15-745: Spring 2011

Homework 1

Salil Joshi
salilj@andrew.cmu.edu

Cyrus Omar
cyrus@cmu.edu

1 Using FunctionInfo Results

1. Function signatures, as used for forward references, are represented as functions within the LLVM IR. Since signatures have no body, they have zero basic blocks.

2. The number of call sites a function has is important in determining whether inlining it will increase the size of the assembly unacceptably. If this number is small, inlining can occur safely. The threshold is higher for functions with fewer instructions.

3. The number of call sites can be a heuristic indicator of the relative importance of a function in a program. The number of basic blocks can similarly serve as a heuristic measure of the complexity of control flow through a function.

   If the number of basic blocks is small, it may be fruitful to use asymptotically inefficient optimization techniques because the absolute magnitude of time it takes remains small.

   The number of instructions can be useful in estimating the amount of time it may take to run a pass on the function.

2 Implementation

We created a per-function pass (to facilitate constant propagation within the function) and iterated through each instruction, dispatching on its instruction type with a `switch` statement.

1. **Constant Propagation** When a constant was stored in a variable, we found places where it was being loaded into a register, and replaced all uses of the register with the constant by going through its use list.

2. **Algebraic Identities** We check mainly for two types of algebraic identities: $a \oplus \text{identity} = a$ and $a \oplus \text{zero} = \text{zero}$, where *identity* and *zero* may differ for each operator $\oplus$ (and not all operators have both). Depending on the instructions opcode, we choose an identity element and/or a zero element, and check if the instruction is of the same form as the LHS of these two equations, and if so, we substitute the RHS.

   Thus on the whole the logic is the same for all operators, although considerations such as commutativity do come into play. Furthermore, we also took into account some identities that are not of the forms given above (eg., for the remainder operators).

3. **Constant Folding** When both operands of an arithmetic expression were constants, we replaced the instruction with a value resulting from computing the operation statically. We wrote functions taking the operation and operands, one for integer operations and another for floating point operations, to produce this value and used a common function to insert this into the function.

   We note that floating point constant folding can produce exceptional conditions. We did not implement any mechanism for notifying the user of these conditions or aborting the constant folding operation in this condition.

4. **Strength Reduction** We detected uses of the multiplication operation on integers. When one of the operands was a power of two, we transformed the instruction into an equivalent left shift instruction, taking the log of that operand as the second argument.

3 Source Listing: FunctionInfo

```
#include "llvm/Pass.h"
#include "llvm/Function.h"
#include "llvm/Module.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/CallSite.h"

#include <ostream>
#include <fstream>
```

```
#include <iostream>

using namespace llvm;

namespace
{

  struct FInfo {
    StringRef name;
    size_t args;
    size_t blocks;
    size_t insts;
    int calls;
  };

class FunctionInfo : public ModulePass
{
  //output the function information to a file
  void printFunctionInfo(Module& M)
  {
    std::string name = M.getModuleIdentifier() + ".finfo";
    std::ofstream file(name.c_str());

  }

public:
static char ID;

FunctionInfo() :
  ModulePass(ID)
{
}

~FunctionInfo()
{
}

  // We don't modify the program, so we preserve all analyses
  virtual void getAnalysisUsage(AnalysisUsage &AU) const
  {
    AU.setPreservesAll();
  }

  virtual bool runOnFunction(Function &F)
  {
    FInfo info;
```

```
    //implement this
    info.name = F.getName();
    info.args = F.arg_size();

    info.calls = 0;
    for (Value::use_iterator U = F.use_begin(); U != F.use_end(); ++U) {
      if (Instruction *I = dyn_cast<Instruction>(*U)) {
        CallSite CS(I);
        if (CS.getCalledValue() == &F) {
          info.calls++;
        }
      }
    }

    info.blocks = F.size();
    info.insts = 0;
    for (Function::iterator B = F.begin(); B != F.end(); ++B) {
      info.insts += B->size();
    }

    errs() << info.name << "\t" << info.args << "\t" << info.calls << "\t"
      << info.blocks << "\t" << info.insts << "\n";
    return false;
  }

  virtual bool runOnModule(Module& M)
  {
    errs() << "Name\t#Args\t#Calls\t#Blocks\t#Insts\n";
    for (Module::iterator MI = M.begin(), ME = M.end(); MI != ME; ++MI)
      {
runOnFunction(*MI);
      }
    printFunctionInfo(M);
    return false;
  }
};

char FunctionInfo::ID = 0;
RegisterPass<FunctionInfo> X("function-info", "15745: Functions Information");
}
```

