Lab4.2 实验报告

姓名 学号

实验要求

本次实验需要认真阅读实验文档和论文，理解GVN算法，按照自己对于伪代码的理解，阅读助教的代码框架，理解里面的数据结构和变量的含义，然后设计自己的变量和数据结构，补全助教的代码框架。

之后通过在测试样例上和正确的答案的比较，不断进行debug，并完善自己的理解，分析之前没有想到的情况，并修改自己的代码思路，逐渐和正确的输出逼近，让代码的逻辑更加严密。

最后通过在线测试，进一步验证自己的代码实现。

实验难点

实验遇到的第一个挑战就是对算法的理解。一开始我对于等价类、值表达式、值编号等的理解都不太清晰，导致完全无法理解伪代码和助教的代码中很多变量和结构的含义。后来我通过仔细阅读文档和论文，并结合助教的提示，才终于理解了 `CongruenceClass` 中的 `index_, members_, leader_, value_expr_, value_phi_` 等变量在程序中的作用，由于一开始写得过于急躁，导致第一版代码错误太多，完全无法运行，只能在再次仔细阅读文档，有了更深理解之后，从头重新写了第二版代码。

实验遇到的第二个挑战是代码的调试。由于对于c++的一些细节的理解并不是很深，所以出现了很多语言层面的bug。比如shared_pointer的使用，以及dynamic_pointer_cast(我搞了半天才意识到智能指针不能用dynamic_cast直接转换)。这些语言上的错误导致我很长一段时间程序都无法通过编译，报出很多鲜红的错误。然而更加难以排除的是一些很隐蔽的bug，比如在比较Expression的时候，我从早上debug到下午，才发现原来我用 == 比较了PhiExpression指针，但是 == 只对 Expression 指针有重载，对它的子类不起作用，所以这里会直接用内置的比较指针的方法比较 PhiExpression指针，于是这里的比较就总是不正确。后来我用dynamic_pointer_cast把 PhiExpression 强制转为 Expression才解决了这个问题。

实验遇到的第三个挑战是循环的依赖问题。因为一开始没有考虑到这个问题，我的所有等价类都是直接用 value_expr_区分的，导致循环出现了无限递归。我试了好几种解决方案，最后终于通过构建Expression的子类 IndexExpression, 用 `index_` 来标记等价类，并且每一次循环的开始都把 `next_value_number_` 置为1，经过反复调试才终于实现了迭代的正确收敛。

实验遇到的第四个挑战是常量折叠。一开始我只考虑了指令中的两边操作数都是 ConstantExpression 的情况，没有注意到按照我的设计，还有可能是 IndexExpression，IndexExpression 对应的等价类是常量。后来经过仔细判断，我才终于考虑到所有情况，通过了效率测试。

实验设计

首先最顶层的是 `detectEquivalences` 函数，根据助教的提示，需要进行深度优先遍历来访问各个基本块，所以我还编写了 `dfs_bb` 来实现这一点。

```
void GVN::detectEquivalences(std::unique_ptr<FuncInfo> &func_info_) {
    bool changed = false;
    Top.insert(std::make_shared<CongruenceClass>(0));
    // initialize pout with top
    for (auto &bb : func_->get_basic_blocks()) {
        assert(&bb != nullptr);
```

```
        pout_[&bb] = Top;
    }
    bool is_first = true;

    // iterate until converge
    do {
        changed = false;
        next_value_number_ = 1;
        // see the pseudo code in documentation
        for (auto &bb : func_->get_basic_blocks()) { // you might need to visit
the blocks in depth-first order
            // get PIN of bb by predecessor(s)
            // iterate through all instructions in the block
            // and the phi instruction in all the successors
            if (visited.find(&bb) != visited.end()) continue;
            visited.insert(&bb);
            if(dfs_bb(&bb, is_first, func_info_) == true) changed = true;
            if(is_first == true) is_first = false;
        }
        visited.clear();
    } while (changed);
}
```

