15-745: Spring 2011

Homework 2

Salil Joshi
salilj@andrew.cmu.edu

Cyrus Omar
cyrus@cmu.edu

1 The Dataflow framework

We developed a general dataflow framework that can be used to implement specific dataflow analyses like Reaching Definitions and Liveness. This framework is implemented as a class `Dataflow`. This class implements both the forward and backward analysis algorithms in a generalized form.

The user of this framework must define a class that inherits from both `Dataflow` and `FunctionPass` (and must register her pass with LLVM as usual). The `Dataflow` class takes a boolean template argument `forward`. If this is true, then dataflow is run in the forward direction, otherwise backward. The class also has as fields the maps `in` and `out` from `BasicBlock*` to `BitVector*` which represent the in and out sets for each block. These may be used by any subclass.

The following functions and data members must be implemented by any subclass of Dataflow:

- `BitVector * top`; This is a `BitVector` that must be initialized by the subclass before the `runOnFunction` method of the parent `Dataflow` class is called. It represents the **top** element of the underlying sub-lattice.

- `void getBoundaryCondition(BitVector*)`. This takes a pointer to a `BitVector`, and modifies this `BitVector` appropriately. For a forward pass, this is the `out[b]` of the Entry node, and for a backward pass it is the `in[b]` of the Exit node.

- `BitVector * initialInteriorPoint(BasicBlock&)`; For a forward pass, this function should return the initial `out[b]` for any block `b`. For a backward pass it should return the initial `in[b]` for any block `b`. Returning a copy of `top` is always correct, but not always the most efficient solution.

- `void meet(BitVector*, const BitVector*);` This function takes the **meet** of the two `BitVector`s passed in as arguments, and sets the first argument to the result.

- `BitVector* transfer(BasicBlock&);` This is the transfer function for the specific analysis to be run.

Note that in the above the `BitVector`s for top, the boundary conditions and initial interior points must all be distinct (i.e. must not point to the same object).

When the subclass wishes to run the dataflow pass, it must call the `runOnFunction` method of its parent `Dataflow` class, passing in a reference to the `Function` on which the pass is to be run.

2 Reaching Definitions

We made use of the dataflow framework described above to implement Reaching Definition analysis. This is a forward function pass, so the class `ReachingDefinitions` is a subclass of `FunctionPass` and of `Dataflow<true>`.

Since the `in` and `out` sets are maintained as `BitVector`s, we maintain a bidirectional map from definitions to their index in the `BitVector`. This map is implemented by the two fields `index` and `r_index`. In order to initialize these three fields correctly, we go through all instructions once before running the actual dataflow algorithm, in order to give each definition an index.

We also store a mapping (called `instOut`) from each instruction to the set of reaching definitions for the program point just after the instruction. This is not needed for the analysis itself, but it is needed to display the output, since we wish to display the reaching definitions at every program point. `in` and `out` are not sufficient for this purpose.

This pass then calls `runOnFunction` on its parent `Dataflow`, and then prints the results.

The functions and fields required by the dataflow framework are initialized as follows:

- `top`: This is a `BitVector` of size `maxIndex+1`, with all bits set to `false`. It represents the empty set.

- `getBoundaryCondition`: This is the same as `top`

- `initialInteriorPoint`: This is also the same as `top` for every block

- `meet`: Since the meet in the underlying sub-lattice is union, this function simply takes the bitwise OR of the two `BitVector`s passed in.
- `transfer`: This function iterates through the instructions in the block, and does the following:
  - For each instruction, it stores the out set for this instruction in `instOut`
  - If this instruction is a definition, it sets the corresponding bit in the result `BitVector`. The `index` map is used to find the index corresponding to this definition in the `BitVector`
  - If it is a phi node, it unsets the corresponding bit for the wrapped instructions since the phi node hides it (until it shows up again later in the block at least.)
- `displayResults`: This function displays the results.
  - It displays the the entry node's information first.
  - It then iterates through the blocks, displaying their `out` before going through the instructions and showing their outs.
  - Finally, after the last instruction it displays the final instruction's out.

