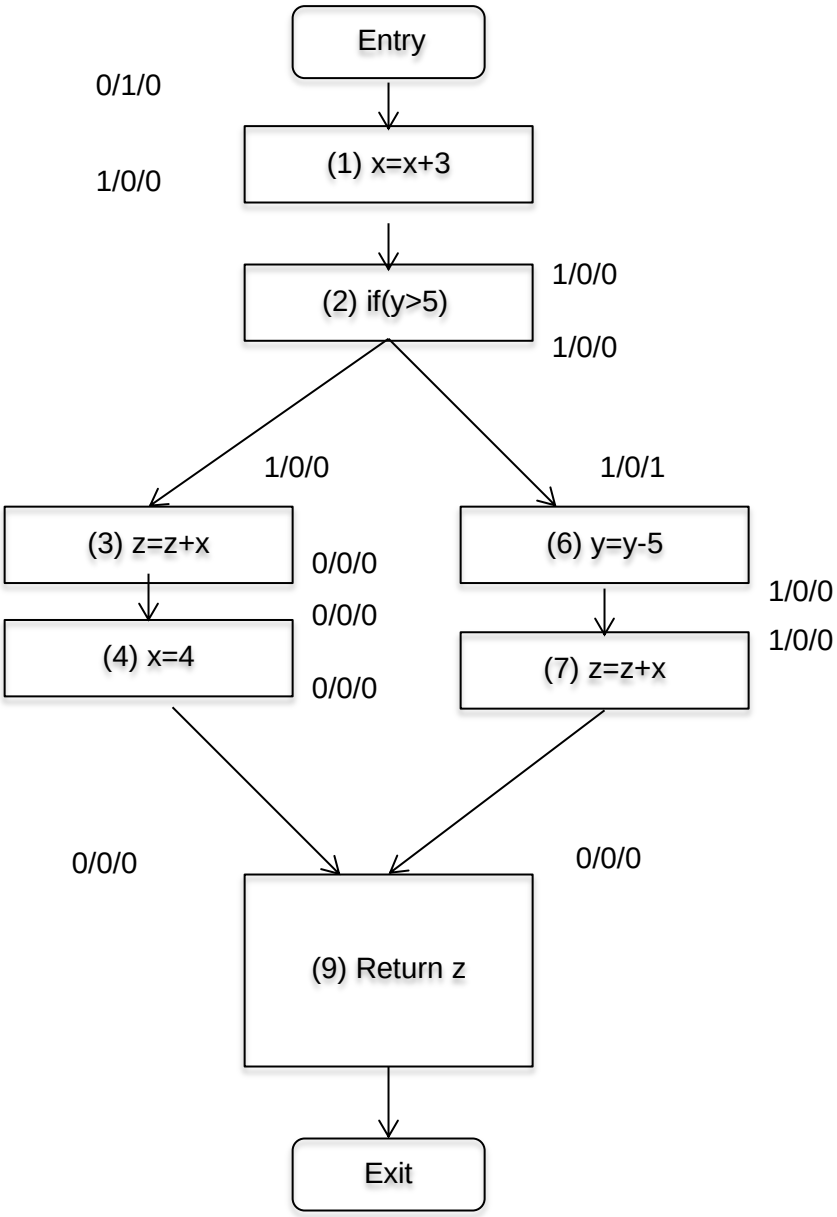
3.1 Lazy Code Motion

CFG for original code:

