Homework Assignment 1 Solutions

CMU 15-745: Optimizing Compilers (Spring 2015)

Joy Arulraj (jarulraj), Nisarg Shah (nkshah)

1 FunctionInfo

1.1 Implementation

We iterate over all functions in the module and for each function, we use helper functions in the LLVM `Function` class, such as `getName()`, to obtain the required info. We use the `size()` function in the `Instruction` class to compute the number of instructions in all the basic blocks in the function.

1.2 Source Code Listing

The listing starts from the next page.

```
// 15-745 S15 Assignment 1: FunctionInfo.cpp
// Group: jarulraj, nkshah
////////////////////////////////////////////////////////////////////////////////

#include "llvm/Pass.h"
#include "llvm/IR/Function.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) {
			outs() << "Module " << M.getModuleIdentifier().c_str() << "\n";
			outs() << "Name,\tArgs,\tCalls,\tBlocks,\tInsns\n";

			// Print info about each function
			for (Module::iterator MI = M.begin(), ME = M.end(); MI != ME; ++MI) {
				runOnFunction(*MI);
			}
		}

	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) {
			bool is_var_arg = false;
			size_t arg_count = 0;
			size_t callsite_count = 0;
			size_t block_count = 0;
			size_t instruction_count = 0;

			// Get all the required information
			std::string function_name = F.getName(); // Get name
			is_var_arg = F.isVarArg(); // Check if # arguments is variable
			if (!is_var_arg) {
				arg_count = F.arg_size(); // # fixed args
			}

			callsite_count = F.getNumUses(); // # direct call sites
			block_count = F.size(); // # basic blocks

			// # instructions
			for (Function::iterator FI = F.begin(), FE = F.end(); FI != FE; ++FI) {
				instruction_count += FI->size();
			}

			// Print Information
			outs() << function_name  << ",\t";
```

```
                        if (is_var_arg) {
                                outs() << "*,\t";
                        } else {
                                outs() << arg_count << ",\t";
                        }
                        outs() << callsite_count << ",\t" << block_count << ",\t"
                               << instruction_count << "\n";

                        return false;
                }

                virtual bool runOnModule(Module& M) {
                        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");

}
```

1.3 Test Cases

The test cases start from the next page.

```c
int loop (int a, int b, int c);

// TEST 1 : Only 1 block
void z()
{
}

// TEST 2 : Recursion
int y(int a)
{
    int s = loop(1, 100, 3);

    if(s > 0)
        y(a);

    return s;
}


// TEST 3 : Var Arg
#include <stdarg.h>

int FindMax (int n, ...)
{
    int i,val,largest;
    va_list vl;
    va_start(vl,n);
    largest=va_arg(vl,int);
    for (i=1;i<n;i++)
    {
        val=va_arg(vl,int);
        largest=(largest>val)?largest:val;
    }
    va_end(vl);
    return largest;
}
```

1.4 Expected Results

```
Module test-inputs/others.bc
Name,   Args,   Calls,  Blocks, Insns
z,      0,      0,      1,      1
y,      1,      1,      3,      6
loop,   3,      1,      0,      0
FindMax,        *,      0,      8,      29
llvm.va_start,  1,      1,      0,      0
llvm.va_end,    1,      1,      0,      0
```

2 LocalOpts

2.1 Implementation

We wrote a per-basic block pass that performs the required transformations. For each block, we keep applying the 3 transformations within a **loop**, until the block is not modified after an entire iteration. This is to ensure that the benefits of each optimization are relayed onto other optimizations and we are not constrained by the order in which we apply the optimizations within a single iteration. The transformations are described below:

- **Algebraic Identities :** We iterate over all instructions and look for binary operations involving scalar integer type operands with one of them being a constant. Depending on the type of operation, we apply identities that involve either the operator's **identity element** or **inverse element**. For instance, for the addition operator `Instruction::Add`, we use the identities $x + 0 = x$ and $0 + x = x$. For the subtraction operator `Instruction::Sub`, we apply the identities $x - 0 = x$ and $x - x = 0$. We do similar optimizations for other operators like `Instruction::Mul`, `Instruction::UDiv`, and `Instruction::And`. In all cases, we use `ReplaceInstWithValue` to perform the optimization.

- **Constant Folding :** We iterate over all instructions and look for binary operations involving scalar constant integer type operands. Depending on the type of operation, we evaluate the expression and replace all uses of that instruction with the constant using `ReplaceInstWithValue`. We do not handle floating point operations exhaustively. All common instructions like `Instruction::Add`, `Instruction::Sub`, `Instruction::Mul`, and `Instruction::Div` are handled.