4 Source Listing: LocalOpts

```
#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 <ostream>

using namespace llvm;

APFloat::roundingMode rMode = APFloat::rmNearestTiesToEven;
namespace
{
  struct OptInfo {
      unsigned constProp;
      unsigned constFold;
      unsigned algebraic;
      unsigned strengthRed;
  };


  template<class C, class AP>
    bool valEquals (C *L, AP* R);

  template<>
    bool valEquals (ConstantInt *L, APInt *R)
      {return L->getValue().eq(*R);}
  template<>
    bool valEquals (ConstantFP *L, APFloat *R) {return L->getValueAPF().compare(*R) == APFloat::cmpE

  struct LocalOpts : public FunctionPass
  {
    static char ID;
    LocalOpts() : FunctionPass(ID) {}


    template<class C, class AP>
      Value * constIdentity (Value *L, Value * R, AP *identity, AP *zero) {
        if (C *LC = dyn_cast<C>(L)) {
          if (identity && valEquals<C,AP>(LC,identity)) {
            // identity op R = R
            return R;
          } else if (zero && valEquals<C,AP>(LC,zero)) {
            // zero op R = zero
            return C::get(LC->getContext(), *zero);
```

```
      }
    }
    return NULL;
  }

template<class C, class AP>
  Value * remIdentity (Value *L, Value * R, AP *unit, AP *zero) {
    // a mod unit = zero
    if (C *RC = dyn_cast<C>(R)) {
      if (unit && valEquals<C,AP>(RC,unit)) {
        return C::get(RC->getContext(), *zero);
      }
    }
    return NULL;
  }

template<class C, class AP>
  Value * commIdentities (Value *L, Value * R, AP *identity, AP *zero) {
    // apply constIdentity to L and R, then swap left and right operands.
    // Only for commutative operators.
    if (Value * changedVal = constIdentity<C,AP>(L,R,identity,zero)) {
      return changedVal;
    } else if (Value * changedVal = constIdentity<C,AP>(R,L,identity,zero)) {
      return changedVal;
    } else {
      return NULL;
    }
  }

template<class C>
  Value * selfInverse (Value *L, Value *R, C * zero) {
    if (cast<Instruction>(L)->isIdenticalTo(cast<Instruction>(R))) {
      // L - L = zero
      return zero;
    } else return NULL;
  }


// Replace all uses of an instruction with the value val. Also delete instruction.
// At the end, the iterator i points to the instruction just before the one that was removed.
void replaceUsesAndDelete(BasicBlock::iterator &i, Value * val) {
  i->replaceAllUsesWith(val);
  BasicBlock::iterator j = i;
  ++i;
  j->eraseFromParent();
  --i;
```

```
}

// For constant folding integer operators. Simply perform the relevant operation on the operands
ConstantInt* evalBinaryIntOp(unsigned op, ConstantInt * left, ConstantInt * right) {
  switch (op) {
    case Instruction::Add:
      return ConstantInt::get(left->getContext(),
          left->getValue() + right->getValue());
    case Instruction::Sub:
      return ConstantInt::get(left->getContext(),
          left->getValue() - right->getValue());
    case Instruction::Mul:
      return ConstantInt::get(left->getContext(),
          left->getValue() * right->getValue());
    case Instruction::UDiv:
      return ConstantInt::get(left->getContext(),
          left->getValue().udiv(right->getValue()));
    case Instruction::SDiv:
      return ConstantInt::get(left->getContext(),
          left->getValue().sdiv(right->getValue()));
    case Instruction::URem:
      return ConstantInt::get(left->getContext(),
          left->getValue().urem(right->getValue()));
    case Instruction::SRem:
      return ConstantInt::get(left->getContext(),
          left->getValue().srem(right->getValue()));
    case Instruction::Shl:
      return ConstantInt::get(left->getContext(),
          left->getValue().shl(right->getValue()));
    case Instruction::LShr:
      return ConstantInt::get(left->getContext(),
