

```
// 15-745 S14 Assignment 2: dataflow.cpp
// Group: aebtekar, auc
////////////////////////////////////

#include "dataflow.h"

// Constructor to define a semi-lattice
Lattice::Lattice(std::vector<std::string> n, bool i)
{
    size = n.size();
    names = n;
    intersect = i;
    top = Elem(size, intersect);
}

// Meet operation is either union or intersection
Elem Lattice::meet(const Elem& elem1, const Elem& elem2)
{
    Elem ret(size);
    for (int i = 0; i < ret.size(); ++i)
    {
        if (intersect)
            ret[i] = elem1[i] & elem2[i];
        else
            ret[i] = elem1[i] | elem2[i];
    }
    return ret;
}

// Print the set of strings corresponding to an element of the lattice
void Lattice::print(Elem elem)
{
    std::cout << '{';
    bool first = true;
    for (int i = 0; i < elem.size(); ++i)
        if (elem[i])
        {
            if (first)
                first = false;
            else
                std::cout << ',';
            std::cout << names[i];
        }
    std::cout << '}' << std::endl;
}

namespace llvm {

void 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 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));
        }
    }

// Data flow analysis in the forward direction
void forwardSearch(Function& F, Lattice* lattice, Elem (*transFun)(Instruction*, Elem))
{
    size_t numBlocks = F.size();
    std::map<BasicBlock*, Elem> in;
    std::map<BasicBlock*, Elem> out;

    // initialization
    for (ilist_iterator<BasicBlock> BI = F.begin(), BE = F.end(); BI != BE; ++BI)
    {
        in[BI] = out[BI] = lattice->top;
    }
    bool change;
    do
    {
        change = false;
        // iterate through blocks in forward order
        for (ilist_iterator<BasicBlock> BI = F.begin(), BE = F.end(); BI != BE; ++BI)
        {
            // in = meet(out(predecessors))
            in[BI] = lattice->top;
            for (pred_iterator PI = pred_begin(BI), PE = pred_end(BI); PI != PE; ++PI)
            {
                BasicBlock* pred = *PI;
                in[BI] = lattice->meet(in[BI], out[pred]);
            }
            // out = transitionFunction(in)
            Elem elem = in[BI];
            for (ilist_iterator<Instruction> II = BI->begin(), IE = BI->end(); II != IE; ++II)
            {
                elem = transFun(II, elem);
            }
            if (out[BI] != elem)
            {
                out[BI] = elem;
                change = true;
            }
        }
    } while (change);

    // print result
    for (ilist_iterator<BasicBlock> BI = F.begin(), BE = F.end(); BI != BE; ++BI)
    {
        Elem elem = in[BI];
        for (ilist_iterator<Instruction> II = BI->begin(), IE = BI->end(); II != IE; ++II)
        {
            if (!isa<PHINode>(*II))
                lattice->print(elem);
            elem = transFun(II, elem);
        }
        lattice->print(elem);
    }

// Data flow analysis in the forward direction
void backwardSearch(Function& F, Lattice* lattice, Elem (*transFun)(Instruction*, Elem))
{
    size_t numBlocks = F.size();
```

```

std::map<BasicBlock*, Elem> in;
std::map<BasicBlock*, Elem> out;

// initialization
for (ilist_iterator<BasicBlock> BI = F.begin(), BE = F.end(); BI != BE; ++BI)
{
    in[BI] = out[BI] = lattice->top;
}
bool change;
do
{
    change = false;
    // iterate through blocks in backward order
    for (ilist_iterator<BasicBlock> BI = F.end(), BE = F.begin(); BI != BE; )
    {
        // out = meet(in(successors))
        out[--BI] = lattice->top;
        for (succ_iterator SI = succ_begin(BI), SE = succ_end(BI); SI != SE; ++SI)
        {
            BasicBlock* succ = *SI;
            out[BI] = lattice->meet(out[BI], in[succ]);
        }
        // in = transitionFunction(out)
        Elem elem = out[BI];
        for (ilist_iterator<Instruction> II = BI->end(), IE = BI->begin(); II != IE; )
        {
            elem = transFun(--II, elem);
        }
        if (in[BI] != elem)
        {
            in[BI] = elem;
            change = true;
        }
    }
}
while (change);

// print result
for (ilist_iterator<BasicBlock> BI = F.begin(), BE = F.end(); BI != BE; ++BI)
{
    std::vector<Elem> elems;
    Elem elem = out[BI];
    elems.push_back(elem);
    for (ilist_iterator<Instruction> II = BI->end(), IE = BI->begin(); II != IE; )
    {
        elem = transFun(--II, elem);
        if (!isa<PHINode>(*II))
            elems.push_back(elem);
    }
    for (int i = elems.size()-1; i >= 0; --i)
        lattice->print(elems[i]);
}
}

```