3 Liveness

This is a backwards dataflow, so the class `Liveness` is a subclass of `FunctionPass` and of `Dataflow<false>`.

It is structured similarly to the above. We store a mapping, `instIn`, from each instruction to the set of reaching defintions for the program point just before the instruction, rather than the other way around, since our meet points are on successors.

- `meet`: Again implemented as union.
- `transfer`: We iterate over the instructions in reverse, removing the instruction from the bit vector if it is part of it and adding the operands. Phi instructions are treated specially: phi instructions for the current block are added to the internal base case for `out` so that they propagate properly internal to the block, then removed from `in` at the end so they do not leak into other blocks.
- `displayResults`: Again structured similarly with some inversions:

– Again displays the entry node first.
– Then iterates over blocks and instructions in the blocks, displaying their `in` information this time. The first instruction is skipped because the out node will capture it.
– The out node is displayed after this.
– A final out node is displayed at the end.
– We have to check for phi nodes in the next block this time to make sure they are not displayed.

4 Source Listing

4.1 Framework

```
/** CMU 15-745: Optimizing Compilers
    Spring 2011
    Salil Joshi and Cyrus Omar
 **/

#include "llvm/Pass.h"
#include "llvm/BasicBlock.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Instruction.h"
#include "llvm/Instructions.h"
#include "llvm/Constants.h"
#include "llvm/ADT/APFloat.h"
#include "llvm/Transforms/Utils/BasicBlockUtils.h"
#include "llvm/ADT/BitVector.h"
#include "llvm/ADT/ValueMap.h"
#include "llvm/Support/CFG.h"

#include <ostream>
#include <list>

using namespace llvm;

namespace
{
    template<bool forward>
    struct Dataflow
    {
        Dataflow() {
          in = new DomainMap();
          out = new DomainMap();
        }

      typedef ValueMap<BasicBlock*, BitVector*> DomainMap;

      // in[b] where b is a basic block
      DomainMap *in;

      // out[b] where out is a basic block
      DomainMap *out;

// bitvector such that meet(x, top) = x
// must be specified by subclass
      BitVector *top;

        ~Dataflow() {
         for (DomainMap::iterator i = in->begin(), ie = in->end(); i != ie; i++) {
             delete i->second;
         }
         for (DomainMap::iterator i = out->begin(), ie = out->end(); i != ie; i++) {
             delete i->second;
           }
           delete top;
        }

        virtual bool runOnFunction(Function &f) {

            BasicBlock& entry = f.getEntryBlock();

            // initialize in and out
            for (Function::iterator bi = f.begin(), be = f.end(); bi != be; bi++) {
                if (forward) {
                 /* Forward flow means we need to first apply meet with out[b]
          for all incoming blocks, b.

          With loops, out[b] may not always have been generated already,
          so we need initial interior points. */
       (*out)[&(*bi)] = initialInteriorPoint(*bi);

       /* just a dummy bitvector of the same length as top,
          it will never be read before being written to */
       (*in)[&(*bi)] = new BitVector(*top);
      } else {
       // opposite logic for reverse flow
       (*in)[&(*bi)] = initialInteriorPoint(*bi);
       (*out)[&(*bi)] = new BitVector(*top);

       // there isn't a unique exit node so we apply the boundary condition
