15-745: Optimizing Compilers
Assignment 1 Writeup
Ben Humberston, bhumbers
Prashanth Suresh, psuresh

Implementation Summary

FunctionInfo:
All implementation is found in FunctionInfo::printFunctionInfo(). We iterate through all functions in the module and calculate statistics for each independently. This includes an inner iteration for call counts where we examine all instructions in the module to test whether it matches the current function under consideration; this could be optimized by collecting the call counts for all functions at once in a single pass through the module, but we consider the unoptimized form workable.

LocalOpts:
All implementation is found in LocalOpts::applyLocalOptimizations(). Until we find that no more optimizations are possible we repeatedly iterate through all instructions in the module and apply algebraic identity simplifications, constant folding, and power reduction optimizations. We handle cases where only one operand is a constant in either the first or second operand position using condition-dependent pointers to the constant and non-constant terms

Test Listing
(Note: Only listing tests additional tests implemented by our group)

- ./FunctionInfo/loop.c
  - Fact() & call(): Tests behavior on a slightly more complex, recursive function
- ./FunctionInfo/tests/var_args_test.c
  - var_args_func(): Verifies correct identification of variable argument function
- ./LocalOpts/test-inputs/algebraic.c
  - add_zero(), mul_one(), div_one(): Verify simple algebraic identity behavior
  - alg_identity_combo(): Verifies optimization behavior for slightly more complex algebra
  - alg_const_folds(): Verifies optimization behavior for multi-pass constant folding
  - undef_var_test(): Verifies that crash doesn't occur when one term is undefined

Test Output

See README.txt for shell commands to run all tests

FunctionInfo Expected Test Outputs

```
15745 Function Information Pass
Module loop.bc
Name, Args, Calls,     Blocks,    Insns
g_incr,    1,    0,    1,    4
loop, 3,    0,    3,    10
Fact, 1,    0,    6,    41
var_ar,    *,    0,    1,    2
call, 0,    0,    1,    1
```