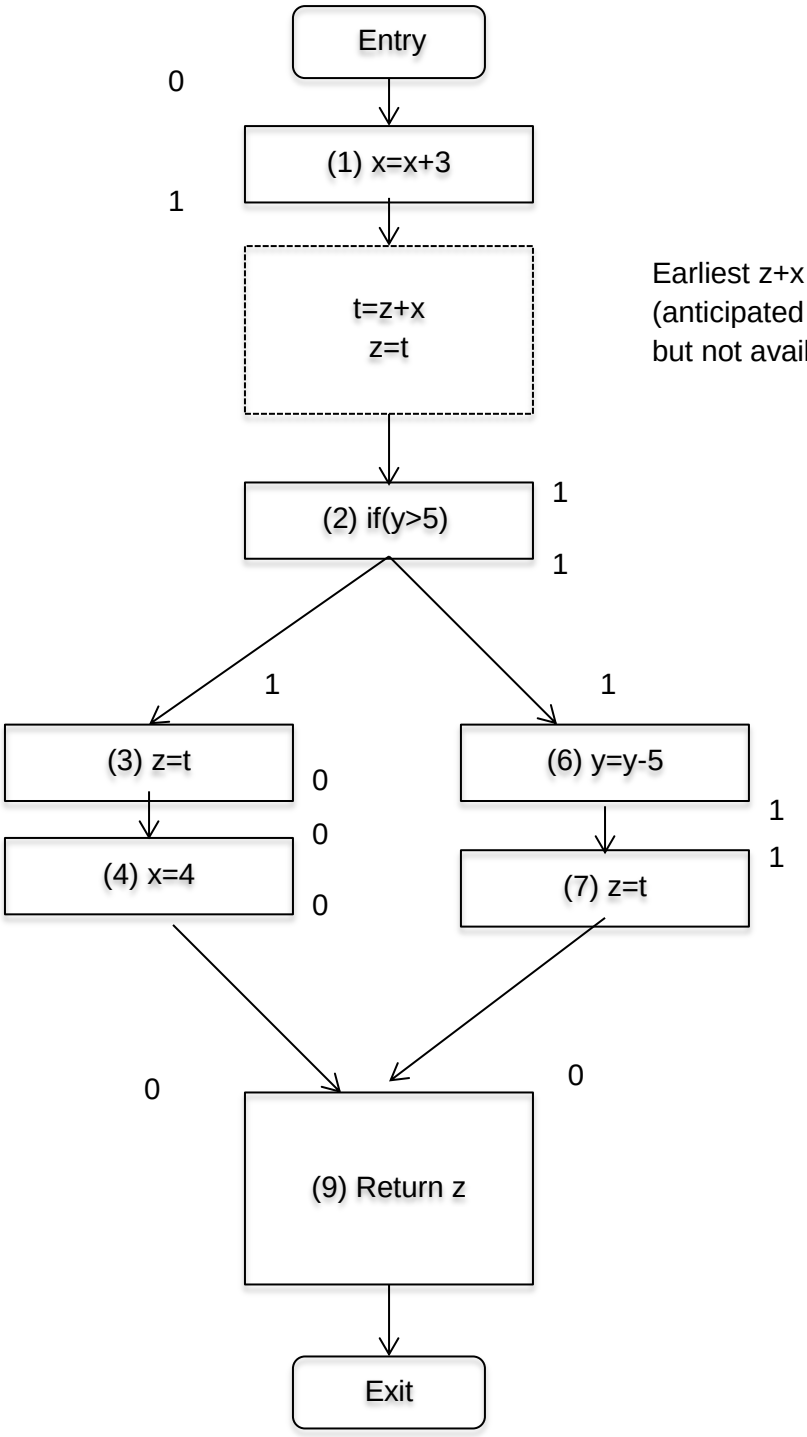


Pass 1: CFG with Anticipation

Values Key: (z+x) / (x+3) / (y-5)