       // when we reach a node with no successors in the loop below...
      }
 }

if (forward)
   // boundary conditions for entry node
   getBoundaryCondition((*in)[&entry]);


 /* worklist maintains a list of all basic blocks on whom the transfer
  * function needs to be applied.*/
 std::list<BasicBlock*> *worklist = new std::list<BasicBlock*>();

 //Initially, every node is in the worklist
 bfs(f,*worklist);

 if (!forward)
   //for backward passes, start at exit nodes and work backwards
   worklist->reverse();

 while (!worklist->empty()) {
   if (forward) {
     reversePostOrder(*worklist);
```

```
      } else {
        postOrder(*worklist);
      }
    }

    delete worklist;
    return false;
  }

  virtual void bfs(Function &f, std::list<BasicBlock*> &worklist) {
    BasicBlock * curNode;
    ValueMap<BasicBlock*, bool> *visited = new ValueMap<BasicBlock*,bool>();

    for (Function::iterator bb = f.begin(), be = f.end(); bb != be; bb++) {
      (*visited)[&*bb] = false;
    }

    std::list<BasicBlock*> *l = new std::list<BasicBlock*>();

    l->push_back(&f.getEntryBlock());
    while (!l->empty()) {
      curNode = *(l->begin());
      l->pop_front();
      worklist.push_back(curNode);

      (*visited)[curNode] = true;
      for (succ_iterator SI = succ_begin(curNode), SE = succ_end(curNode); SI !
        if (!(*visited)[*SI]) {
          l->push_back(*SI);
        }
      }
    }

    delete visited;
    delete l;
  }

  virtual void reversePostOrder(std::list<BasicBlock*> &q) {
```

```
  BasicBlock * curNode = *q.begin();
  q.pop_front();

  pred_iterator PI = pred_begin(curNode),
                PE = pred_end(curNode);
  if (PI != PE) {
    // begin with a copy of out[first predecessor]
    *(*in)[curNode] = *(*out)[*PI];

    // fold meet over predecessors
    for (PI++; PI != PE; PI++) {
      meet((*in)[curNode], (*out)[*PI]);
    }
  } // (otherwise entry node, in[entry] already set above)

  // apply transfer function
  BitVector* newOut = transfer(*curNode);
  if (*newOut != *(*out)[curNode]) {
    // copy new value
    *(*out)[curNode] = *newOut;
    for (succ_iterator SI = succ_begin(curNode), SE = succ_end(curNode); SI !
      q.push_back(*SI);
    }
  }
  delete newOut;
}

virtual void postOrder(std::list<BasicBlock*> &q) {
  BasicBlock *curNode = *q.begin();
  q.pop_front();

  succ_iterator SI = succ_begin(curNode), SE = succ_end(curNode);
  if (SI != SE) {
    // begin with a copy of in[first successor]
    *(*out)[curNode] = *(*in)[*SI];

    // fold meet operator over successors
    for (SI++; SI != SE; SI++) {
```

```
                meet((*out)[curNode], (*in)[*SI]);
              }
            } else {
              // boundary condition when it is an exit block
              getBoundaryCondition((*out)[curNode]);
            }

            // apply transfer function
            BitVector* newIn = transfer(*curNode);
            if (*newIn != *(*in)[curNode]) {
              // copy new value
              *(*in)[curNode] = *newIn;
              for (pred_iterator PI = pred_begin(curNode), PE = pred_end(curNode); PI !
                q.push_back(*PI);
              }
            }
            delete newIn;
          }

          virtual void getBoundaryCondition(BitVector*) = 0;
          virtual void meet(BitVector*, const BitVector*) = 0;
          virtual BitVector * initialInteriorPoint(BasicBlock&) = 0;
          virtual BitVector* transfer(BasicBlock&) = 0;
    };
}
```