在`dfs_bb`中，我按照深度优先的顺序对基本块进行了访问，在访问的时候对于基本块中的每一个指令都使用`transferFunction`进行流式分析，根据 pins和指令来修改 pouts，从而得到最后基本块的对应`pout_`的值。

```
bool GVN::dfs_bb(BasicBlock * bb_ptr, bool is_first, std::unique_ptr<FuncInfo>
&func_info_) {
    bool changed = false;
    if (is_first == true) {
        pin_[bb_ptr].clear();
    }
    else {
        if (bb_ptr->get_pre_basic_blocks().size() == 2) {
            pin_[bb_ptr] = join(pout_[bb_ptr->get_pre_basic_blocks().front()],
pout_[bb_ptr->get_pre_basic_blocks().back()]);
        }
        else if (bb_ptr->get_pre_basic_blocks().size() == 1){
            pin_[bb_ptr] = pout_[bb_ptr->get_pre_basic_blocks().front()];
        }
    }

    partitions pins = pin_[bb_ptr];
    partitions pouts;
    for (auto &instr : bb_ptr->get_instructions()) {
        if (instr.is_phi()) continue;
        pouts = transferFunction(&instr, &instr, pins, func_info_);
        pins = pouts;
    }
    for (auto &bb_succ_ptr : bb_ptr->get_succ_basic_blocks()) {
        for (auto &instr : bb_succ_ptr->get_instructions()) {
            if (instr.is_phi()) {
                for (int i = 1; i < instr.get_operands().size(); i += 2) {
                    if(instr.get_operand(i) == bb_ptr) {
```

```
                        pouts = transferFunction(&instr, instr.get_operand(i -
1), pins, func_info_);
                        pins = pouts;
                    }
                }
            }
        }

    if((pout_[bb_ptr] == pouts) == false){
        pout_[bb_ptr] = pouts;
        changed = true;
    }

    for (auto &bb_succ_ptr : bb_ptr->get_succ_basic_blocks()) {
        if (visited.find(bb_succ_ptr) == visited.end()){
            visited.insert(bb_succ_ptr);
            if(dfs_bb(bb_succ_ptr, false, func_info_) == true) changed = true;
        }
    }

    return changed;
}
```

其中`transferFunction`的实现如下

```
GVN::partitions GVN::transferFunction(Instruction *x, Value *e, partitions &pin,
std::unique_ptr<FuncInfo> &func_info_) {
    partitions pout = clone(pin);
    for (auto &Ci : pout) {
        if (Ci->members_.find(x) != Ci->members_.end()) {
            Ci->members_.erase(x);
        }
        if (Ci->members_.size() == 0) {
            pout.erase(Ci);
            break;
        }
    }

    auto ve = valueExpr(e, func_info_, pout);
    if (ve == nullptr) return pout;
    auto vpf = valuePhiFunc(x, x->get_parent(), ve, pout);
    bool inserted = false;
    if (ve->get_expr_type() == Expression::e_index) {
        auto ve_new = std::dynamic_pointer_cast<IndexExpression>(ve);
        for (auto &C : pout) {
            if (C->index_ == ve_new->get_index()) {
                C->members_.insert(x);
            }
        }
        inserted = true;
    }
    for (auto &Ci : pout) {
        if (inserted == true) break;
        if (Ci->value_expr_ == ve) {
            Ci->members_.insert(x);
```

```
            inserted = true;
            break;
        }
        if ((vpf == nullptr) == false && (std::dynamic_pointer_cast<Expression>
(Ci->value_phi_) == std::dynamic_pointer_cast<Expression>(vpf))) {
            Ci->members_.insert(x);
            inserted = true;
            break;
        }
    }

    if (inserted == false) {
        auto Cj = createCongruenceClass(next_value_number_++);
        if (ve->get_expr_type() == Expression::e_constant) {
            Cj->leader_ = std::dynamic_pointer_cast<ConstantExpression>(ve)-
>get_val();
        }
        else {
            Cj->leader_ = x;
        }
        Cj->value_expr_ = ve;
        Cj->value_phi_ = vpf;
        Cj->members_.insert(x);
        while(pout.find(Cj) != pout.end()){
            Cj->index_++;
            next_value_number_++;
        }
        pout.insert(Cj);
    }
    return pout;
}
```

基本上是按照伪代码的逻辑，先删除pin中的这条指令，然后计算 value expression 和 value phi expression, 并根据结果修改pout。