- **Strength Reduction :** We iterate over all instructions and look for binary operations involving scalar integer type operands with one of them being a constant. Depending on the type of operation, we perform strength reduction. For instance, when the RHS involves a `Instruction::Mul` with a number that is a power of 2, we replace that instruction with another instruction that uses `Instruction::Shl`. This is done using `ReplaceInstWithInst`. We also handle division by a power of 2. We extended the optimization to handle multiplication by numbers of the form $2^k + 1$ and $2^k - 1$. For instance, multiplication with 3 is replaced by left shift and addition operators.

2.2 Source Code Listing

The listing starts from the next page.

```
// 15-745 S15 Assignment 1: LocalOpts.cpp
// Group: jarulraj, nkshah

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

using namespace std;
using namespace llvm;

// Get constant from value
#define get_const(t,val) ConstantInt::get((IntegerType*)t,val)


// DEBUG mode: Prints all the optimizations performed to stdout, and the original and final code so we can compare!
//#define DEBUG 1

#ifdef DEBUG
#define DBG(a) a
#else
#define DBG(a)
#endif

namespace
{
        // Struct storing statistics about the number of different optimizations performed
        struct LocalOptsInfoStruct {
                LocalOptsInfoStruct() : numAlgebraicOpts(0), numConstantFolds(0), numStrengthReds(0) {}
                int numAlgebraicOpts;
                int numConstantFolds;
                int numStrengthReds;
        } LocalOptsInfo;

        // If n is a power of 2, then return that power, else return -1
        int64_t find_log (int64_t n)
        {
                if (n <= 0)
                        return -1;
                int64_t res = 0;
                while (((n & 1) == 0) && n > 1) { // While n is even and more than 1
                        n >>= 1;
                        ++res;
                }

                if (n == 1)
                        return res;
                else
                        return -1;
        }

        // If n is (a power of 2) +- 1, then set that power in the exponent argument and return n-that_power_of_2.
        // Else, return -2.
        int64_t find_log_improved (int64_t n, int64_t &exponent)
        {
                if (n <= 0)
                        return -1;
                int64_t res = -2;
                if ((exponent = find_log(n)) >= 0) {
                        return 0;
                }
                else if ((exponent = find_log(n-1)) >= 0) {
                        return 1;
```

```
		}
		else if ((exponent = find_log(n+1)) >= 0) {
			return -1;
		}
		else {
			return -2;
		}
	}

	class LocalOpts : public ModulePass
	{

		// Algebraic Optimizations
		bool algebraic_optimizations(BasicBlock& B)
		{
			bool modified = false;

			// Iterate over all instructions
			for(BasicBlock::iterator BI = B.begin(); BI != B.end(); )
			{
				Instruction& inst(*BI);
				bool inst_removed = false;   // Is inst removed ?

				// In binary operations
				if(BinaryOperator *bop = dyn_cast<BinaryOperator>(&inst))
				{
					Value *val1(bop->getOperand(0));  // Get the first and the second operand (as values)
					Value *val2(bop->getOperand(1));
					IntegerType *itype = dyn_cast<IntegerType>(BI->getType());

					// Skip non-Integer types
					if(itype == NULL)
					{
						BI++;
						continue;
					}

					const APInt ap_int_zero = APInt(itype->getBitWidth(), 0);
					const APInt ap_int_one  = APInt(itype->getBitWidth(), 1);

					ConstantInt *ci;
					ConstantInt *ci_zero = ConstantInt::get(val1->getContext(), ap_int_zero);
					ConstantInt *ci_one  = ConstantInt::get(val1->getContext(), ap_int_one);

					switch (bop->getOpcode()) // Fold depending on what operator is used
					{

					case Instruction::Add:  // Addition
						// 0 + x = x
						if(ConstantInt::classof(val1))
						{
							ci = dyn_cast<ConstantInt>(val1);
							if(ci->getValue().eq(ap_int_zero))
							{
								DBG(outs()<<"AlgebraicIdentities :: 0 + x = x \n");
								ReplaceInstWithValue(B.getInstList(), BI, val2);
								inst_removed = true;
								LocalOptsInfo.numAlgebraicOpts++;
							}

						}
						// x + 0 = x
						else if(ConstantInt::classof(val2))
						{
							ci = dyn_cast<ConstantInt>(val2);
							if(ci->getValue().eq(ap_int_zero))
							{
```

```
                                DBG(outs()<<"AlgebraicIdentities :: x + 0 = x \n");
                                ReplaceInstWithValue(B.getInstList(), BI, val1);
                                inst_removed = true;
                                LocalOptsInfo.numAlgebraicOpts++;
                        }
                }

                break;

        case Instruction::Sub:  // Subtraction
                // x - 0 = x
                if(ConstantInt::classof(val2))
                {
                        ci = dyn_cast<ConstantInt>(val2);
                        if(ci->getValue().eq(ap_int_zero))
                        {
                                DBG(outs()<<"AlgebraicIdentities :: x - 0 = x \n");
                                ReplaceInstWithValue(B.getInstList(), BI, val1);
                                inst_removed = true;
                                LocalOptsInfo.numAlgebraicOpts++;
                        }

                }
                //  x - x = 0
                else if(val1 == val2)
                {
                        DBG(outs()<<"AlgebraicIdentities :: x - x = 0 \n");
                        ReplaceInstWithValue(B.getInstList(), BI, ci_zero);
                        inst_removed = true;
                        LocalOptsInfo.numAlgebraicOpts++;
                }

                break;

        case Instruction::Mul:  // Multiplication
                // x * 1 = x, x * 0 = 0
                if(ConstantInt::classof(val2))
                {
                        ci = dyn_cast<ConstantInt>(val2);
                        if(ci->getValue().eq(ap_int_one))
                        {
                                DBG(outs()<<"AlgebraicIdentities :: x * 1 = x \n");
                                ReplaceInstWithValue(B.getInstList(), BI, val1);
                                inst_removed = true;
                                LocalOptsInfo.numAlgebraicOpts++;
                        }
                        else if(ci->getValue().eq(ap_int_zero))
                        {
                                DBG(outs()<<"AlgebraicIdentities :: x * 0 = 0 \n");
                                ReplaceInstWithValue(B.getInstList(), BI, ci_zero);
                                inst_removed = true;
