Lab4.2实验报告

PB20111637 黎帆

实验要求

请按照自己的理解，写明本次实验需要做什么

Global Value Numbering（Gvn）全局值编号，是一个基于 SSA 格式的优化，通过建立变量，表达式到值编号的映射关系，从而检测到冗余计算代码并删除。本次的实验的结构结合了参考论文**Detection of Redundant Expressions: A Complete and Polynomial-Time Algorithm in SSA**。改论文提出了一种适合 SSA IR 、多项式算法复杂度的数据流分析方式的算法，能够实现对冗余代码完备的检测。本次实验要求我们结合论文、lecture-31、之前提供的`Light IR` 等等实现该数据流分析算法，从而达到删除冗余代码、优化的目的。

在本次实验中，请仔细阅读3.1 GVN pass 实现内容要求，根据要求补全`src/optimization/GVN.cpp`，`include/optimization/GVN.h`中关于 GVN pass 数据流分析部分，同时需要在 `Reports/4-ir-opt/` 目录下撰写实验报告。

补充完全整体框架，增添所需要用的类。

实验难点

实验中遇到哪些挑战

- 头文件（GVN.h）中类的补充，与之前的实验不同，本次实验需要自己设计补充相关类，框架总体上也不具有定式，对于躺在定式框架舒适圈中的我来说是一个非常大的挑战。
- GVN.cpp中的valuePhiFunc函数中存在大量的关系式判断和赋值，容易出现逻辑错误或者值类型不匹配等问题，之前在这方面花费了很多时间调试逻辑关系和值类型的匹配。
- 这算是自己少有的大量使用C++语言的实验，在之前几年对C++这一语言的联系较少，所以存在对语法不够熟悉，导致自己在编译过程中花费大量时间debug。同时还拘泥于以前的c语言思维，导致在实现部分功能的时候耗时久、见效少。
- 样例很稀缺，也很难按照执行流程完整理解

实验设计

detect

初始化pin_、pout并且对全局变量和参数函数进行处理。通过深度优先对块进行遍历，大体上的思路和伪代码中的思路相同。

```
void GVN::detectEquivalences() {
    // initialize pout with top
    // iterate until convergence
    bool changed=false;
    int flag = 0;
    partitions Top;
    pin_[func_->get_entry_block()]={};
    Top.insert(createCongruenceClass(0)); // init
    auto module = func_->get_parent();
    // initialize pout with top
    // iterate until converge
    for (auto &bb : func_->get_basic_blocks()) {
        pout_[&bb] = std::move(Top);
    }
    for (auto &arg : func_->get_args()) {
        bool empty = (arg)->set_name("arg" +
std::to_string(next_value_number_));
        auto value = ValueExpression::create(arg);
        auto cc = createCongruenceClass(next_cc_number_++, arg, value);
        pin_[func_->get_entry_block()].insert(cc);
        if (empty) {
            next_value_number_ = next_value_number_ + 1;
        }
    }
    for (auto &globalVar : module->get_global_variable()) {
        bool empty = false;
        auto value = ValueExpression::create(&globalVar);
        auto cc = createCongruenceClass(next_cc_number_++, &globalVar,
value);
        pin_[func_->get_entry_block()].insert(cc);
        if (empty) {
            next_value_number_ = next_value_number_ + 1;
        }
    }
    auto entry=func_->get_entry_block();
    Instruction *stmt_entry;
    for (auto &instr : entry->get_instructions()) {
```

```
        stmt_entry = &instr;
        break;
    }
    BasicBlock *bb;
    // LOG_INFO<<"first instruction: "<<stmt_entry->print()<<"\n";
    pout_[func_->get_entry_block()] =
std::move(transferFunction(stmt_entry, stmt_entry, pin_[func_-
>get_entry_block()],bb));
    int it = 1;
    do {
         changed=false;
        // 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
            bool pre_empty=false;
            auto pre_bbs=(&bb)->get_pre_basic_blocks();
            pre_empty=pre_bbs.empty();
            if (pre_empty) {
                (&bb)->set_name("label_entry");
            }
            else {
                bool empty=(&bb)->set_name("label" +
std::to_string(next_value_number_));
                if (empty) {
                    next_value_number_++;
                }
            }

           if (pre_bbs.size()==1) {
                if (!pre_empty) {
                    pin_[&bb]=pout_[pre_bbs.front()];
                }
            }
            else if (pre_bbs.size()==2) {
                auto l_pre = pre_bbs.front();
                auto r_pre = pre_bbs.back();
                if (pout_[r_pre] == Top) {
                    pin_[(&bb)] = clone(pout_[l_pre]);
                } else if (pout_[l_pre] == Top) {
                    pin_[(&bb)] = clone(pout_[r_pre]);
                } else {
```

```
                    pin_[&bb] = std::move(join(pout_[l_pre],
pout_[r_pre]));
                }
            }
            else {
                if (!pre_empty) {
                    pin_[&bb]=pout_[pre_bbs.front()];
                }
            }
            // get PIN of bb by predecessor(s)
            auto p = clone(pin_[&bb]);
            for (auto &instr : bb.get_instructions()) {
                p = std::move(transferFunction(&instr, &instr, p,&bb));
            }
            for (auto bb_suc : (&bb)->get_succ_basic_blocks()) {
                for (auto &instr : bb_suc->get_instructions()) {
                    if ((&instr && (&instr)->is_phi())) {
                        auto lhs = (&instr)->get_operand(0);
                        auto l_bb = (&instr)->get_operand(1);
                        auto rhs = (&instr)->get_operand(2);
                        auto r_bb = (&instr)->get_operand(3);
                        if ((&bb)->get_name() == r_bb->get_name()) {
                            p = std::move(transferFunction(&instr, rhs,
p,&bb));
                        } else if ((&bb)->get_name() == l_bb-
>get_name()) {
                            p = std::move(transferFunction(&instr, lhs,
p,&bb));
                        } else {
                            p = std::move(transferFunction(&instr, rhs,
p,&bb));
                        }
                    }
                }
            }
            LOG_INFO<<"OK-succeed-basicblocks-detect";
            // LOG_INFO<<"bb: "<<(&bb)->print();
            // for (auto &cc : p)
            //     LOG_INFO << utils::print_congruence_class(*cc);
            // LOG_INFO<<"OK-partition\n";
            // check changes in pout
            if (!operator==(p,pout_[&bb])) {
                if(it>10){
                    for(auto &cc : p)
                        LOG_INFO << utils::print_congruence_class(*cc);
```

```
                    LOG_INFO<<"partition pout_, bb: "<<(&bb)->print();
                    for(auto &cc : pout_[&bb])
                        LOG_INFO << utils::print_congruence_class(*cc);
                    getchar();
                }
                changed = true;
            }
            LOG_INFO<<"compare finish: "<<changed;
            pout_[&bb] = std::move(p);
        }
        it++;
    } while (changed);
}
```