其中`valueExpr`实现如下：

```
shared_ptr<Expression> GVN::valueExpr(Value * val, std::unique_ptr<FuncInfo>
&func_info_, const partitions & P) {
    // TODO
    /* if this is a value that alread have an index */

    for (auto it = P.begin(); it != P.end(); it++) {
        auto tmp = (*it)->members_.find(val);
        if(tmp != (*it)->members_.end()) {
            return IndexExpression::create((*it)->index_);
        }
    }

    /* if this is a constant */
    if (dynamic_cast<Constant*>(val) != nullptr) {
        std::shared_ptr<ConstantExpression> ve =
GVNExpression::ConstantExpression::create(dynamic_cast<Constant*>(val));
        return ve;
    }
```

```
    /* if this is an argument */
    else if (dynamic_cast<Argument*>(val) != nullptr) {
        std::shared_ptr<ArguExpression> ve =
GVNExpression::ArguExpression::create(dynamic_cast<Argument*>(val));
        return ve;
    }

    /* if this is a global variable */
    else if (dynamic_cast<GlobalVariable*>(val) != nullptr) {
        std::shared_ptr<GlobalExpression> ve =
GVNExpression::GlobalExpression::create(dynamic_cast<GlobalVariable*>(val));
        return ve;
    }

    /* if this is an instruction */
    if (dynamic_cast<Instruction*>(val) != nullptr) {
        Instruction * instr = dynamic_cast<Instruction*>(val);
        if (instr->is_void()) {
            return nullptr;
        }

        if (instr->isBinary()) {
            Value * lval = instr->get_operand(0);
            Value * rval = instr->get_operand(1);
            if ((dynamic_cast<Constant*>(lval) == nullptr ||
dynamic_cast<Constant*>(rval) == nullptr) == false) {
                Constant* tmp = folder_->compute(instr, dynamic_cast<Constant*>
(lval), dynamic_cast<Constant*>(rval));
                return valueExpr(tmp, func_info_, P);
            }
            auto lhs = valueExpr(lval, func_info_, P);
            auto rhs = valueExpr(rval, func_info_, P);
            auto val_const = fold_constant(instr, lhs, rhs, P, P, P);
            if (val_const != nullptr) return val_const;
            Instruction::OpID op = instr->get_instr_type();
            std::shared_ptr<BinaryExpression> ve = BinaryExpression::create(op,
lhs, rhs);
            return ve;
        }

        else if (instr->is_cmp()) {
            auto lhs = valueExpr(instr->get_operand(0), func_info_, P);
            auto rhs = valueExpr(instr->get_operand(1), func_info_, P);
            CmpInst::CmpOp op = dynamic_cast<CmpInst*>(instr)->get_cmp_op();
            std::shared_ptr<CmpExpression> ve =
GVNExpression::CmpExpression::create(op, lhs, rhs);
            return ve;
        }

        else if (instr->is_fcmp()) {
            auto lhs = valueExpr(instr->get_operand(0), func_info_, P);
            auto rhs = valueExpr(instr->get_operand(1), func_info_, P);
            FCmpInst::CmpOp op = dynamic_cast<FCmpInst*>(instr)->get_cmp_op();
            std::shared_ptr<FCmpExpression> ve =
GVNExpression::FCmpExpression::create(op, lhs, rhs);
            return ve;
        }
```

```
        else if (instr->is_alloca()) {
            std::shared_ptr<AllocaExpression> ve =
GVNExpression::AllocaExpression::create(instr);
            return ve;
        }

        else if (instr->is_load()) {
            std::shared_ptr<LoadExpression> ve =
GVNExpression::LoadExpression::create(instr);
            return ve;
        }

        else if (instr->is_call()) {
            Instruction * instr_new = instr;
            Function * func = dynamic_cast<Function*>(instr_new-
>get_operand(0));
            bool is_pure = func_info_->is_pure_function(func);
            std::vector<std::shared_ptr<Expression>> args;
            for (int i = 1; i < instr_new->get_num_operand(); i++){
                args.push_back(valueExpr(instr_new->get_operand(i), func_info_,
P));
            }
            std::shared_ptr<CallExpression> ve =
GVNExpression::CallExpression::create(func, args, args.size(), is_pure, instr);
            return ve;
        }

        else if (instr->is_gep()) {
            Instruction * instr_new = instr;
            std::vector<std::shared_ptr<Expression>> indxs;
            for (int i = 0; i < instr_new->get_num_operand(); i++){
                indxs.push_back(valueExpr(instr_new->get_operand(i), func_info_,
P));
            }
            std::shared_ptr<GepExpression> ve =
GVNExpression::GepExpression::create(indxs, indxs.size());
            return ve;
        }

        else if (instr->is_si2fp()) {
            Value* val = instr->get_operand(0);
            if ((dynamic_cast<Constant*>(val) == nullptr) == false) {
                Constant * tmp = folder_->compute(instr, dynamic_cast<Constant*>
(val));
                return valueExpr(tmp, func_info_, P);
            }
            auto orig = valueExpr(instr->get_operand(0), func_info_, P);
            auto tmp = fold_constant(instr, orig, P);
            if (tmp != nullptr) {
                return tmp;
            }
            std::shared_ptr<Si2fpExpression> ve =
GVNExpression::Si2fpExpression::create(orig, dynamic_cast<SiToFpInst*>(instr)-
>get_dest_type());
            return ve;
        }

        else if (instr->is_fp2si()) {
```

```
            Value* val = instr->get_operand(0);
            if ((dynamic_cast<Constant*>(val) == nullptr) == false) {
                Constant * tmp = folder_->compute(instr, dynamic_cast<Constant*>
(val));
                return valueExpr(tmp, func_info_, P);
            }
            auto orig = valueExpr(instr->get_operand(0), func_info_, P);
            auto tmp = fold_constant(instr, orig, P);
            if (tmp != nullptr) {
                return tmp;
            }
            std::shared_ptr<Fp2siExpression> ve =
GVNExpression::Fp2siExpression::create(orig, dynamic_cast<FpToSiInst*>(instr)-
>get_dest_type());
            return ve;
        }

        else if (instr->is_zext()) {
            Value* val = instr->get_operand(0);
            if ((dynamic_cast<Constant*>(val) == nullptr) == false) {
                Constant * tmp = folder_->compute(instr, dynamic_cast<Constant*>
(val));
                return valueExpr(tmp, func_info_, P);
            }
            auto orig = valueExpr(val, func_info_, P);
            auto tmp = fold_constant(instr, orig, P);
            if (tmp != nullptr) {
                return tmp;
            }
            std::shared_ptr<ZextExpression> ve =
GVNExpression::ZextExpression::create(orig, dynamic_cast<ZextInst*>(instr)-
>get_dest_type());
            return ve;
        }

        else {
            return nullptr;
        }
    }

    else {
        return nullptr;
    }
}
```