                                LocalOptsInfo.numAlgebraicOpts++;
                        }
                }
                // 1 * x = x, 0 * x = 0
                else if(ConstantInt::classof(val1))
                {
                        ci = dyn_cast<ConstantInt>(val1);
                        if(ci->getValue().eq(ap_int_one))
                        {
                                DBG(outs()<<"AlgebraicIdentities :: 1 * x = x \n");
                                ReplaceInstWithValue(B.getInstList(), BI, val2);
                                inst_removed = true;
                                LocalOptsInfo.numAlgebraicOpts++;
                        }
                        else if(ci->getValue().eq(ap_int_zero))
                        {
```

```
                                DBG(outs()<<"AlgebraicIdentities :: 0 * x = 0 \n");
                                ReplaceInstWithValue(B.getInstList(), BI, ci_zero);
                                inst_removed = true;
                                LocalOptsInfo.numAlgebraicOpts++;
                        }

                }

                break;

        case Instruction::UDiv:  // Division
        case Instruction::SDiv:
                // x / 1 = x
                if(ConstantInt::classof(val2))
                {
                        ci = dyn_cast<ConstantInt>(val2);
                        if(ci->getValue().eq(ap_int_one))
                        {
                                DBG(outs()<<"AlgebraicIdentities :: x / 1 = x \n");
                                ReplaceInstWithValue(B.getInstList(), BI, val1);
                                inst_removed = true;
                                LocalOptsInfo.numAlgebraicOpts++;
                        }

                }
                //  0 / x = 0
                else if(ConstantInt::classof(val1))
                {
                        ci = dyn_cast<ConstantInt>(val1);
                        if(ci->getValue().eq(ap_int_zero))
                        {
                                DBG(outs()<<"AlgebraicIdentities :: 0 / x = 0 \n");
                                ReplaceInstWithValue(B.getInstList(), BI, ci_zero);
                                inst_removed = true;
                                LocalOptsInfo.numAlgebraicOpts++;
                        }
                }
                //  x / x = 1
                else if(val1 == val2)
                {
                        DBG(outs()<<"AlgebraicIdentities :: x / x = 1 \n");
                        ReplaceInstWithValue(B.getInstList(), BI, ci_one);
                        inst_removed = true;
                        LocalOptsInfo.numAlgebraicOpts++;
                }

                break;

        case Instruction::And:  // And
                // x && 0 = 0
                if(ConstantInt::classof(val2))
                {
                        ci = dyn_cast<ConstantInt>(val2);
                        if(ci->getValue().eq(ap_int_zero))
                        {
                                DBG(outs()<<"AlgebraicIdentities :: x && 0 = 0 \n");
                                ReplaceInstWithValue(B.getInstList(), BI, ci_zero);
                                inst_removed = true;
                                LocalOptsInfo.numAlgebraicOpts++;
                        }
                }
                // 0 && x = 0
                else if(ConstantInt::classof(val1))
                {
                        ci = dyn_cast<ConstantInt>(val1);
                        if(ci->getValue().eq(ap_int_zero))
                        {
```

```
                                DBG(outs()<<"AlgebraicIdentities :: 0 && x = 0 \n");
                                ReplaceInstWithValue(B.getInstList(), BI, ci_zero);
                                inst_removed = true;
                                LocalOptsInfo.numAlgebraicOpts++;
                        }
                }

                break;

        case Instruction::Or:  // Or
                // x || 0 = x
                if(ConstantInt::classof(val2))
                {
                        ci = dyn_cast<ConstantInt>(val2);
                        if(ci->getValue().eq(ap_int_zero))
                        {
                                DBG(outs()<<"AlgebraicIdentities :: x || 0 = x \n");
                                ReplaceInstWithValue(B.getInstList(), BI, val1);
                                inst_removed = true;
                                LocalOptsInfo.numAlgebraicOpts++;
                        }
                }
                // 0 || x = x
                else if(ConstantInt::classof(val1))
                {
                        ci = dyn_cast<ConstantInt>(val1);
                        if(ci->getValue().eq(ap_int_zero))
                        {
                                DBG(outs()<<"AlgebraicIdentities :: 0 || x = x \n");
                                ReplaceInstWithValue(B.getInstList(), BI, val2);
                                inst_removed = true;
                                LocalOptsInfo.numAlgebraicOpts++;
                        }
                }
                break;

        case Instruction::Xor:  // Xor
                // x ^ 0 = x
                if(ConstantInt::classof(val2))
                {
                        ci = dyn_cast<ConstantInt>(val2);
                        if(ci->getValue().eq(ap_int_zero))
                        {
                                DBG(outs()<<"AlgebraicIdentities :: x ^ 0 = x \n");
                                ReplaceInstWithValue(B.getInstList(), BI, val1);
                                inst_removed = true;
                                LocalOptsInfo.numAlgebraicOpts++;
                        }
                }
                // 0 ^ x = x
                else if(ConstantInt::classof(val1))
                {
                        ci = dyn_cast<ConstantInt>(val1);
                        if(ci->getValue().eq(ap_int_zero))
                        {
                                DBG(outs()<<"AlgebraicIdentities :: 0 ^ x = x \n");
                                ReplaceInstWithValue(B.getInstList(), BI, val2);
                                inst_removed = true;
                                LocalOptsInfo.numAlgebraicOpts++;
                        }
                }

                break;



        default:
```

```
                                        break;
                                }
                        }

                        // Increment the iterator only if the current instruction was not removed.
                        if(!inst_removed)
                                BI++;

                        // Modified block
                        if(inst_removed)