4.2 Reaching Definitions

```
/** CMU 15-745: Optimizing Compilers
    Spring 2011
    Salil Joshi and Cyrus Omar
 **/

#include "llvm/Pass.h"
#include "llvm/BasicBlock.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Instruction.h"
```

```
#include "llvm/Instructions.h"
#include "llvm/Constants.h"
#include "llvm/ADT/APFloat.h"
#include "llvm/Transforms/Utils/BasicBlockUtils.h"
#include "llvm/ADT/BitVector.h"
#include "llvm/ADT/ValueMap.h"
#include "llvm/Support/CFG.h"
#include "llvm/Support/InstIterator.h"
#include "llvm/Assembly/Writer.h"

#include "dataflow.cpp"

#include <ostream>

using namespace llvm;

namespace
{
    struct ReachingDefinitions : public Dataflow<true>, public FunctionPass
    {
        static char ID;

        ReachingDefinitions() : Dataflow<true>(), FunctionPass(ID) {
          index = new ValueMap<Value*, int>();
          r_index = new std::vector<Value*>();
          instOut = new ValueMap<Instruction*, BitVector*>();
        }

        // map from instructions/argument to their index in the bitvector
        ValueMap<Value*, int> *index;

        // map from index in bitvector back to instruction/argument
        std::vector<Value*> *r_index;

        // convenience
        int numTotal;
        int numArgs;
```

```
// map from instructions to bitvector corresponding to program point AFTER that i
    ValueMap<Instruction*, BitVector*> *instOut;


    virtual void meet(BitVector *op1, const BitVector *op2) {
     // union
      *op1 |= *op2;
    }

    virtual void getBoundaryCondition(BitVector *entry) {
     // in[b] = just the arguments if no predecessors / entry node
      *entry = BitVector(numTotal, false);
      for (int i=0; i < numArgs; ++i) {
      (*entry)[i] = true;
      }
    }

    bool isDefinition(Instruction *ii) {
      // All other types of instructions are definitions
      return (!(isa<TerminatorInst>(ii) || isa<StoreInst>(ii) || (isa<CallInst>(i
    }

    BitVector* initialInteriorPoint(BasicBlock& bb) {
     // out[b] = empty set initially
      return new BitVector(numTotal, false);
    }

    virtual bool runOnFunction(Function &F) {
     numTotal = 0;
     numArgs = 0;

     // add function arguments to maps
     for (Function::arg_iterator ai = F.arg_begin(), ae = F.arg_end(); ai != ae;
     (*index)[&*ai] = numArgs;
     r_index->push_back(&*ai);
     numArgs++;
     }
      numTotal = numArgs;
```

```
 // add definitions to maps
 for (inst_iterator ii = inst_begin(&F), ie = inst_end(&F); ii != ie; ii++) {
 if (isDefinition(&*ii)) {
 (*index)[&*ii] = numTotal;
 r_index->push_back(&*ii);
 numTotal++;
 }
 }

  // initialize instOut
  for (inst_iterator ii = inst_begin(&F), ie = inst_end(&F); ii != ie; ii++)
    (*instOut)[&*ii] = new BitVector(numTotal, false);
  }
 top = new BitVector(numTotal, false);

  // run data flow
 Dataflow<true>::runOnFunction(F);

  // print out instructions with reaching variables between each instruction
 displayResults(F);

  // didn't modify nothing
 return false;
}

virtual BitVector* transfer(BasicBlock& bb) {
  // we iterate over instructions beginning with in[bb]
  BitVector* prev = (*in)[&bb];

  // temporary variables for convenience
  BitVector* instVec = prev; // for empty blocks
  Instruction* inst;

  for (BasicBlock::iterator ii = bb.begin(), ie = bb.end(); ii != ie; ii++) {
    // begin with previous reaching definitions
    inst = &*ii;
    instVec = (*instOut)[inst];
```

```
      *instVec = *prev;

      // if this instruction is a new definition, add it
      if (isDefinition(inst))
        (*instVec)[(*index)[inst]] = true;

      // if it is a phi node, kill the stuff
      if (isa<PHINode>(inst)) {
        PHINode* p = cast<PHINode>(inst);
        unsigned num = p->getNumIncomingValues();
        for (int i=0; i < num; ++i) {
          Value* v = p->getIncomingValue(i);
          if (isa<Instruction>(v) || isa<Argument>(v)) {
            (*instVec)[(*index)[v]] = false;
          }
        }
      }

      prev = instVec;
    }

    // return a copy of the final instruction's post-condition to put in out[bb
    return new BitVector(*instVec);
  }

  virtual void displayResults(Function &F) {
    // iterate over basic blocks
    Function::iterator bi = F.begin(), be = (F.end());
    for (; bi != be; bi++) {
      errs() << bi->getName() << ":\n"; //Display labels for basic blocks
      // display in[bb]
      if (!isa<PHINode>(*(bi->begin())))
        printBV( (*in)[&*bi] );

      // iterate over remaining instructions except very last one
      // we don't print out[i] for the last one because we should actually prin
      // result of the meet operator at those points, i.e. in[next block]...
      BasicBlock::iterator ii = bi->begin(), ie = --(bi->end());
```

```
          for (; ii != ie; ii++) {
            errs() << "\t" << *ii << "\n";
            if (!isa<PHINode>(*(++ii))) {
              --ii;
              printBV( (*instOut)[&*ii] );
            } else --ii;

          }
          errs() << "\t" << *(ii) << "\n";
          errs() << "\n";
        }
        // ...unless there are no more blocks
        printBV( (*out)[&*(--be)] );
      }

      virtual void printBV(BitVector *bv) {
        errs() << "{ ";
        for (int i=0; i < numTotal; i++) {
          if ( (*bv)[i] ) {
            WriteAsOperand(errs(), (*r_index)[i], false);
            errs() << " ";
          }
        }
        errs() << "}\n";
      }

    };

    char ReachingDefinitions::ID = 0;
    static RegisterPass<ReachingDefinitions> x("ReachingDefinitions", "ReachingDefini
}
```