这是一个递归构造的过程，符合expression本身的递归定义性质，递归到找到式子组成部分所在的等价类，或者遇到Constant等叶子节点为止。

在`BinaryExpression`中，我融合进了常量折叠，使用下面的函数进行表达式向常量的转换

```
std::shared_ptr<Expression> GVN::fold_constant(Instruction * instr,
std::shared_ptr<Expression> lhs, std::shared_ptr<Expression> rhs, const
partitions & Pl, const partitions & Pr, const partitions & P) {
    if (lhs->get_expr_type() == Expression::e_index
    && get_C(std::dynamic_pointer_cast<IndexExpression>(lhs)->get_index(), Pl)
!= nullptr
```

```
    && get_C(std::dynamic_pointer_cast<IndexExpression>(lhs)->get_index(), Pl)-
>value_expr_ != nullptr
    && get_C(std::dynamic_pointer_cast<IndexExpression>(lhs)->get_index(), Pl)-
>value_expr_->get_expr_type() == Expression::e_constant) {
        if (rhs->get_expr_type() == Expression::e_index
        && get_C(std::dynamic_pointer_cast<IndexExpression>(rhs)->get_index(),
Pr) != nullptr
        && get_C(std::dynamic_pointer_cast<IndexExpression>(rhs)->get_index(),
Pr)->value_expr_ != nullptr
        && get_C(std::dynamic_pointer_cast<IndexExpression>(rhs)->get_index(),
Pr)->value_expr_->get_expr_type() == Expression::e_constant) {
            auto lhs_const = get_C(std::dynamic_pointer_cast<IndexExpression>
(lhs)->get_index(), Pl)->leader_;
            auto rhs_const = get_C(std::dynamic_pointer_cast<IndexExpression>
(rhs)->get_index(), Pr)->leader_;
            Constant* tmp = folder_->compute(instr, dynamic_cast<Constant*>
(lhs_const), dynamic_cast<Constant*>(rhs_const));
            return valueExpr(tmp, func_info_, P);
        }
        else if (rhs->get_expr_type() == Expression::e_constant) {
            auto lhs_const = get_C(std::dynamic_pointer_cast<IndexExpression>
(lhs)->get_index(), Pl)->leader_;
            auto rhs_const = std::dynamic_pointer_cast<ConstantExpression>(rhs)-
>get_val();
            Constant* tmp = folder_->compute(instr, dynamic_cast<Constant*>
(lhs_const), rhs_const);
            return valueExpr(tmp, func_info_, P);
        }
    }
    else if (lhs->get_expr_type() == Expression::e_constant) {
        if (rhs->get_expr_type() == Expression::e_index
        && get_C(std::dynamic_pointer_cast<IndexExpression>(rhs)->get_index(),
Pr) != nullptr
        && get_C(std::dynamic_pointer_cast<IndexExpression>(rhs)->get_index(),
Pr)->value_expr_ != nullptr
        && get_C(std::dynamic_pointer_cast<IndexExpression>(rhs)->get_index(),
Pr)->value_expr_->get_expr_type() == Expression::e_constant) {
            auto lhs_const = std::dynamic_pointer_cast<ConstantExpression>(lhs)-
>get_val();
            auto rhs_const = get_C(std::dynamic_pointer_cast<IndexExpression>
(rhs)->get_index(), Pr)->leader_;
            Constant* tmp = folder_->compute(instr, lhs_const,
dynamic_cast<Constant*>(rhs_const));
            return valueExpr(tmp, func_info_, P);
        }
        else if (rhs->get_expr_type() == Expression::e_constant) {
            auto lhs_const = std::dynamic_pointer_cast<ConstantExpression>(lhs)-
>get_val();
            auto rhs_const = std::dynamic_pointer_cast<ConstantExpression>(rhs)-
>get_val();
            Constant* tmp = folder_->compute(instr, lhs_const, rhs_const);
            return valueExpr(tmp, func_info_, P);
        }
    }
    return nullptr;
}
```

```
std::shared_ptr<Expression>  GVN::fold_constant(Instruction * instr,
std::shared_ptr<Expression> orig, const partitions & P) {
    if (orig->get_expr_type() == Expression::e_index
    && get_C(std::dynamic_pointer_cast<IndexExpression>(orig)->get_index(), P)
!= nullptr
    && get_C(std::dynamic_pointer_cast<IndexExpression>(orig)->get_index(), P)-
>value_expr_ != nullptr
    && get_C(std::dynamic_pointer_cast<IndexExpression>(orig)->get_index(), P)-
>value_expr_->get_expr_type() == Expression::e_constant){
        auto orig_const = dynamic_cast<Constant*>
(get_C(std::dynamic_pointer_cast<IndexExpression>(orig)->get_index(), P)-
>leader_);
        Constant* tmp = folder_->compute(instr, orig_const);
        return valueExpr(tmp, func_info_, P);
    }
    else {
        return nullptr;
    }
}
```