                                modified = true;
                }

                return modified;
        }

        // Constant Folding
        bool constant_folding(BasicBlock& B)
        {
                bool modified = false;

                // Iterate over all instructions
                for(BasicBlock::iterator BI = B.begin(); BI != B.end(); )
                {
                        Instruction& inst(*BI);
                        bool known_inst = true;
                        bool inst_removed = false;  // Is this instruction removed

                        // In binary operations
                        if(BinaryOperator *bop = dyn_cast<BinaryOperator>(&inst))
                        {
                                Value *val1(bop->getOperand(0));  // Get the first and the second operand (as values)
                                Value *val2(bop->getOperand(1));

                                if(!ConstantInt::classof(val1) || !ConstantInt::classof(val2)) {
                                        // The values are not const integers, then there’s nothing to do!
                                        ++BI;
                                        continue;
                                }

                                ConstantInt *ci1 = dyn_cast<ConstantInt>(val1); // Get constants from values
                                ConstantInt *ci2 = dyn_cast<ConstantInt>(val2);
                                ConstantInt *ci_final;

                                switch (bop->getOpcode()) // Fold depending on what operator is used
                                {

                                case Instruction::Add:  // Addition
                                        ci_final = get_const(ci1->getType(), (ci1->getSExtValue()) + (ci2->getSExtValue()));
                                        break;
                                case Instruction::Sub: // Subtraction
                                        ci_final = get_const(ci1->getType(), (ci1->getSExtValue()) - (ci2->getSExtValue()));
                                        break;
                                case Instruction::Mul: // Multiplication
                                        ci_final = get_const(ci1->getType(), (ci1->getSExtValue()) * (ci2->getSExtValue()));
                                        break;
                                case Instruction::UDiv: // Division
                                case Instruction::SDiv:
                                        if(ci2->getSExtValue() != 0)
                                                ci_final = get_const(ci1->getType(), (ci1->getSExtValue()) / (ci2->getSExtValue()));
                                        else
                                                known_inst = false; // Divide by zero
                                        break;
                                case Instruction::Shl:  // Left Shift
                                        ci_final = get_const(ci1->getType(), (ci1->getSExtValue()) << (ci2->getSExtValue()));
                                        break;
                                case Instruction::LShr: // (Logical) Right Shift
```

```
                                ci_final = get_const(ci1->getType(), (ci1->getSExtValue()) >> (ci2->getSExtValue()));
                                break;

                        default:
                                // Unknown instruction
                                known_inst = false;
                                break;
                        }

                        // Known instruction
                        if(known_inst)
                        {
                                DBG(outs() << "ConstantFolding: " << ci1->getSExtValue() << " " << bop->getOpcodeName() << " "  << ci2->getSExtValue()
                                    << " = " << ci_final->getSExtValue() << "\n");
                                ReplaceInstWithValue(B.getInstList(),BI,ci_final);
                                LocalOptsInfo.numConstantFolds++;
                                inst_removed = true;
                        }

                }

                // Increment the iterator only if the current instruction was not removed.
                // This is chosen instead of doing --BI when we remove the instruction because --BI crashes when BI is at B.begin()
                if (!inst_removed)
                        ++BI;

                // Modified block
                if(inst_removed)
                        modified = true;

        }

        return modified;
}

// Strength Reduction
bool strength_reduction(BasicBlock& B)
{
        bool modified = false;

        // Iterate over all instructions
        for(BasicBlock::iterator BI = B.begin(), BE = B.end(); BI != BE; ) {
                Instruction& inst(*BI);
                bool inst_removed = false;  // Is this instruction removed due to strength reduction

                // In binary operations
                if(BinaryOperator *bop = dyn_cast<BinaryOperator>(&inst))
                {
                        Value *val1(bop->getOperand(0));  // Get the first and the second operand (as values)
                        Value *val2(bop->getOperand(1));

                        switch (bop->getOpcode()) { // Switch on the operator

                        case Instruction::Mul:
                                // x * 2^k, 2^k * x ==> (x << k)
                                if (ConstantInt::classof(val1)) {   // Make sure the constant value, if any, is val2
                                        Value *t(val1);
                                        val1 = val2;
                                        val2 = t;
                                }

                                if (ConstantInt::classof(val2)) { // Now if it is really a constant
                                        ConstantInt *ci2 = dyn_cast<ConstantInt>(val2);
                                        int64_t log_i = 0;
                                        int64_t check_pow = find_log_improved(ci2->getSExtValue(),log_i);
                                        if (check_pow != -2) {  // if val2 is 2^k +/- 1
                                                BinaryOperator *modified_inst(BinaryOperator::Create(Instruction::Shl, val1, get_const(ci2->getType(), lo
```

```
g_i)));
                                                                if (check_pow == 0) { // val2 is 2^k --> only need one instruction
                                                                        DBG(outs() << "Replacing: " << val1->getName() << " * " << ci2->getSExtValue() << " with " << val
1->getName() << " << " << log_i << "\n");