4.3 Liveness

```
/** CMU 15-745: Optimizing Compilers
    Spring 2011
    Salil Joshi and Cyrus Omar
```

```
 **/
#include "llvm/Pass.h"
#include "llvm/BasicBlock.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Instruction.h"
#include "llvm/Instructions.h"
#include "llvm/Constants.h"
#include "llvm/ADT/APFloat.h"
#include "llvm/Transforms/Utils/BasicBlockUtils.h"
#include "llvm/ADT/BitVector.h"
#include "llvm/ADT/ValueMap.h"
#include "llvm/Support/CFG.h"
#include "llvm/Support/InstIterator.h"
#include "llvm/Assembly/Writer.h"

#include "dataflow.cpp"

#include <ostream>

using namespace llvm;

namespace
{
    struct Liveness : public Dataflow<false>, public FunctionPass
    {
        static char ID;

        Liveness() : Dataflow<false>(), FunctionPass(ID) {
          index = new ValueMap<Value*, int>();
          r_index = new std::vector<Value*>();
          instIn = new ValueMap<Instruction*, BitVector*>();
        }

        // map from instructions/argument to their index in the bitvector
        ValueMap<Value*, int> *index;

        // map from index in bitvector back to instruction/argument
        std::vector<Value*> *r_index;
```

```
// convenience
int numTotal;
int numArgs;

// map from instructions to bitvector corresponding to program point BEFORE t
ValueMap<Instruction*, BitVector*> *instIn;

virtual void meet(BitVector *op1, const BitVector *op2) {
  // union
  *op1 |= *op2;
}

virtual void getBoundaryCondition(BitVector *entry) {
  // out[b] = empty set if no successors
  *entry = BitVector(numTotal, false);
}

bool isDefinition(Instruction *ii) {
  return (!(isa<TerminatorInst>(ii) || isa<StoreInst>(ii) || (isa<CallInst>(i
}

BitVector* initialInteriorPoint(BasicBlock& bb) {
  // in[b] = empty set initially
  return new BitVector(numTotal, false);
}

virtual bool runOnFunction(Function &F) {
  numTotal = 0;
  numArgs = 0;

  // add function arguments to maps
  for (Function::arg_iterator ai = F.arg_begin(), ae = F.arg_end(); ai != ae;
    (*index)[&*ai] = numArgs;
    r_index->push_back(&*ai);
    numArgs++;
  }
  numTotal = numArgs;
```

```
    // add definitions to maps
    for (inst_iterator ii = inst_begin(&F), ie = inst_end(&F); ii != ie; ii++)
      if (isDefinition(&*ii)) {
        (*index)[&*ii] = numTotal;
        r_index->push_back(&*ii);
        numTotal++;
      }
    }

    // initialize instIn
    for (inst_iterator ii = inst_begin(&F), ie = inst_end(&F); ii != ie; ii++)
      (*instIn)[&*ii] = new BitVector(numTotal, false);
    }
    top = new BitVector(numTotal, false);

    // run data flow
    Dataflow<false>::runOnFunction(F);

    // print out instructions with reaching variables between each instruction
    displayResults(F);

    // didn't modify nothing
    return false;
  }

  virtual BitVector* transfer(BasicBlock& bb) {
    // we iterate over instructions in reverse beginning with out[bb]
    BitVector* next = new BitVector(*((*out)[&bb]));

    // temporary variables for convenience
    BitVector* instVec = next; // for empty blocks