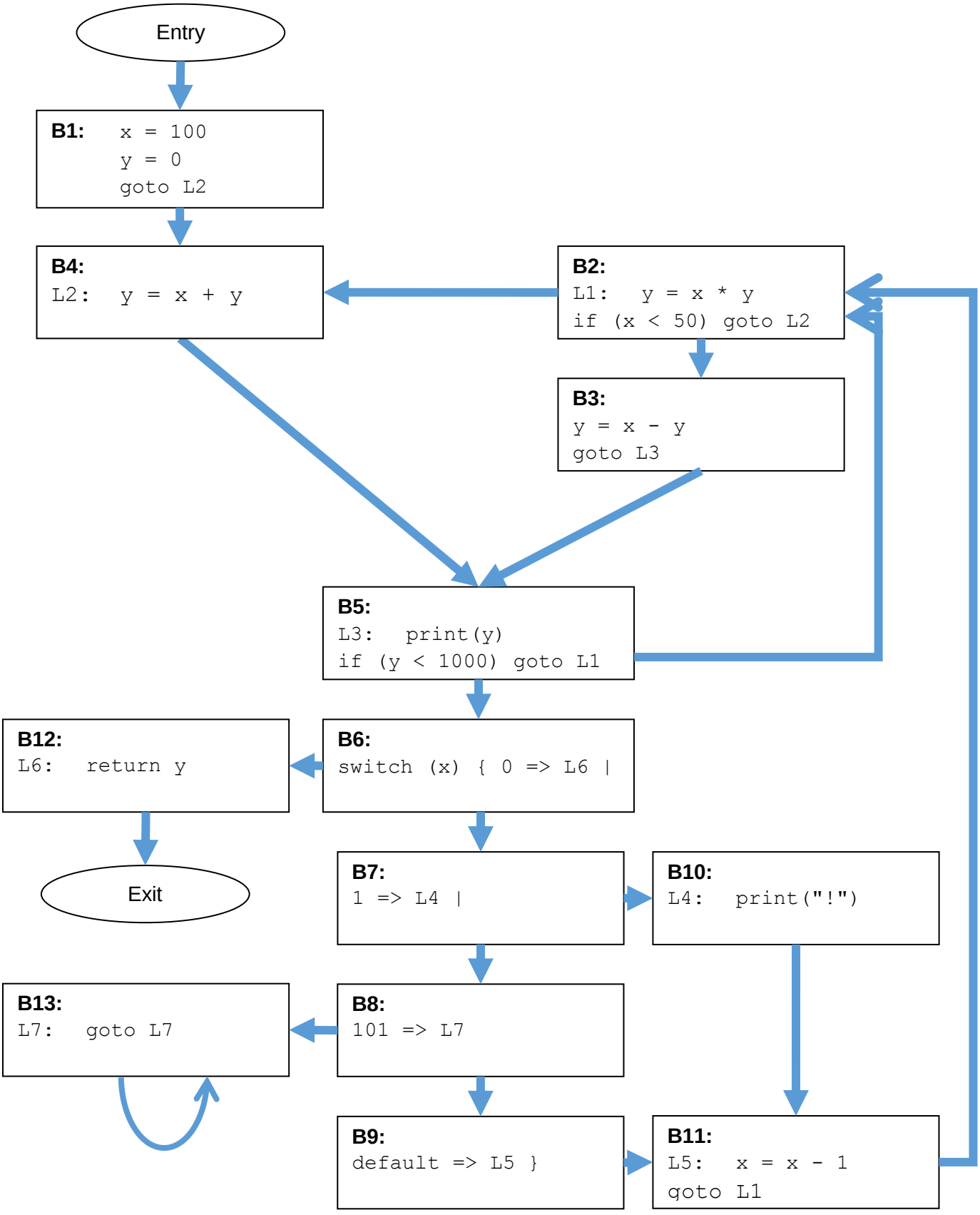
LocalOpts Expected Test Outputs

```
Module loop-m2r.bc
Optimized an instance of additive identity.
Transformations Applied:
    Algebraic Identities: 1
    Constant Folding:     0
    Strength Reduction:   0

Module algebraic-m2r.bc
Optimized an instance of additive identity.
Optimized an instance of multiplicative identity.
Optimized an instance of division identity.
Optimized an instance of division identity.
Optimized an instance of multiplicative identity.
Const-folded an expression: mul 6, 7
Const-folded an expression: add1 6, 42
Optimized an instance of multiplicative identity.
Optimized an instance of additive identity.
Const-folded an expression: sub 42, 1
Const-folded an expression: add 41, 3
Transformations Applied:
    Algebraic Identities: 7
    Constant Folding:     4
    Strength Reduction:   0


Module constfold-m2r.bc
Const-folded an expression: add 4, 2
Const-folded an expression: add2 0, 2
Const-folded an expression: add1 6, 3
Const-folded an expression: add3 2, 3
Const-folded an expression: mul 5, 9
Transformations Applied:
    Algebraic Identities: 0
    Constant Folding:     5
    Strength Reduction:   0

Module strength-m2r.bc
About to apply a bitshift: 2, 1
Optimized an instance of additive identity.
About to apply a bitshift: 8, 3
Transformations Applied:
    Algebraic Identities: 1
    Constant Folding:     0
    Strength Reduction:   2
```

5.1 CFG Basics

Basic blocks:

```
B1:          x = 100
             y = 0
             goto L2

B2:    L1:   y = x * y
             if (x < 50) goto L2

B3:          y = x - y
             goto L3

B4:    L2:   y = x + y

B5:    L3:   print(y)
             if (y < 1000) goto L1

B6:          switch (x) { 0 => L6 |
B7:               1 => L4 |
B8:               101 => L7
B9:               default => L5 }

B10:   L4:   print("!")

B11:   L5:   x = x - 1
             goto L1

B12:   L6:   return y
B13:   L7:   goto L7
```


Entry
B1:
x = 100
y = 0
goto L2
B4:
L2: y = x + y
B2:
L1: y = x * y
if (x < 50) goto L2
B3:
y = x - y
goto L3
B5:
L3: print(y)
if (y < 1000) goto L1
B12:
L6: return y
B6:
switch (x) { 0 => L6 |
Exit
B7:
1 => L4 |
B10:
L4: print("!")
B13:
L7: goto L7
B8:
101 => L7
B9:
default => L5 }
B11:
L5: x = x - 1
goto L1