          left->getValue().lshr(right->getValue()));
    case Instruction::AShr:
      return ConstantInt::get(left->getContext(),
          left->getValue().ashr(right->getValue()));
    case Instruction::And:
      return ConstantInt::get(left->getContext(),
          left->getValue() & right->getValue());
    case Instruction::Or:
      return ConstantInt::get(left->getContext(),
          left->getValue() | right->getValue());
    case Instruction::Xor:
      return ConstantInt::get(left->getContext(),
          left->getValue() ^ right->getValue());
  }
}
```

```
// For constant folding float operators. Simply perform the relevant operation on the operands.
// This will silently ignore errors like overflow/underflow
ConstantFP* evalBinaryFloatOp(unsigned op, ConstantFP* left, ConstantFP* right) {
  APFloat lhs = left->getValueAPF();
  APFloat rhs = right->getValueAPF();
  switch (op) {
    case Instruction::FAdd:
      lhs.add(rhs, rMode);
    case Instruction::FSub:
      lhs.subtract(rhs, rMode);
    case Instruction::FMul:
      lhs.multiply(rhs, rMode);
    case Instruction::FDiv:
      lhs.divide(rhs, rMode);
    case Instruction::FRem:
      lhs.remainder(rhs);
  }
  return ConstantFP::get(left->getContext(), lhs);
}

//Constant Folding
Value* evalBinaryOp(unsigned op, Value* left, Value* right) {
  //Operands passed in are always constants
  switch (op) {
    case Instruction::Add:
    case Instruction::Sub:
    case Instruction::Mul:
    case Instruction::UDiv:
    case Instruction::SDiv:
    case Instruction::URem:
    case Instruction::SRem:
    case Instruction::Shl:
    case Instruction::LShr:
    case Instruction::AShr:
    case Instruction::And:
    case Instruction::Or:
    case Instruction::Xor:
      return evalBinaryIntOp(op,
          cast<ConstantInt>(left),
          cast<ConstantInt>(right));
    case Instruction::FAdd:
    case Instruction::FSub:
    case Instruction::FMul:
    case Instruction::FDiv:
    case Instruction::FRem:
```

```
      return evalBinaryFloatOp(op,
          cast<ConstantFP>(left),
          cast<ConstantFP>(right));
      return left;
  }
}

//Strength reduction. Change multiplication by power of 2 to shift
bool multiplyToShift(BasicBlock::iterator &i, ConstantInt * Op1, Value * Op2) {
  const APInt multiple = Op1->getValue();
  if (multiple.isPowerOf2()) {
    unsigned lg = multiple.logBase2();
    BinaryOperator* newInst = BinaryOperator::Create(
        Instruction::Shl,
        Op2, ConstantInt::get(Op1->getType(), lg, false));
    i->getParent()->getInstList().insertAfter(i, newInst);
    replaceUsesAndDelete(i,newInst);
    ++i; //skip over this newly created instruction
    return true;
  }
  return false;
}



bool applyIdentity(BasicBlock::iterator &i, Value * val) {
  if (val) {
    replaceUsesAndDelete(i,val);
    return true;
  } else return false;
}

bool runOnBasicBlock(BasicBlock &bb, OptInfo & optinf) {
  bool modified = false;

  // Iterate over instructions
  for (BasicBlock::iterator i = bb.begin(), e = bb.end(); i != e; ++i)
  {
    Value* L; Value* R;
    if (i->getNumOperands() == 2) {
      L = i->getOperand(0);
      R = i->getOperand(1);
    }

    // *** Constants for zero, one etc. ***
    // These constants are useful for many of the identies that follow
```

```
APInt zeroAPI; APInt oneAPI; APInt allOnes;
APFloat zeroAPF = APFloat((float)0);
APFloat oneAPF = APFloat((float)1);
if (const IntegerType * ity = dyn_cast<IntegerType>(i->getType())) {
  zeroAPI = APInt(ity->getBitWidth(), 0);
  oneAPI = APInt(ity->getBitWidth(), 1);
  allOnes = ~zeroAPI;
}

unsigned op = i->getOpcode();

// *** Algebraic Identities and Constant Propagation ***
switch (op) {
  default:
    break;
  case Instruction::Store:
    //Constant Propagation