主要的思想就是判断是否可以算出常量，如果可以就算，不可以就返回空指针。由于需要判断是否是`IndexExpression`指向的等价类是常数（为此还要先搜索到对应的等价类），所以使用了大量的判断，表达式写的比较复杂。

下面是`valuePhiFunc`，由于再次牵涉到`IndexExpression`，所以需要大量强制转换和查找判断，导致整个程序比较混乱。不过大体结构还是按照伪代码的思路写的。

```
shared_ptr<PhiExpression> GVN::valuePhiFunc(Instruction * instr, BasicBlock
*bb_ptr, shared_ptr<Expression> ve, const partitions &P) {
    // TODO
    if (ve->get_expr_type() == Expression::e_bin) {
        std::shared_ptr<GVNExpression::BinaryExpression> ve_bi =
std::dynamic_pointer_cast<BinaryExpression>(ve);
        std::shared_ptr<Expression> lve_bi = ve_bi->get_lve();
        std::shared_ptr<Expression> rve_bi = ve_bi->get_rve();
        if (lve_bi->get_expr_type() == Expression::e_index && rve_bi-
>get_expr_type() == Expression::e_index) {
            std::shared_ptr<GVNExpression::IndexExpression> lve_idx =
std::dynamic_pointer_cast<IndexExpression>(lve_bi);
            std::shared_ptr<GVNExpression::IndexExpression> rve_idx =
std::dynamic_pointer_cast<IndexExpression>(rve_bi);
            if (get_C(lve_idx->get_index(), P) == nullptr || get_C(rve_idx-
>get_index(), P) == nullptr) return nullptr;
            std::shared_ptr<GVNExpression::PhiExpression> lve_phi =
get_C(lve_idx->get_index(), P)->value_phi_;
            std::shared_ptr<GVNExpression::PhiExpression> rve_phi =
get_C(rve_idx->get_index(), P)->value_phi_;
            if ((lve_phi == nullptr || rve_phi == nullptr) == false) {
                std::shared_ptr<BinaryExpression> vei =
BinaryExpression::create(ve_bi->get_op(), lve_phi->get_lve(), rve_phi-
>get_lve());
                auto vei_const = fold_constant(instr, lve_phi->get_lve(),
rve_phi->get_lve(), pout_[bb_ptr->get_pre_basic_blocks().front()], pout_[bb_ptr-
>get_pre_basic_blocks().front()], pout_[bb_ptr-
>get_pre_basic_blocks().front()]);
                auto vi = getVN(pout_[bb_ptr->get_pre_basic_blocks().front()],
vei_const);
```

```
                if (vi == nullptr){
                    vi = getVN(pout_[bb_ptr->get_pre_basic_blocks().front()],
vei);
                    if (vi == nullptr) {
                        vi = valuePhiFunc(instr, bb_ptr-
>get_pre_basic_blocks().front(), vei, pout_[bb_ptr-
>get_pre_basic_blocks().front()]);
                    }
                }
                std::shared_ptr<BinaryExpression> vej =
BinaryExpression::create(ve_bi->get_op(), lve_phi->get_rve(), rve_phi-
>get_rve());
                auto vej_const = fold_constant(instr, lve_phi->get_rve(),
rve_phi->get_rve(), pout_[bb_ptr->get_pre_basic_blocks().back()], pout_[bb_ptr-
>get_pre_basic_blocks().back()], pout_[bb_ptr->get_pre_basic_blocks().back()]);
                auto vj = getVN(pout_[bb_ptr->get_pre_basic_blocks().back()],
vej_const);
                if (vj == nullptr) {
                    vj = getVN(pout_[bb_ptr->get_pre_basic_blocks().back()],
vej);
                    if (vj == nullptr) {
                        vj = valuePhiFunc(instr, bb_ptr-
>get_pre_basic_blocks().back(), vej, pout_[bb_ptr-
>get_pre_basic_blocks().back()]);
                    }
                }
                if ((vi == nullptr || vj == nullptr) == false) {
                    return PhiExpression::create(vi, vj);
                }
            }
        }
    }
    return nullptr;
}
```