                                                                        ReplaceInstWithInst(B.getInstList(),BI,modified_inst);
                                                                }
                                                                else if (check_pow == 1) {  // 2^k + 1
                                                                        DBG(outs() << "Replacing: " << val1->getName() << " * " << ci2->getSExtValue() << " with " << val
1->getName() << " << " << log_i << " + " << val1->getName() << "\n");
                                                                        B.getInstList().insert(BI,modified_inst);
                                                                        BinaryOperator *final_inst(BinaryOperator::Create(Instruction::Add, modified_inst, val1));
                                                                        ReplaceInstWithInst(B.getInstList(),BI,final_inst);
                                                                }
                                                                else { // 2^k - 1
                                                                        DBG(outs() << "Replacing: " << val1->getName() << " * " << ci2->getSExtValue() << " with " << val
1->getName() << " << " << log_i << " - " << val1->getName() << "\n");
                                                                        B.getInstList().insert(BI,modified_inst);
                                                                        BinaryOperator *final_inst(BinaryOperator::Create(Instruction::Sub, modified_inst, val1));
                                                                        ReplaceInstWithInst(B.getInstList(),BI,final_inst);
                                                                }
                                                                LocalOptsInfo.numStrengthReds++;
                                                        }
                                                }
                                                break;

                                        case Instruction::UDiv: // Division
                                        case Instruction::SDiv:
                                                // x / 2^k ==> (x >> k)
                                                if (ConstantInt::classof(val2)) { // If val2 is a constant
                                                        ConstantInt *ci2 = dyn_cast<ConstantInt>(val2);
                                                        int64_t log_i = find_log(ci2->getSExtValue());
                                                        if (log_i >= 0) { // If it is a power of 2
                                                                BinaryOperator *modified_inst(BinaryOperator::Create(Instruction::LShr, val1, get_const(ci2->getType(), l
og_i)));
                                                                DBG(outs() << "Replacing: " << val1->getName() << " / " << ci2->getSExtValue() << " with " << val1->getNa
me() << " >> " << log_i << "\n");
                                                                ReplaceInstWithInst(B.getInstList(),BI,modified_inst);
                                                                LocalOptsInfo.numStrengthReds++;
                                                        }
                                                }
                                                break;

                                        default:
                                                // Unhandled instruction
                                                break;
                                        }
                                }

                                // Increment the iterator only if the current instruction was not removed.
                                if (!inst_removed)
                                        ++BI;

                                // Modified block
                                if(inst_removed)
                                        modified = true;
                        }

                        return modified;
                }

                // Printing summary statistics
                void printLocalOptsSummary()
                {
                        outs() << "Transformations applied:\n";
                        outs() << "Algebraic identities: " << LocalOptsInfo.numAlgebraicOpts << "\n";
                        outs() << "Constant folding: " << LocalOptsInfo.numConstantFolds << "\n";
                        outs() << "Strength reduction: " << LocalOptsInfo.numStrengthReds << "\n";
```

```
		}

	public:
		static char ID;

		LocalOpts(): ModulePass(ID)
		{
		}

		~LocalOpts()
		{
		}

		// We only do local optimizations, so we don't modify the CFG.
		virtual void getAnalysisUsage(AnalysisUsage &AU) const
		{
			AU.setPreservesCFG();
		}

		virtual bool runOnModule(Module& M)
		{
			for (Module::iterator MI = M.begin(), ME = M.end(); MI != ME; ++MI)
			{
				Function& F(*MI);

				for(Function::iterator FI = F.begin(), FE = F.end(); FI != FE; ++FI) {
					BasicBlock& B(*FI);

					// In the debug mode, print the original code to stdout
					DBG(outs() << "ORIGINAL CODE:\n\n");
					DBG(B.dump());

					// In the debug mode, print every optimization performed to stdout
					DBG(outs() << "\nOPTIMIZATIONS PERFORMED:\n\n");

					// Loop till block is modified
					while(1)
					{
						bool modified = false;

						// Algebraic optimization pass
						modified = modified || algebraic_optimizations(B);
						if(modified)
						{
							DBG(B.dump());
							DBG(outs() << "-------------------------------\n");
						}

						// Constant folding pass
						modified = modified || constant_folding(B);
						if(modified)
						{
							DBG(B.dump());
							DBG(outs() << "-------------------------------\n");
						}

						// Strength reduction pass
						modified = modified || strength_reduction(B);
						if(modified)
						{
							DBG(B.dump());
							DBG(outs() << "-------------------------------\n");
						}

						if(!modified)
							break;
					};
```

```
                                        // In the debug mode, print the final code to stdout
                                        DBG(outs() << "\nFINAL CODE:\n\n");
                                        DBG(B.dump());
                                }
                        }

                        // Print the summary statistics to stdout
                        printLocalOptsSummary();
                        return true;
                }
        };

// Register the pass
        char LocalOpts::ID = 0;
        RegisterPass<LocalOpts> X("local-opts", "15745: Local Optimizations");
}
```