    //If a constant is being stored into a variable, find the place where it is being
    //loaded into a register, and replace all uses of the register with the constant.
    {
      Value * ptr = cast<StoreInst>(i)->getPointerOperand();
      Value * val = cast<StoreInst>(i)->getValueOperand();
      if (isa<Constant>(*val)) {
        for (Value::use_iterator u = ptr->use_begin(); u != ptr->use_end(); ++u) {
          if (LoadInst *l = dyn_cast<LoadInst>(*u)) {
            l->replaceAllUsesWith(val);
            optinf.constProp++;
            modified = true;
          }
        }
      }
      continue; //Cannot apply any other optimization on a store instr
      break;
    }
  case Instruction::Add:
    {
      if (applyIdentity(i, commIdentities<ConstantInt,APInt>(L, R, &zeroAPI, NULL))) {
        // a + 0 = 0 + a = a
        optinf.algebraic++; modified = true; continue;
      }
    }
    break;
  case Instruction::FAdd:
    {
      if (applyIdentity(i, commIdentities<ConstantFP,APFloat>(L, R, &zeroAPF, NULL))) {
        // a + 0 = 0 + a = a
```

```
      optinf.algebraic++; modified = true; continue;
    }
  }
  break;
case Instruction::Sub:
  // Algebraic identities
  {
    ConstantInt * zeroCI = ConstantInt::get(L->getContext(), zeroAPI);
    if (applyIdentity(i, selfInverse<ConstantInt>(L, R, zeroCI))) {
      // a - a = 0
      optinf.algebraic++; modified = true; continue;
    } else if (applyIdentity(i,
                constIdentity<ConstantInt,APInt>(R, L, &zeroAPI, NULL))) {
      // a - 0 = a
      optinf.algebraic++; modified = true; continue;
    }
  }
  break;
case Instruction::FSub:
  {
    ConstantFP * zeroCF = ConstantFP::get(L->getContext(), zeroAPF);
    if (applyIdentity(i, selfInverse<ConstantFP>(L, R, zeroCF))) {
      // a - a = 0
      optinf.algebraic++; modified = true; continue;
    } else if (applyIdentity(i, constIdentity<ConstantFP,APFloat>(R, L, &zeroAPF, NULL)))
      // a - 0 = a
      optinf.algebraic++; modified = true; continue;
    }
  }
  break;
case Instruction::Mul:
  {
    if (applyIdentity(i, commIdentities<ConstantInt,APInt>(L, R, &oneAPI, &zeroAPI))) {
      // a * 1 = 1 * a = a
      // a * 0 = 0 * a = 0
      optinf.algebraic++; modified = true; continue;
    }
  }
  break;
case Instruction::FMul:
  {
    if (applyIdentity(i, commIdentities<ConstantFP,APFloat>(L, R, &oneAPF, &zeroAPF))) {
      // a * 1 = 1 * a = a
      // a * 0 = 0 * a = 0
      optinf.algebraic++; modified = true; continue;
    }
```

```
  }
  break;
case Instruction::UDiv:
case Instruction::SDiv:
  {
    ConstantInt * oneCI = ConstantInt::get(L->getContext(), oneAPI);
    if (applyIdentity(i, selfInverse(L, R, oneCI))) {
      // a / a = 1
      optinf.algebraic++; modified = true; continue;
    } else if (applyIdentity(i, constIdentity<ConstantInt,APInt>(L, R, NULL, &zeroAPI))) {
      // 0/a = 0
      optinf.algebraic++; modified = true; continue;
    } else if (applyIdentity(i, constIdentity<ConstantInt,APInt>(R, L, &oneAPI, NULL))) {
      // a/1 = 1
      optinf.algebraic++; modified = true; continue;
    }
    break;
  }
case Instruction::FDiv:
  {
    ConstantFP * oneCF = ConstantFP::get(L->getContext(), oneAPF);
    if (applyIdentity(i, selfInverse(L, R, oneCF))) {
      // a / a = 1
      optinf.algebraic++; modified = true; continue;
    } else if (applyIdentity(i, constIdentity<ConstantFP,APFloat>(L, R, NULL, &zeroAPF))) {
      // 0/a = 0
      optinf.algebraic++; modified = true; continue;
    } else if (applyIdentity(i, constIdentity<ConstantFP,APFloat>(R, L, &oneAPF, NULL))) {
      // a/1 = 1
      optinf.algebraic++; modified = true; continue;
    }
    break;
  }
case Instruction::URem:
case Instruction::SRem:
  {
    ConstantInt * zeroCI = ConstantInt::get(L->getContext(), zeroAPI);
