```
// 15-745 S14 Assignment 1: LocalOpts.cpp
// Group:
// aebtekar  Aram Ebtekar
// auc       Alejandro Carbonara
//////////////////////////////////////////////////////////////////////////////

#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Constants.h"
#include "llvm/IR/InstrTypes.h"

#include <ostream>
#include <fstream>
#include <iostream>
#include <string>

using namespace llvm;

namespace
{

class LocalOpts : public ModulePass
{
  int optCount_algebraic;
  int optCount_constfold;
  int optCount_strength;

  int log2(long long v)
  {
    // only continue if v is a power of 2
    if (v <= 0 || (v & (v-1)))
      return -1;
    // compute its log base 2
    int ret = 0;
    while (v > 1)
    {
      v >>= 1;
      ++ret;
    }
    return ret;
  }

  // Optimize the function.
  void optimizeFunction(Function& F)
  {
    // Instructions to erase
    std::vector<Instruction*> garbage;

    for (ilist_iterator<BasicBlock> BI = F.begin(); BI != F.end(); ++BI)
    for (ilist_iterator<Instruction> II = BI->begin(); II != BI->end(); ++II)
    {
      //std::cout << II->getOpcodeName() << "(" << II->getNumOperands() << ")" << s
td::endl;
      if (II->getNumOperands() == 2)
      {
        Value* op0 = II->getOperand(0);
        Value* op1 = II->getOperand(1);
        ConstantInt* const0 = dyn_cast<ConstantInt>(op0);
        ConstantInt* const1 = dyn_cast<ConstantInt>(op1);
        long long val0, val1;
        if (const0 != NULL)
          val0 = const0->getSExtValue();
        if (const1 != NULL)
          val1 = const1->getSExtValue();

        if (const0 != NULL && const1 != NULL)
        {
          // constant folding
          switch (II->getOpcode())
          {
            case Instruction::Add:
              II->replaceAllUsesWith(ConstantInt::getSigned(II->getType(), val0+val
1));
              break;
            case Instruction::Sub:
              II->replaceAllUsesWith(ConstantInt::getSigned(II->getType(), val0-val
1));
              break;
            case Instruction::Mul:
              II->replaceAllUsesWith(ConstantInt::getSigned(II->getType(), val0*val
1));
              break;
            case Instruction::SDiv:
              II->replaceAllUsesWith(ConstantInt::getSigned(II->getType(), val0/val
1));
              break;
          }
          ++optCount_constfold;
          garbage.push_back(II);
          continue;
        }
        // algebraic and strength reductions
        switch (II->getOpcode())
        {
          case Instruction::Add:
            if (const0 != NULL && val0 == 0) // 0 + x -> x
            {
              II->replaceAllUsesWith(op1);
              ++optCount_algebraic;
              garbage.push_back(II);
            }
            else if (const1 != NULL && val1 == 0) // x + 0 -> x
            {
              II->replaceAllUsesWith(op0);
              ++optCount_algebraic;
              garbage.push_back(II);
            }
            break;
          case Instruction::Sub:
            if (const1 != NULL && val1 == 0) // x - 0 -> x
            {
              II->replaceAllUsesWith(op0);
              ++optCount_algebraic;
              garbage.push_back(II);
            }
            else if (op0 == op1) // x - x -> 0
            {
              II->replaceAllUsesWith(ConstantInt::getSigned(II->getType(), 0));
              ++optCount_algebraic;
              garbage.push_back(II);
            }
            break;
          case Instruction::Mul:
            if (const0 != NULL && val0 == 1) // 1 * x -> x
            {
              II->replaceAllUsesWith(op1);
              ++optCount_algebraic;
              garbage.push_back(II);
            }
            else if (const1 != NULL && val1 == 1) // x * 1 -> x
            {
              II->replaceAllUsesWith(op0);
              ++optCount_algebraic;
              garbage.push_back(II);
            }
            else if (const0 != NULL && log2(val0) != -1) // 2^n * x -> x << n
```

```
          {
            Instruction* shift = BinaryOperator::Create(Instruction::Shl, op1,
              ConstantInt::getSigned(II->getType(), log2(val0)),
              "shl", II);
            II->replaceAllUsesWith(shift);
            garbage.push_back(II);
          }
          else if (const1 != NULL && log2(val1) != -1) // x * 2^n -> x << n
          {
            Instruction* shift = BinaryOperator::Create(Instruction::Shl, op0,
              ConstantInt::getSigned(II->getType(), log2(val1)),
              "shl", II);
            II->replaceAllUsesWith(shift);
            ++optCount_strength;
            garbage.push_back(II);
          }
          break;
        case Instruction::SDiv:
          if (const1 != NULL && val1 == 1) // x / 1 -> x
          {
            II->replaceAllUsesWith(op0);
            ++optCount_algebraic;
            garbage.push_back(II);
          }
          else if (op0 == op1) // x / x -> 1
          {
            II->replaceAllUsesWith(ConstantInt::getSigned(II->getType(), 1));
            ++optCount_algebraic;
            garbage.push_back(II);
          }
          else if (const1 != NULL && log2(val1) != -1) // x / 2^n -> x >> n
          {
            Instruction* shift = BinaryOperator::Create(Instruction::AShr, op0,
              ConstantInt::getSigned(II->getType(), log2(val1)),
              "ashr", II);
            II->replaceAllUsesWith(shift);
            ++optCount_strength;
            garbage.push_back(II);
          }
          break;
      }
    }
  }
  // Get rid of unwanted instructions
  for (std::vector<Instruction*>::iterator II = garbage.begin(); II != garbage.en
d(); ++II)
    (*II)->eraseFromParent();
 }

public:

 static char ID;

 LocalOpts() : ModulePass(ID) { }

 ~LocalOpts() { }

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

 virtual bool runOnFunction(Function& F)
 {
  optimizeFunction(F);
 }

 virtual bool runOnModule(Module& M)
 {
  optCount_algebraic = 0;
  optCount_constfold = 0;
  optCount_strength = 0;
  for (Module::iterator MI = M.begin(), ME = M.end(); MI != ME; ++MI)
  {
   runOnFunction(*MI);
  }
  // Print optimization summary.
  std::cout << "Transformations applied:" << std::endl;
  std::cout << "\tAlgebraic identities:\t" << optCount_algebraic << std::endl;
  std::cout << "\tConstant folding:\t" << optCount_constfold << std::endl;
  std::cout << "\tStrength reduction:\t" << optCount_strength << std::endl;
  return optCount_algebraic + optCount_constfold + optCount_strength > 0;
 }
};

// LLVM uses the address of this static member to identify the pass, so the
// initialization value is unimportant.
char LocalOpts::ID = 0;
RegisterPass<LocalOpts> X("local-opts", "15745: Local Optimization");

}
```