2.3 Test Cases

The test cases start from the next page.

```c
int compute ()
{
  int result = 0;
  int a = 2;
  int b = 3;
  int c = 4 + a + b;
  int d = 5;

  result += a;
  result += b;

  result = b / 32;
  result += 0;
  result = 0 + result;
  result -= 0;

  result += a * 2;
  result = d * 8;
  result -= b / 2;

  result *= 0;
  result *= 1;

  result -= b / 4;
  result *= (b/b);
  result += (b-b);
  result /= result;
  result -= result;
  //result = result && 0;

  result = a * 8;
  result *= c;
  result /= 2;
  return result;
}
```

```
#include <stdio.h>

unsigned triangularNumber(unsigned n);

const char *suffix[] = {" ","st","nd","rd"};

int abc(int argc, char *argv[]) {
    int n = 0,m;
    unsigned t;

    if ((argc < 2) ||
            (sscanf(argv[1],"%d",&n) != 1) ||
            (n <= 0)) {
        printf("\nusage %s <n>\n",argv[0]);
        printf(" where <n> is a positive integer\n");
        return -1;
    }
    t = (int)(n * (((float)n + 1.0) / 2.0));
    t *= 3;
    if (t == triangularNumber(n)) {
        m = n % 10;
        printf("\nThe %d%s triangular number is %d\n",
                n,
                ((m > 0) && (m < 4)) ? suffix[m] : "th",
                t);
    } else {
        printf("\nerror\n");
    }
    return 0;
}

unsigned triangularNumber(unsigned n) {
    if (n == 1) return 1;
    return n + triangularNumber(n-1);
}

void heapsort(int arr[], unsigned int N)
{
    unsigned int n = N, i = n/2, parent, child;
    int t;

    for (;;) { /* Loops until arr is sorted */
        if (i > 0) { /* First stage - Sorting the heap */
            i--;           /* Save its index to i */
            t = arr[i];    /* Save parent value to t */
        } else {     /* Second stage - Extracting elements in-place */
            n--;           /* Make the new heap smaller */
            if (n == 0) return; /* When the heap is empty, we are done */
            t = arr[n];    /* Save last value (it will be overwritten) */
            arr[n] = arr[0]; /* Save largest value at the end of arr */
        }

        parent = i; /* We will start pushing down t from parent */
        child = i*2 + 1; /* parent's left child */

        /* Sift operation - pushing the value of t down the heap */
        while (child < n) {
            if (child + 1 < n  &&  arr[child + 1] > arr[child]) {
                child++; /* Choose the largest child */
            }
            if (arr[child] > t) { /* If any child is bigger than the parent */
                arr[parent] = arr[child]; /* Move the largest child up */
                parent = child; /* Move parent pointer to this child */
                //child = parent*2-1; /* Find the next child */
                child = parent*2+1; /* the previous line is wrong*/
            } else {
                break; /* t's place is found */
```

```
            }
        }
        arr[parent] = t; /* We save t in the heap */
    }
}

#include "stdio.h"
#include "stdlib.h"

#define SCALE 10000
#define ARRINIT 2000

void pi_digits(int digits) {
    int carry = 0;
    int arr[digits + 1];
    for (int i = 0; i <= digits; ++i)
        arr[i] = ARRINIT;
    for (int i = digits; i > 0; i-= 14) {
        int sum = 0;
        for (int j = i; j > 0; --j) {
            sum = sum * j + SCALE * arr[j];
            arr[j] = sum % (j * 2 - 1);
            sum /= j * 2 - 1;
        }
        printf("%04d", carry + sum / SCALE);
        carry = sum % SCALE;
    }
}

int main(int argc, char** argv) {
    int n = argc == 2 ? atoi(argv[1]) : 100;
    pi_digits(n);

    return 0;
}
```