    if (applyIdentity(i, selfInverse(L, R, zeroCI))) {
      // a mod a = 0
      optinf.algebraic++; modified = true; continue;
    } else if (applyIdentity(i, constIdentity<ConstantInt,APInt>(L, R,
                                                                 &oneAPI, &zeroAPI))) {
      // 0 mod a = 0
      // 1 mod a = 1
      optinf.algebraic++; modified = true; continue;
    } else if (applyIdentity(i, remIdentity<ConstantInt,APInt>(L, R,
```

```
                                                                    &oneAPI, &zeroAPI))) {
      // a mod 1 = 0
      optinf.algebraic++; modified = true; continue;
    }
    break;
  }
case Instruction::FRem:
  {
    ConstantFP * zeroCF = ConstantFP::get(L->getContext(), zeroAPF);
    if (applyIdentity(i, selfInverse(L, R, zeroCF))) {
      // a mod a = 0
      optinf.algebraic++; modified = true; continue;
    } else if (applyIdentity(i, constIdentity<ConstantFP,APFloat>(L, R,
                                                                  &oneAPF, &zeroAPF))) {
      // 0 mod a = 0
      // 1 mod a = 1
      optinf.algebraic++; modified = true; continue;
    } else if (applyIdentity(i, remIdentity<ConstantFP,APFloat>(L, R,
                                                                &oneAPF, &zeroAPF))) {
      // a mod 1 = 0
      optinf.algebraic++; modified = true; continue;
    }
    break;
  }
case Instruction::Shl:
case Instruction::LShr:
case Instruction::AShr:
  break;
case Instruction::And:
  {
    if (applyIdentity(i, commIdentities<ConstantInt,APInt>(L, R, &allOnes, &zeroAPI))) {
      // a && T = T && a = a
      // a && 0 = 0 && a = 0
      optinf.algebraic++; modified = true; continue;
    }
    break;
  }
case Instruction::Or:
  {
    if (applyIdentity(i, commIdentities<ConstantInt,APInt>(L, R, &zeroAPI, &allOnes))) {
      // a || 0 = 0 || a = a
      // a || T = T || a = T
      optinf.algebraic++; modified = true; continue;
    }
    break;
  }
```

```
      case Instruction::Xor:
        {
          if (applyIdentity(i, commIdentities<ConstantInt,APInt>(L, R, &zeroAPI, NULL))) {
            // a xor 0 = 0 xor a = a
            optinf.algebraic++; modified = true; continue;
          }
          break;
        }
    }


    // *** Constant Folding ***
    if (i->getNumOperands() == 2 && isa<Constant>(L) && isa<Constant>(R)) {
      Value * result = evalBinaryOp(op, L, R);
      replaceUsesAndDelete(i,result);
      optinf.constFold++; modified = true; continue;
    }


    // *** Strength reduction ***
    switch (op) {
      default:
        break;
      case Instruction::Mul:
        // Change multiplication by power of 2 to left shift
        if (ConstantInt* LC = dyn_cast<ConstantInt>(L)) {
          if (multiplyToShift(i,LC,R)) {
            optinf.strengthRed++; modified = true; continue;
          }
        } else if (ConstantInt* RC = dyn_cast<ConstantInt>(R)) {
          if (multiplyToShift(i,RC,L)) {
            optinf.strengthRed++; modified = true; continue;
          }
        }
    }
  }

  return modified;
}

virtual bool runOnFunction(Function &f) {
  bool modified = false;
  OptInfo optinf;
  optinf.constFold = 0;
  optinf.constProp = 0;
  optinf.algebraic = 0;
```

```
      optinf.strengthRed = 0;
      for (Function::iterator bb = f.begin(); bb != f.end(); bb++) {
        modified = runOnBasicBlock(*bb, optinf);
      }
      errs() << "Optimizations performed:\n";
      errs() << "Constant Propagation: " << optinf.constProp << "\n";
      errs() << "Constant Folding: " << optinf.constFold << "\n";
      errs() << "Algebraic Idenities: " << optinf.algebraic << "\n";
      errs() << "Strength Reduction: " << optinf.strengthRed << "\n";
      return modified;
    }
  };

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