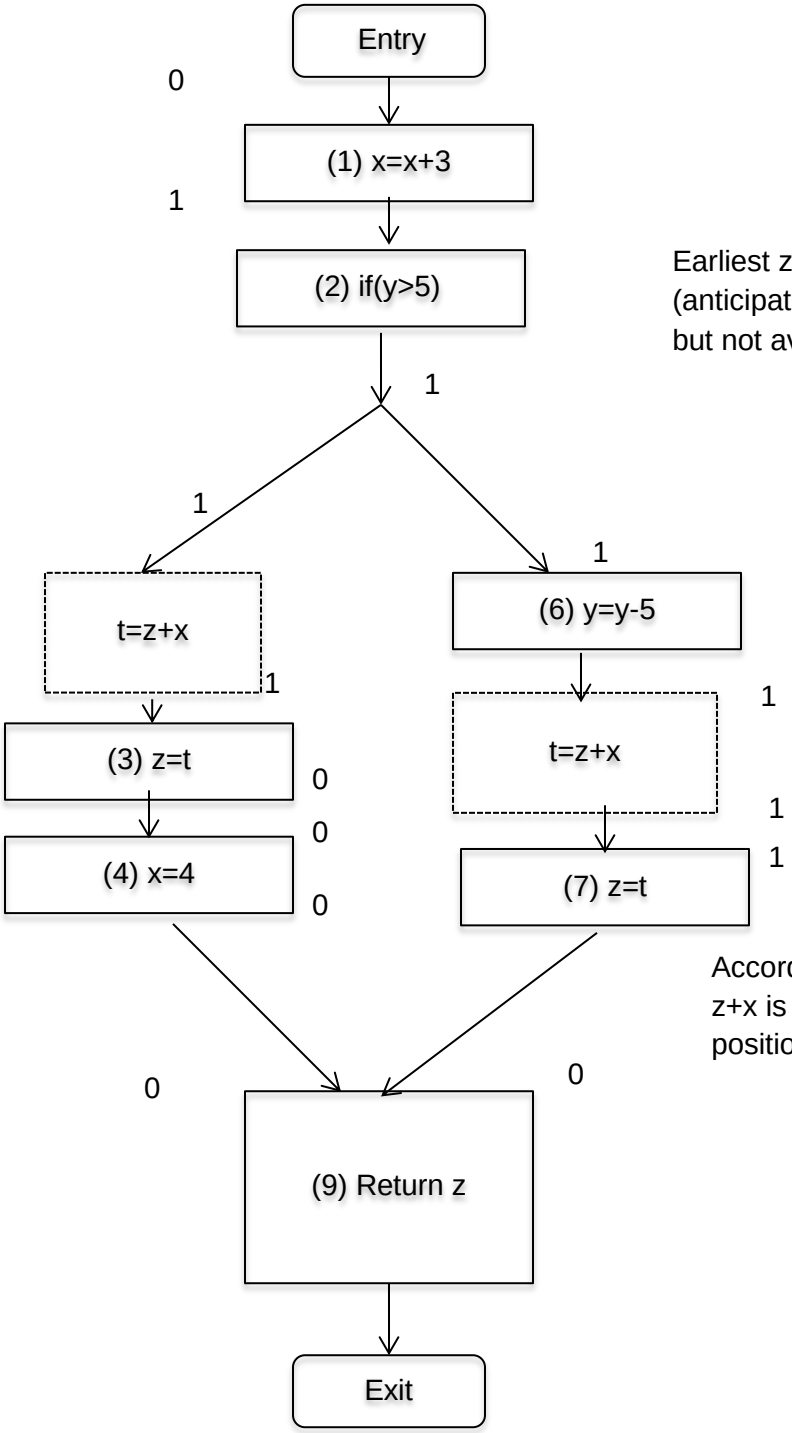


Pass 2: Early Placement Pass



Pass 3: Lazy Code Motion

Place at most postponable position.