2.4 Expected Results

First test case :

```
Transformations applied:
Algebraic identities: 7
Constant folding: 19
Strength reduction: 0
```

Second test case :

```
Transformations applied:
Algebraic identities: 0
Constant folding: 0
Strength reduction: 6
```

3 CFG Basics

- The maximal basic blocks are outlined below.

Basic Block Label		Code
B1		x = 50 y = 8 if (x > y) goto L2
B2		x = 50 goto L3
B3	L1:	x = 27 return x
B4	L2:	y = x + 1 if (y < x) goto L2
B5		x = 24
B6	L3:	z = x + y switch (z) { 26 => L1 ∥ 32 => L4 ∥ default => L5 }
B7	L4:	print("success")
B8	L5:	x = 50 return x

- The CFG is given in Figure 1.

4 Available Expressions

The tables of EVAL, KILL, IN, and OUT sets obtained after doing the available expression analysis are provided below.

BB	**EVAL**	**KILL**
1	$\{c+d, a \times a\}$	$\{b+c, b+d, b \times b\}$
2	$\{i+d, c+d\}$	$\{b+c, b+d, b \times b\}$
3	$\{b+d, c+d\}$	$\{a \times a\}$
4	$\{b+d, b \times b\}$	$\emptyset$
5	$\emptyset$	$\{i+1\}$