5.2 Available Expressions

BB	EVAL	KILL
1	b+c, b*b, b*d	
2	b+c, i+1	
3	b*b	
4	b*b	b*d
5	i+1	

BB	IN	OUT
1	{}	b+c, b*b, b*d
2	b+c, b*b, b*d	b+c, b*b, b*d, i+1
3	b+c, b*b, b*d, i+1	b+c, b*b, b*d, i+1
4	b+c, b*b, b*d, i+1	b+c, b*b, i+1
5	b+c, b*b, i+1	b+c, b*b, i+1

5.3 Faint Analysis

Note: The following data flow pass is almost exactly the same as the liveness pass from Lecture 4, with the exception that the "Use" set in the transfer function does not contain locally exposed uses where the variable being used is itself on the LHS. Additionally, the way we mark faint/dead vs. live variables at each code point (part 9) is modified to ensure that faint variables are correctly marked within a block.

We considered an alternative algorithm which had an additional constraint on insertion into the "Use" set stating that the variable on the LHS of each use must be in the live set being provided in the block's out edge (backward analysis). If $in(exit)$ were a non-empty set (eg, if it consisted of the variables to be returned from a function), we believe this algorithm would correctly mark as faint all variables which are not used to calculate the members of that set. However, because $in(exit) = \emptyset$, adding this constraint would have the effect of marking *all* variables in all blocks as faint (since all variables are dead/faint at exit), so all assignments would be to faint variables and may be eliminated!

Thus, we do not include this constraint and allow faint variables within a block to be considered "live" in predecessor blocks, so that the faintness condition only applies within blocks, not between them. This seems to match the intent of the two cases provided in the problem description.

1. Set of elements: Set of live *variables*

2. Direction: Backward analysis

3. Transfer Function:

$in_b = f_b(out) = LocallyExposedUsesExceptSelf_B \cup (out - Def_B)$

Here, Def_B is the set of variables defined in basic block B and $LocallyExposedUsesExceptSelf_B$ is the set of locally exposed uses in B, except where the usage has the variable itself in the LHS of the assignment (this prevents adding a variable to the live set when its only future LHS "user" is itself).

4. Meet Operator: Union ∪. $out(B) = \cup\, in(succ(B))$

(cont'd on next page)

5. Boundary Condition: $in(exit) = \emptyset$

6. Initial interior points: $in(B) = \emptyset$ for all interior basic blocks B

7. Block Ordering:
It is expected that blocks will be visited from end back to $entry$ in a reverse breadth-first manner. Since liveness of a variable depends on the future blocks in a program, this back-to-front ordering ensures that the live set is complete for each block when it is considered.

8. Convergence:
This pass is guaranteed to converge. We repeatedly iterate over all blocks, applying the transfer function, until no changes occur to $in(B)$ for any block B. The maximum possible live set at each $in(B)$ will be the universe of variables for the CFG. Since the transfer function which updates $in(B)$ only removes elements if they are defined in a block, $in(B)$ will grow monotonically on each repetition of the pass. Thus, convergence occurs when all $in(B)$ sets reach their maximum (finite) size.