另外就是`join`和`intersect`

```
GVN::partitions GVN::join(const partitions &P1, const partitions &P2) {
    // TODO: do intersection pair-wise
    if (P1 == Top) return P2;
    else if (P2 == Top) return P1;
    partitions P;
    P.clear();
    for (auto &Ci : P1) {
        for (auto &Cj : P2) {
            std::shared_ptr<CongruenceClass> Ck = intersect(Ci, Cj);
            if (Ck->members_.size() > 0) {
                while(P.find(Ck) != P.end()){
                    Ck->index_++;
                    next_value_number_++;
                }
                P.insert(Ck);
            }
        }
    }
    return P;
}
```

```
std::shared_ptr<CongruenceClass> GVN::intersect(std::shared_ptr<CongruenceClass>
Ci,
                                                std::shared_ptr<CongruenceClass>
Cj) {
    // TODO
    std::shared_ptr<CongruenceClass> Ck = createCongruenceClass(0);
    if (Ci->index_ == Cj->index_) {
        Ck->index_ = Ci->index_;
    }
    if (Ci->leader_ == Cj->leader_) {
        Ck->leader_ = Ci->leader_;
    }
    if (Ci->value_expr_ == Cj->value_expr_) {
        Ck->value_expr_ = Ci->value_expr_;
    }
    if (std::dynamic_pointer_cast<Expression>(Ci->value_phi_) ==
std::dynamic_pointer_cast<Expression>(Cj->value_phi_)) {
        Ck->value_phi_ = Ci->value_phi_;
    }
    std::set_intersection(Ci->members_.begin(), Ci->members_.end(),
                          Cj->members_.begin(), Cj->members_.end(),
                          std::inserter(Ck->members_ , Ck->members_.begin() ));

    if (Ci->leader_ == Cj->leader_) {
        Ck->index_ = Ci->index_;
        Ck->value_expr_ = Ci->value_expr_;
        Ck->value_phi_ = Ci->value_phi_;
        Ck->leader_ = Ci->leader_;
    }

    else if (Ck->members_.size() > 0 && Ck->index_ == 0) {
        Ck->index_ = next_value_number_++;
        Ck->leader_ = *(Ck->members_.begin());
        std::shared_ptr<GVNExpression::Expression> lhs =
GVNExpression::IndexExpression::create(Ci->index_);
        std::shared_ptr<GVNExpression::Expression> rhs =
GVNExpression::IndexExpression::create(Cj->index_);
        std::shared_ptr<GVNExpression::PhiExpression> vph =
GVNExpression::PhiExpression::create(lhs, rhs);
        Ck->value_phi_ = vph;
    }

    return Ck;
}
```

基本上和伪代码一致。不过一开始`intersect`只考虑到了处理phi时带来的相同`members_`，没有意识到如果循环的话有其他传下来的变量也会让`members_`交集不为空，导致出现了一些冗余的phi，我通过对`leader_`的额外判断排除了这种情况。

上面就是大体程序实现的框架，还有一些辅助的运算符重载等，比如对于比较partition和CongruenceClass,

```
bool operator==(const GVN::partitions &p1, const GVN::partitions &p2) {
    // TODO: how to compare partitions?
    if (p1.size() != p2.size()) {
        return false;
```

```
    }
    std::set<std::shared_ptr<CongruenceClass>>::iterator it1;
    std::set<std::shared_ptr<CongruenceClass>>::iterator it2;
    for (it1 = p1.begin(); it1 != p1.end(); it1++){
        for (it2 = p2.begin(); it2 != p2.end(); it2++){
            if (*(*it1) == *(*it2)) break;
        }
        if (it2 == p2.end()){
            return false;
        }
    }
    return true;
}

bool CongruenceClass::operator==(const CongruenceClass &other) const {
    // TODO: which fields need to be compared?

    if (this->index_ == 0 && other.index_ == 0) return true;
    if ((this->index_ == other.index_ && this->leader_ == other.leader_ && this-
>value_expr_ == other.value_expr_ && std::dynamic_pointer_cast<Expression>(this-
>value_phi_) == std::dynamic_pointer_cast<Expression>(other.value_phi_) && this-
>members_.size() == other.members_.size()) == false) {
        return false;
    }
    std::set<Value *> tmp;
    std::set_intersection(this->members_.begin(), this->members_.end(),
                          other.members_.begin(), other.members_.end(),
                          std::inserter(tmp , tmp.begin()));
    if (tmp.size() == this->members_.size()){
        return true;
    }
    else{
        return false;
    }
}
```