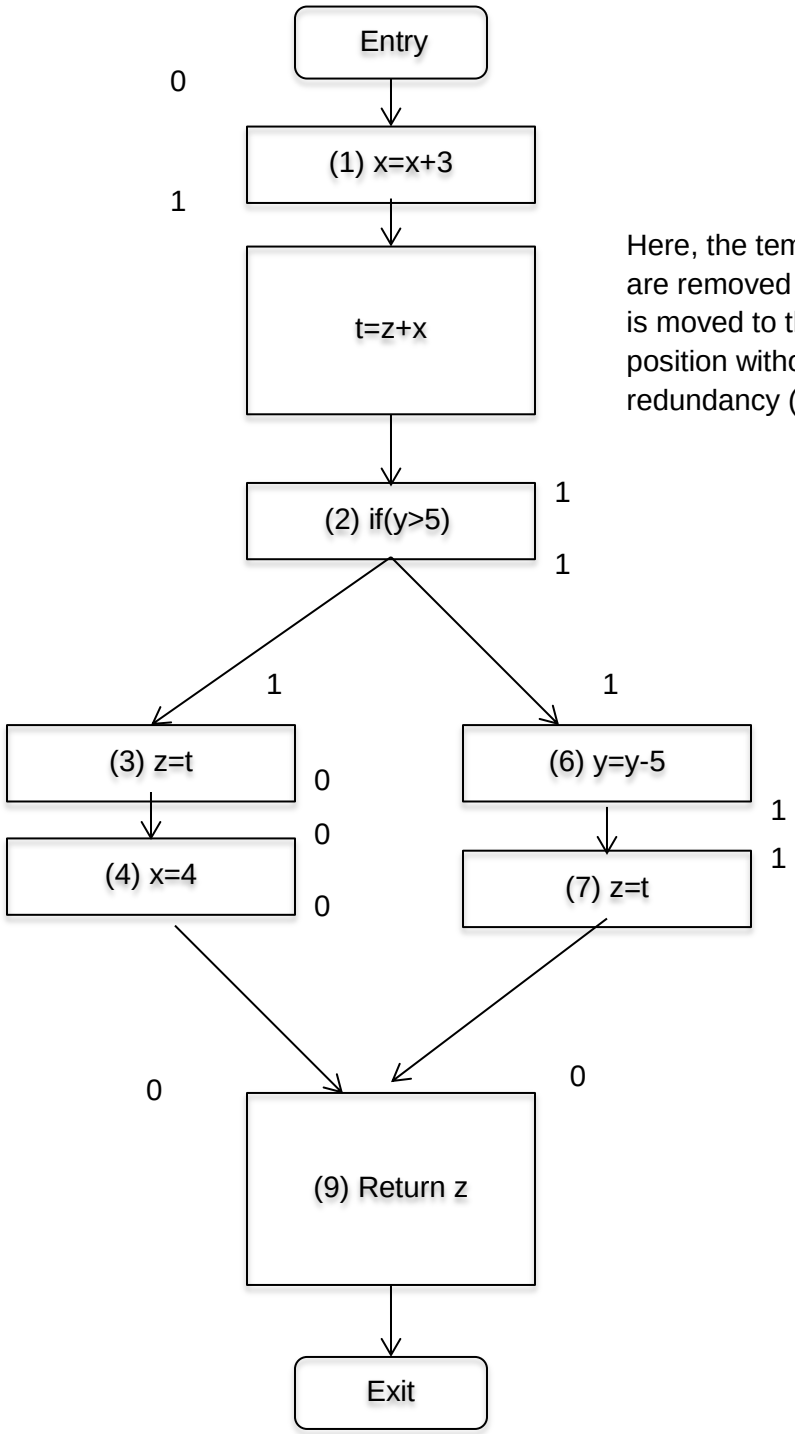


Earliest z+x (anticipated here but not available.)

t=z+x can be postponed to to IN of (3), IN of (6), OUT of (6), IN of (7).

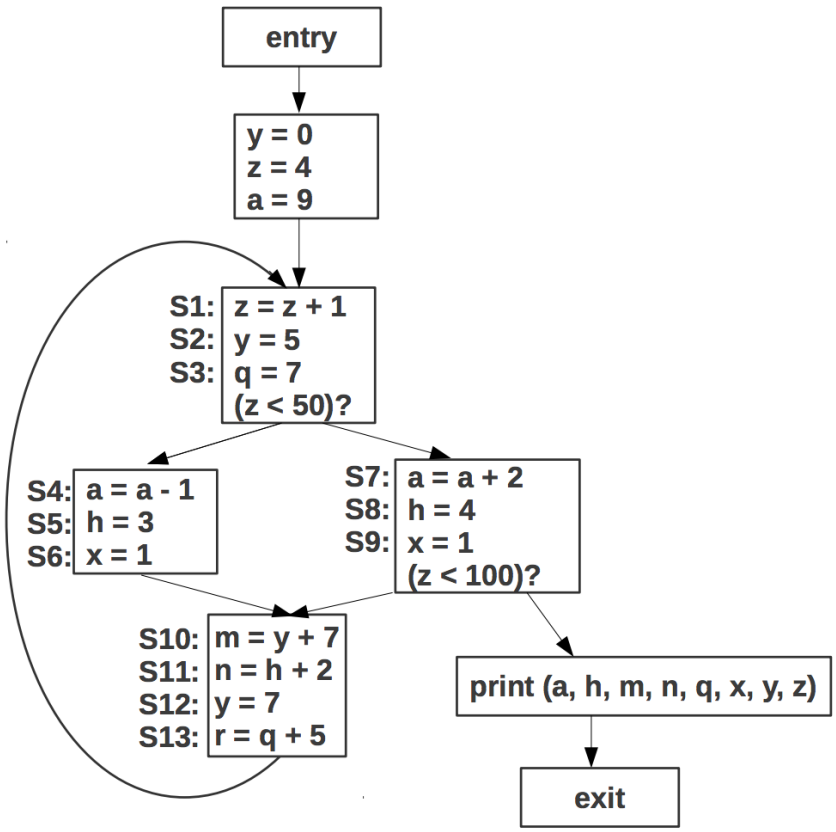
According to the lazy code motion algorithm, z+x is moved to the most postponable position as long as it is not used.

Pass 4: Code Motion and Cleanup Pass



Here, the temporary assignments are removed and the expression is moved to the most postponable position without causing redundancy (latest position).