9. Faintness Algorithm:
For each variable in block B, if the variable is neither a member of $out(B)$ nor part of the RHS of an assignment to a variable that is in $out(B)$, then that variable/assignment is faint in that block. That is, if the variable is not part of nor does it affect some variable in the future live set specified at the out edge of B, then it is considered faint, as the assignment will have no impact on future code.

```
For each basic block B {
      For each variable V assigned in B {
            If (V ∉ out(B) and V′ ∉ out(B) for all variable assignments V′ = … which use V) {
                  Mark V as faint
            }
      }
}
```

```
// 15-745 S14 Assignment 1: FunctionInfo.cpp
// Group: bovik, bovik2
//////////////////////////////////////////////////////////////////////////////

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

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

using namespace llvm;

namespace {

class FunctionInfo : public ModulePass {

  // Output the function information to standard out.
  void printFunctionInfo(Module& M) {
    std::cout << "Module " << M.getModuleIdentifier().c_str() << std::endl;
    std::cout << "Name,\tArgs,\tCalls,\tBlocks,\tInsns\n";

    //Iterate over all functions and collect info for each
    for (Module::iterator modIter = M.begin(); modIter != M.end(); ++modIter) {
      Function* targetFunc = modIter;

      std::string func_name = targetFunc->getName();
      bool is_var_arg = targetFunc->isVarArg();
      size_t arg_count = targetFunc->arg_size();
      size_t callsite_count = 0;
      size_t block_count = targetFunc->getBasicBlockList().size();
      size_t instruction_count = 0;

      //Count all intructions in basic blocks for this func
      for (Function::iterator funIter = modIter->begin(); funIter != modIter->end(); ++funIter)
        instruction_count += funIter->getInstList().size();

      //Iterate through complete module and count number of calls to this function
      int count = 0;
      for (Module::iterator modIter = M.begin(); modIter != M.end(); ++modIter) {
        for (Function::iterator funIter = modIter->begin(); funIter != modIter->end(); ++funIter) {
          for (BasicBlock::iterator bbIter = funIter->begin(); bbIter != funIter->end(); ++bbIter) {
            if (CallInst* callInst = dyn_cast<CallInst>(&*bbIter)) {
              if (callInst->getCalledFunction() == targetFunc) {
                ++callsite_count;
              }
            }
          }
        }
      }

      std::cout << func_name << ",\t";
      if (is_var_arg) {
        std::cout << "*,\t";
      } else {
        std::cout << arg_count << ",\t";
      }
      std::cout << callsite_count << ",\t" << block_count << ",\t" << instruction_count << std::endl;
    }
  }

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) {
    // TODO: implement this.
    return false;
  }

  virtual bool runOnModule(Module& M) {
    std::cerr << "15745 Function Information Pass\n"; // TODO: remove this.
    for (Module::iterator MI = M.begin(), ME = M.end(); MI != ME; ++MI) {
      runOnFunction(*MI);
    }
    printFunctionInfo(M);
    return false;
  }

};

// LLVM uses the address of this static member to identify the pass, so the
// initialization value is unimportant.
char FunctionInfo::ID = 0;
RegisterPass<FunctionInfo> X("function-info", "15745: Function Information");

}
```

```
// 15-745 S14 Assignment 1: LocalOpts.cpp
// Group: bhumbers, psuresh
////////////////////////////////////////////////////////////////////////////////

#include "llvm/Pass.h"
#include "llvm/IR/Constants.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/Instructions.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/IR/Module.h"
#include "llvm/Transforms/Utils/BasicBlockUtils.h"

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

using namespace llvm;

namespace {

class FunctionInfo : public ModulePass {

  void applyLocalOptimizations(Module& M) {
    std::cout << "Module " << M.getModuleIdentifier().c_str() << std::endl;

    int numAlgIdentityOpts = 0;
    int numConstFoldOpts = 0;
    int numStrengthRedOpts = 0;

    bool anyOptsApplied = true; //used to track when all possible optimizations are applied

    //Keep applying optimizations until no longer possible to do so
    while (anyOptsApplied) {
      int prevNumAlgIdentityOpts = numAlgIdentityOpts;
      int prevNumConstFoldOpts = numConstFoldOpts;
      int prevNumStrengthRedOpts = numStrengthRedOpts;

      //Iterate through complete module and optimize
      //NOTE: Most of these optimizations are modeled on the built-in LLVM functionality in ConstantFold.cpp
      int count = 0;
      for (Module::iterator modIter = M.begin(); modIter != M.end(); ++modIter) {
        for (Function::iterator funIter = modIter->begin(); funIter != modIter->end(); ++funIter) {
          for (BasicBlock::iterator bbIter = funIter->begin(); bbIter != funIter->end(); ++bbIter) {
            //If looking at a binary operator, apply one of the opts for this question
            if (BinaryOperator* binInst = dyn_cast<BinaryOperator>(&*bbIter)) {
              //Extract constant integer operands if able