还有Expression子类的设计，主要是根据指令的种类设计相应的Expression子类。

```
class AllocaExpression : public Expression {
  public:
    static std::shared_ptr<AllocaExpression> create(Instruction *instr) { return
std::make_shared<AllocaExpression>(instr); }
    virtual std::string print() { std::string str = "alloca"; return str; }
    // we leverage the fact that alloca instrs in lightIR have unique addresses
(hopfully)
    bool equiv(const AllocaExpression *other) const { return AllocaInstr ==
other->AllocaInstr; }
    AllocaExpression(Instruction *instr) : Expression(e_alloca),
AllocaInstr(instr) {}

  private:
    Instruction* AllocaInstr;
};

class LoadExpression : public Expression {
  public:
```

```
    static std::shared_ptr<LoadExpression> create(Instruction* LoadInstr) {
return std::make_shared<LoadExpression>(LoadInstr); }
    virtual std::string print() { std::string str = "load"; return str; }
    // we leverage the fact that alloca instrs in lightIR have unique addresses
(hopfully)
    bool equiv(const LoadExpression *other) const { return LoadInstr ==
LoadInstr; }
    LoadExpression(Instruction* LoadInstr) : Expression(e_load),
LoadInstr(LoadInstr) {}

  private:
    Instruction* LoadInstr;
};

class IndexExpression : public Expression {
  public:
    static std::shared_ptr<IndexExpression> create(int index) { return
std::make_shared<IndexExpression>(index); }
    virtual std::string print() { std::string str = "v" + std::to_string(index);
return str; }
    // we leverage the fact that alloca instrs in lightIR have unique addresses
(hopfully)
    bool equiv(const IndexExpression *other) const { return index == other-
>index; }
    IndexExpression(int index) : Expression(e_index), index(index) {}
    int get_index() { return index; }

  private:
    int index;
};

class Si2fpExpression : public Expression {
  public:
    static std::shared_ptr<Si2fpExpression> create(std::shared_ptr<Expression>
orig, Type *dest_ty_) { return std::make_shared<Si2fpExpression>(orig,
dest_ty_); }
    virtual std::string print() { std::string str = "si2fp"; return str; }
    // we leverage the fact that alloca instrs in lightIR have unique addresses
(hopfully)
    bool equiv(const Si2fpExpression *other) const { return (*(orig) == *(other-
>orig) && dest_ty_ == other->dest_ty_); }
    Si2fpExpression(std::shared_ptr<Expression> orig, Type *dest_ty_) :
Expression(e_si2fp), orig(orig), dest_ty_(dest_ty_) {}
    Type * get_dest_type() const { return dest_ty_; }

  private:
    std::shared_ptr<Expression> orig;
    Type *dest_ty_;
};

............
```

为此还需要扩展Expression的==重载函数

```
bool GVNExpression::operator==(const Expression &lhs, const Expression &rhs) {
    if (lhs.get_expr_type() != rhs.get_expr_type())
        return false;
```

```
    switch (lhs.get_expr_type()) {
    case Expression::e_constant: return equiv_as<ConstantExpression>(lhs, rhs);
    case Expression::e_bin: return equiv_as<BinaryExpression>(lhs, rhs); //bug
here
    case Expression::e_phi: return equiv_as<PhiExpression>(lhs, rhs);
    case Expression::e_cmp: return equiv_as<CmpExpression>(lhs, rhs);
    case Expression::e_fcmp: return equiv_as<FCmpExpression>(lhs, rhs);
    case Expression::e_alloca: return equiv_as<AllocaExpression>(lhs, rhs);
    case Expression::e_load: return equiv_as<LoadExpression>(lhs, rhs);
    case Expression::e_call: return equiv_as<CallExpression>(lhs, rhs);
    case Expression::e_gep: return equiv_as<GepExpression>(lhs, rhs);
    case Expression::e_si2fp: return equiv_as<Si2fpExpression>(lhs, rhs);
    case Expression::e_fp2si: return equiv_as<Fp2siExpression>(lhs, rhs);
    case Expression::e_zext: return equiv_as<ZextExpression>(lhs, rhs);
    case Expression::e_index: return equiv_as<IndexExpression>(lhs, rhs);
    case Expression::e_argu: return equiv_as<ArguExpression>(lhs, rhs);
    case Expression::e_global: return equiv_as<GlobalExpression>(lhs, rhs);
    }
}
```

下面我们用 bin.cminus优化前后进行对比：

优化前：

```
declare i32 @input()

declare void @output(i32)

declare void @outputFloat(float)

declare void @neg_idx_except()

define i32 @main() {
label_entry:
  %op0 = call i32 @input()
  %op1 = call i32 @input()
  %op2 = icmp sgt i32 %op0, %op1
  %op3 = zext i1 %op2 to i32
  %op4 = icmp ne i32 %op3, 0
  br i1 %op4, label %label5, label %label14
label5:                                           ; preds = %label_entry
  %op6 = add i32 33, 33
  %op7 = add i32 44, 44
  %op8 = add i32 %op6, %op7
  br label %label9
label9:                                           ; preds = %label5,
%label14
  %op10 = phi i32 [ %op8, %label5 ], [ %op17, %label14 ]
  %op11 = phi i32 [ %op7, %label5 ], [ %op16, %label14 ]
  %op12 = phi i32 [ %op6, %label5 ], [ %op15, %label14 ]
  call void @output(i32 %op10)
  %op13 = add i32 %op12, %op11
  call void @output(i32 %op13)
  ret i32 0
label14:                                          ; preds = %label_entry
  %op15 = add i32 55, 55
  %op16 = add i32 66, 66
```

```
  %op17 = add i32 %op15, %op16
  br label %label9
}
```