    Instruction* inst;

    // add local phi-captured nodes to out
    for (BasicBlock::iterator ii = bb.begin(), ib = bb.end(); ii != ib; ++ii) {
      if (isa<PHINode>(*ii)) {
        PHINode* phiInst = cast<PHINode>(&*ii);
        unsigned idx = phiInst->  getBasicBlockIndex (&bb);
        if (idx < phiInst->getNumIncomingValues()){
          Value* v = phiInst -> getIncomingValue(idx);
          if (isa<Instruction>(v) || isa<Argument>(v))
          {
            (*next)[(*index)[v]] = true;
          }
        }
      }
    }

    // go through instructions in reverse
    BasicBlock::iterator ii = --(bb.end()), ib = bb.begin();
    while (true) {

      // inherit data from next instruction
      inst = &*ii;
      instVec = (*instIn)[inst];
      *instVec = *next;

      // if this instruction is a new definition, remove it
      if (isDefinition(inst))
        (*instVec)[(*index)[inst]] = false;

      // add the arguments, unless it is a phi node
      if (!isa<PHINode>(*ii)) {
      User::op_iterator OI, OE;
      for (OI = inst->op_begin(), OE=inst->op_end(); OI != OE; ++OI) {
        if (isa<Instruction>(*OI) || isa<Argument>(*OI)) {
          (*instVec)[(*index)[*OI]] = true;
        }
      }
      }
      next = instVec;

      if (ii == ib) break;
      --ii;
    }
```

```
  // remove the phi nodes from in
  instVec = new BitVector(*instVec);
  for (BasicBlock::iterator ii = bb.begin(), ib = bb.end(); ii != ib; ++ii) {
    if (isa<PHINode>(*ii)) {
      PHINode* phiInst = cast<PHINode>(&*ii);
      unsigned idx = phiInst->  getBasicBlockIndex (&bb);
      if (idx < phiInst->getNumIncomingValues()){
        Value* v = phiInst -> getIncomingValue(idx);
        if (isa<Instruction>(v) || isa<Argument>(v))
        {
          (*next)[(*index)[v]] = false;
        }
      }
    }
  }
  return instVec;
}

virtual void displayResults(Function &F) {
  // iterate over basic blocks
  Function::iterator bi = F.begin(), be = (F.end());
  printBV( (*out)[&*bi] ); // entry node
  for (; bi != be; ) {
    errs() << bi->getName() << ":\n"; //Display labels for basic blocks

    // iterate over remaining instructions except very first one
    BasicBlock::iterator ii = bi->begin(), ie = (bi->end());
    errs() << "\t" << *ii << "\n";
    for (ii++; ii != ie; ii++) {
      if (!isa<PHINode>(*(ii))) {
        printBV( (*instIn)[&*ii] );
      }
      errs() << "\t" << *ii << "\n";
    }

    // display in[bb]
    ++bi;
```

```
          if (bi != be && !isa<PHINode>(*((bi)->begin())))
            printBV( (*out)[&*bi] );

          errs() << "\n";
        }
        printBV( (*out)[&*(--bi)] );
      }

      virtual void printBV(BitVector *bv) {
        errs() << "{ ";
        for (int i=0; i < numTotal; i++) {
          if ( (*bv)[i] ) {
            WriteAsOperand(errs(), (*r_index)[i], false);
            errs() << " ";
          }
        }
        errs() << "}\n";
      }

    };

    char Liveness::ID = 0;
    static RegisterPass<Liveness> x("Liveness", "Liveness", false, false);
}
```