              ConstantInt* constIntA = dyn_cast<ConstantInt>(binInst->getOperand(0));
              ConstantInt* constIntB = dyn_cast<ConstantInt>(binInst->getOperand(1));

              //If both operands are const ints, we can do constant folding and replace with the expression w/ eval r
esult
              if (constIntA && constIntB) {
                const APInt& constIntValA = constIntA->getValue();
                const APInt& constIntValB = constIntB->getValue();

                ConstantInt* evalConst = 0;
                switch (binInst->getOpcode()) {
                  case Instruction::Add: evalConst = ConstantInt::get(constIntA->getContext(), constIntValA + constIn
tValB); break;
                  case Instruction::Sub: evalConst = ConstantInt::get(constIntA->getContext(), constIntValA - constIn
tValB); break;
                  case Instruction::Mul: evalConst = ConstantInt::get(constIntA->getContext(), constIntValA * constIn
tValB); break;
                }
                if (evalConst) {
                  std::cout << "Const-folded an expression: " << binInst->getName().str() << " " << constIntValA.toSt
ring(10, true) << ", " + constIntValB.toString(10, true) << std::endl;
                  ReplaceInstWithValue(binInst->getParent()->getInstList(), bbIter, evalConst);
                  numConstFoldOpts++;
                }
              }
              //Otherwise, if at least one is a constant integer, see what else we can optimize...
              else if (constIntA || constIntB) {
                ConstantInt* constIntTerm = (constIntA) ? constIntA : constIntB; //grab the const term for this binar
y op
                Value* otherTerm = (constIntA) ? binInst->getOperand(1) : binInst->getOperand(0); //grab the other "v
alue" term
```

```
                //Apply any algebraic identity opts
                switch (binInst->getOpcode()) {
                  case Instruction::Add:
                    // Additive identity: x + 0 = x
                    if (constIntTerm->isZero()) {
                      ReplaceInstWithValue(binInst->getParent()->getInstList(), bbIter, otherTerm);
                      std::cout << "Optimized an instance of additive identity." << std::endl;
                      numAlgIdentityOpts++;
                    }
                    break;
                  case Instruction::Mul:
                    // Multiplicative identity: x * 1 = 1
                    if (constIntTerm->isOne()) {
                      ReplaceInstWithValue(binInst->getParent()->getInstList(), bbIter, otherTerm);
                      std::cout << "Optimized an instance of multiplicative identity." << std::endl;
                      numAlgIdentityOpts++;
                    }
                    else if (constIntTerm->getValue().isPowerOf2()) {
                      const int64_t constIntVal = constIntTerm->getSExtValue();
                      Value* shiftVal = ConstantInt::get(constIntTerm->getType(), log2(constIntVal));

                      std::cout << "About to apply a bitshift: " << constIntVal << ", " << log2(constIntVal) << std::
endl;

                      ReplaceInstWithInst(binInst->getParent()->getInstList(),bbIter, BinaryOperator::Create(Instruct
ion::Shl, otherTerm, shiftVal, "shl", (Instruction*)(0)));
                      numStrengthRedOpts++;
                    }
                    break;
                  case Instruction::UDiv:
                  case Instruction::SDiv:
                    //Division identity: x / 1 = x
                    //(Note: not commutative... left term must be variable & right term to be 1)
                    if (constIntB && constIntB->isOne()) {
                      ReplaceInstWithValue(binInst->getParent()->getInstList(), bbIter, binInst->getOperand(0));
                      std::cout << "Optimized an instance of division identity." << std::endl;
                      numAlgIdentityOpts++;
                    }
                    break;
                }
              }
            }
          }
        }
      }

      //(Crudely) check whether any optimizations occurred so that we know whether to keep iterating
      anyOptsApplied =  (prevNumAlgIdentityOpts != numAlgIdentityOpts) ||
                        (prevNumConstFoldOpts != numConstFoldOpts) ||
                        (prevNumStrengthRedOpts != numStrengthRedOpts);
    }

    std::cout << "Transformations Applied:" << std::endl;
    std::cout << "    Algebraic Identities: " << numAlgIdentityOpts << std::endl;
    std::cout << "    Constant Folding:     " << numConstFoldOpts << std::endl;
    std::cout << "    Strength Reduction:   " << numStrengthRedOpts << std::endl;
  }

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) {
    // TODO: implement this.
    return false;
  }
```

```
  virtual bool runOnModule(Module& M) {
    std::cerr << "15-745 Local Optimizations Pass\n"; // TODO: remove this.
    for (Module::iterator MI = M.begin(), ME = M.end(); MI != ME; ++MI) {
      runOnFunction(*MI);
    }
    applyLocalOptimizations(M);
    return false;
  }

private:


};

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

}
```