优化后：

```
declare i32 @input()

declare void @output(i32)

declare void @outputFloat(float)

declare void @neg_idx_except()

define i32 @main() {
label_entry:
  %op0 = call i32 @input()
  %op1 = call i32 @input()
  %op2 = icmp sgt i32 %op0, %op1
  %op3 = zext i1 %op2 to i32
  %op4 = icmp ne i32 %op3, 0
  br i1 %op4, label %label5, label %label14
label5:                                           ; preds = %label_entry
  br label %label9
label9:                                           ; preds = %label5,
%label14
  %op10 = phi i32 [ 154, %label5 ], [ 242, %label14 ]
  call void @output(i32 %op10)
  call void @output(i32 %op10)
  ret i32 0
label14:                                           ; preds = %label_entry
  br label %label9
}
```

可以看出指令条数明显减少，由于常量折叠，所以之前的那些 `%op6 = add i32 33, 33`，`%op7 = add i32 44, 44`，`%op8 = add i32 %op6, %op7` 等都成了常量不需要再计算，最后整合成了 `%op10 = phi i32 [ 154, %label5 ], [ 242, %label14 ]`，而 `%op13` 通过 `valuePhiFunc` 可以判断和 `%op10` 是一样的，所以也被替换，最后指令条数减少了一半。

可见 GVN 算法优化还是很有成效的。

思考题

1. 请简要分析你的算法复杂度

   如果程序有n条指令，那么一个partition有 $O(n)$ 个等价类，每个等价类有 $O(v)$ 大小，v 为程序中变量和常量个数。intersection应该可以在 $O(v)$ 时间内完成，由于有 n^2 量级的intersection，所以一个join需要 $O(n^2 \cdot v)$ 时间，如果有 j 个join，那么join总共需要 $O(n^2 \cdot v \cdot j)$ 时间。

   transfer function需要产生和查找 value expression 或 value ϕ-function, 一个value expression 产生和查找要 $O(n)$时间，计算value ϕ-function需要递归查找值表达式，大概需要 $O(n \cdot j)$ 时间。所以所有指令加起来要 $O(n^2 \cdot j)$ 时间。

   所以一次迭代总共时间复杂度为 $O(n^2 \cdot v \cdot j)$ 时间。

   而最坏情况下要迭代n次，所以最坏时间复杂度 $O(n^3 \cdot v \cdot j)$.

2. `std::shared_ptr`如果存在环形引用，则无法正确释放内存，你的 Expression 类是否存在 circular reference?

   应该是没有的。因为根据树状结构的迭代可以知道，最后迭代的终点是IndexExpression 或者ConstantExpression等，而这些节点里面并没有对其他的Expression子类指针的调用，之前的引用也都是单向的，所以不存在环形引用。

3. 尽管本次实验已经写了很多代码，但是在算法上和工程上仍然可以对 GVN 进行改进，请简述你的 GVN 实现可以改进的地方

   感觉我的代码中有很多重复的部分，比如大量的重复性的判断，以及大量的指针强制类型转换，也许可以通过巧妙的标记来避免一些这种不优雅的代码形式。

   而且我的Expression子类感觉有很多是重复而且相似的，比如 icmp 和 fcmp, 基本上是一样的，其实可以进行合并，这样就不会显得很冗余。

   同时，我的一些地方进行了大量的线性遍历查找，也许可以通过一些合适的数据结构和算法（比如二分查找）来减少这部分带来的不必要的时间复杂度。

实验总结

本次实验中，我进一步对c++的一些特性有了更深刻的理解，比如智能指针，运算符重载，子类和父类的关系等等，意识到了自己对于编程语言方面的理解和掌握还是有很大缺陷的，说明以后还需要多加练习。

通过对于一个以前没有接触过的算法的阅读，理解和实现，我的学习能力和动手能力有了较大的提升，通过不断遇到问题和思考，试探，解决问题，我处理bug的能力也有所提升。

通过对于 GVN 的理解，我对于流式分析也有了更深的理解，感受到如何通过不断迭代，最后得到理想的结果，感受到了这种流式分析巧妙的设计和强大的潜力，在以后的学习和科研中也能够运用这种方法。

通过本次实验，我对于编译优化的作用也有了更为清晰的认识，意识到其实一开始生成的代码是有很大的优化空间的，感受到了编译器的强大。

实验反馈

本次实验难度较大（对与我来说），不过收获也是很大的，对于我的代码能力有所锻炼，也让我感受到了这方面的短板，以后需要多加训练。