3.2 Loop Invariant Code Motion



Loop Invariant Code Motion
Reaching Definitions:
Block 1:
IN={s1, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13, d1, d2, d3}
OUT={s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11,, s13, d3}
Block2 (right):
IN={s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s13, d3}
OUT={s1, s2, s3, s4, s5, s6, s10, s11, s13}
Block3 (Left):
IN={s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s13, d3}
OUT={s1, s2, s3, s7, s8, s9, s10, s11, s13}
Block4:
IN={s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s13}
OUT={s1, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13}

(cont'd on next page)

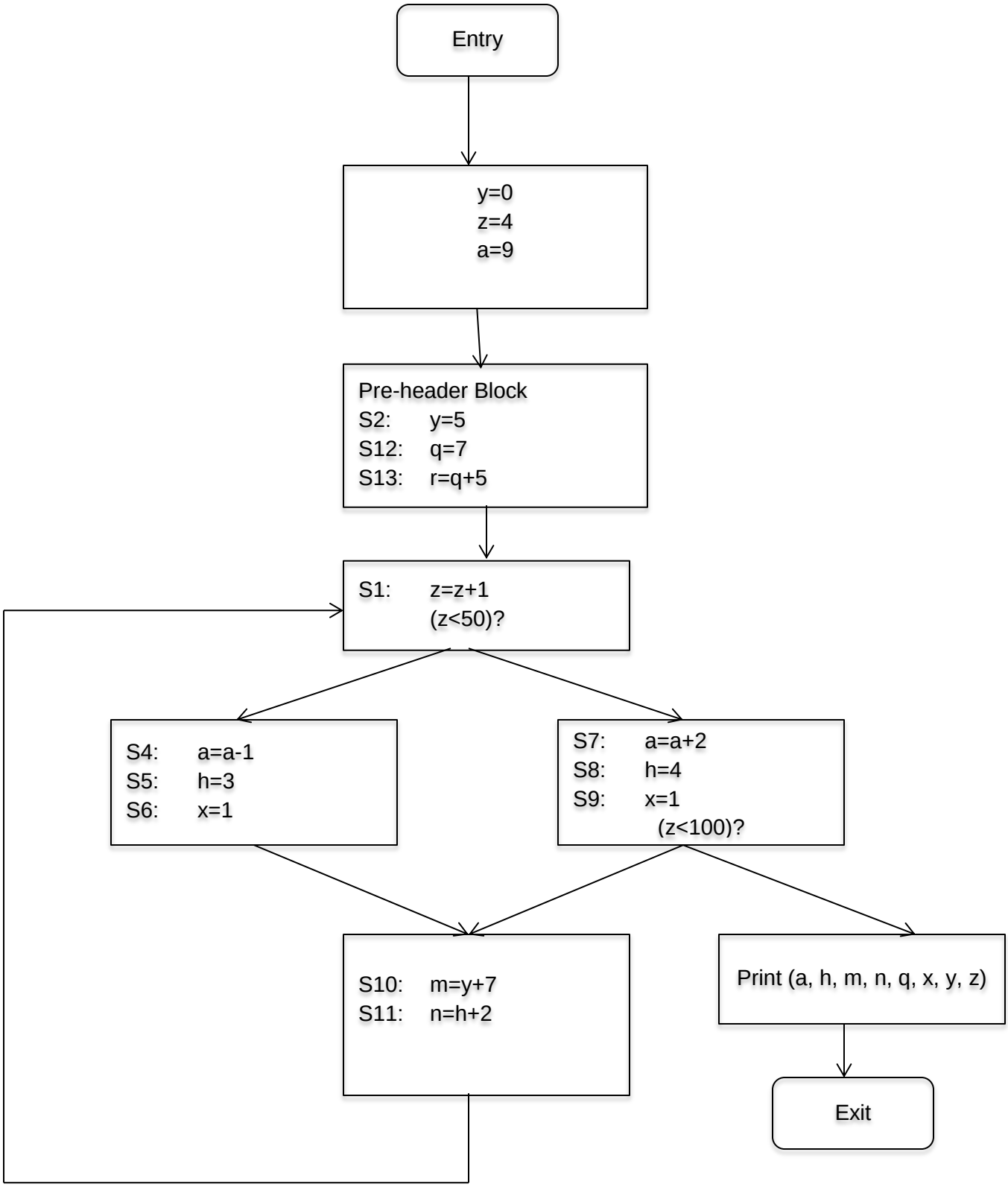
3.2 (cont'd)

Loop Invariant Instructions:
S2, S3, S10, S12, S13

S2 is a constant definition of y.
S3 is a constant definition of q.
S10 is loop invariant since reaching definition of y is invariant.
S12 is a constant definition but it can be removed since it is killed by a redefinition of y at s2 and is not used anywhere else.
S13 is loop invariant since the reaching definition of q is invariant.

Of the 6 only s2, s3, s12 and s13 can be moved to the pre-header by the loop invariant code motion pass.
S10 cannot be moved since there is a path to exit where the previous value of m is reached and is used.

3.2 (cont'd)
After Lazy Code Motion Pass



```
// 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));
    }
  }
}

}
```

```
// 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");

}
```

```
// 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");

}
```