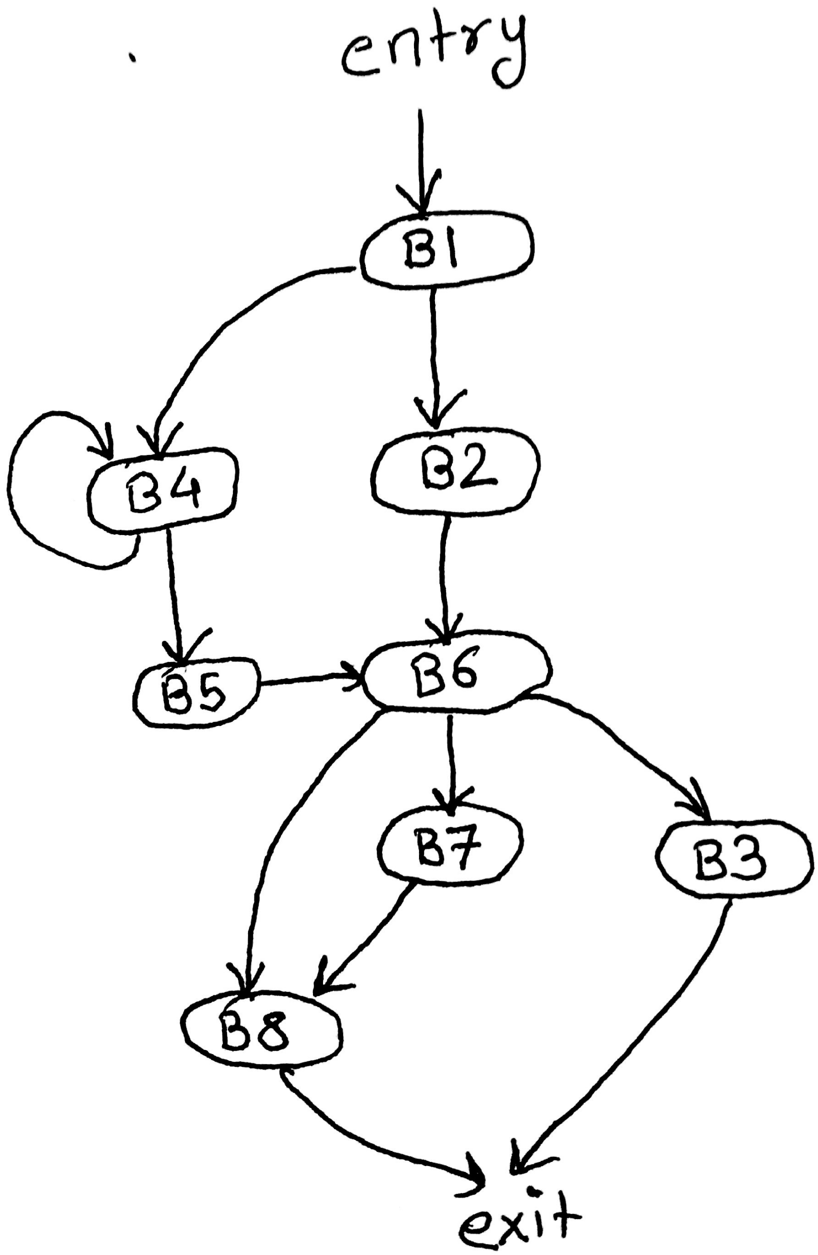

Figure 1: CFG

BB	IN	OUT
1	$\emptyset$	$\{c+d, a \times a\}$
2	$\{c+d\}$	$\{i+d, c+d\}$
3	$\{i+d, c+d\}$	$\{i+d, c+d, b+d\}$
4	$\{i+d, c+d\}$	$\{i+d, c+d, b+d, b \times b\}$
5	$\{i+d, c+d, b+d\}$	$\{c+d, b+d\}$

5 Faint Analysis

1. **What is the set of elements that your analysis operates on?**

   Sets of variables.

2. **What is the direction of your analysis?**

   Backwards (exit to entry).

3. **What is your transfer function? Be sure to clearly define any other sets that your transfer function uses (eg., GEN or KILL etc).**

   To identify the set of faint variables, we first identify the complementary set of **strongly live variables**. A strongly live variable is a variable that is used in the assignment of another strongly live variable or within an expression. If a variable is not strongly live, then it is a **faint variable**. That is, it is either dead or is used in the assignment of another faint variable.

   For any instruction s, let x be variable in the LHS. Let GEN_s be the set of variables in the RHS (like the live variable analysis) and let $KILL_s = \{x\}$. We use the following transfer function, where we **alter data flow only if x is strongly live**. That is :

$$IN(s) = \begin{cases} GEN_s \cup (OUT(s) - KILL_s) & \text{if } x \in OUT(s), \\ OUT(s) & \text{if } x \notin OUT(s). \end{cases}$$

   Now, during our data flow analysis, once we compute $OUT(B)$ for a basic block B, we iterate through the instructions of B in the reverse order, and apply the transfer function given above. When we reach the first instruction of the block, its IN will give us $IN(B)$. It seems that unlike the transfer function of live variable analysis, this transfer function is not composable (i.e., after composing over multiple instructions in the block, its final form is not of the same type $f(x) = GEN_x \cup (x - KILL_x))$ because the instructions on which we alter the data flow depends on $OUT(B)$ itself.

4. **What is your meet operator? Give the equation that uses the meet operator.**

   The meet operator for **strongly live variable analysis** is **union**. Thus,

   $OUT(B) = \cup_{B':successor(B)} IN(B')$.

   This is because a variable that is strongly live in any successor block must also strongly live at the exit of a basic block.

5. **To what value do you initialize exit and/or entry ?**

   $IN(exit) = \emptyset$

   This is because no variable is strongly live at exit. Note that the analysis is done backwards.

6. **To what values do you initialize the in or out sets ?**

   For all other nodes B, $IN(B) = \emptyset$. During the backwards pass, we will populate the IN sets using the meet operator.

7. **Does the order that your analysis visits basic blocks matter? What order would you implement and why ?**

   Yes, since we are doing a backwards analysis, we ideally want to visit all successors of a node before visiting the node itself. Thus, to make the analysis converge faster, we would need to visit the nodes in **postorder traversal** (reverse topological order) i.e. we analyse each node at the end of its recursive DFS.

8. **Will your analysis converge? Why (in words, not a proof) ?**

   Yes, intuitively, the IN and OUT sets of each basic block are n-bit vectors (where n is the number of variables). In each iteration, at least one more bit is set in the IN or OUT set of at least one basic block (otherwise we stop the analysis). Further, no bit that is set is ever unset. So, the analysis must converge in finitely many iterations. Formally, for this data flow framework, the semilattice is **monotone** and its height is **finite**. Therefore, the analysis is guaranteed to converge.

9. **Clearly describe in pseudo-code an algorithm that uses the result of your analysis to identify faint expressions.**

   We first compute the set of strongly live variables $SLV(s)$ at each program point s using the analysis described above. Once we have the result of that analysis, we use this equation to compute the set of faint variables $FV(s)$ at program point s,

   $$FV(s) = V \setminus SLV(s)$$

   Here, V is the set of variables in the program. Any expression *expr* at program point s that is being assigned to a faint variable $fv \in FV(s)$ is a **faint expression**. This can be done thus :

   for (each assignment instruction p) :{
   remove p if $LHS(s) \in FV(s)$, where s is the program point immediately after p
   }