join

按照伪代码的思路，对Ci、Cj取交集。使用is_pure_function()判断语句是否为pure，进一步确定是否插入member。

```
GVN::partitions GVN::join(const partitions &P1, const partitions &P2) {
    // TODO: do intersection pair-wise
    bool flag = false;
    partitions newp={};
    if(P1==P2){
        return P1;
    }
    else{
        for (auto &Ci:P1) {
            for (auto &Cj:P2) {
                auto Ck = intersect(Ci,Cj);
                if(Ck==nullptr||Ck->members_.empty()) continue;
                //if p contain same value,join Ck into it
                flag = false;
                for(auto &cc:newp){
                    if(*(cc->value_expr_)==*(Ck->value_expr_)){
                        bool ck_pure=func_info_-
>is_pure_function(std::dynamic_pointer_cast<FunExpression>(Ck-
>value_expr_)->get_fun());
                        bool cc_pure=func_info_-
>is_pure_function(std::dynamic_pointer_cast<FunExpression>(cc-
>value_expr_)->get_fun());
                        if(Ck->value_expr_-
>get_expr_type()==Expression::e_fun &&!(ck_pure&&cc_pure)){
                        }
```

```
                        else {
                            flag = true;
                            for(auto member:Ck->members_){
                                cc->members_.insert(member);
                            }
                            break;
                        }
                    }
                }
                if(!flag){
                    newp.insert(Ck);
                }
            }
        }
        return newp;
    }
    return P1;
}
```

intersect

对应伪代码编写,编写时需要注意逻辑和值对应关系，否则容易出现混淆。

```
std::shared_ptr<CongruenceClass>
GVN::intersect(std::shared_ptr<CongruenceClass> Ci,

 std::shared_ptr<CongruenceClass> Cj) {
    // TODO
    auto Cnew=createCongruenceClass(0);
    if(*Ci==*Cj){
        if(Cj->index_<Ci->index_){
            return Cj;
        }
        else {
            return Ci;
        }
    }
    else{
        if(Cj->index_ == Ci->index_){
            Cnew->index_=Cj->index_;
        }
        auto name_i=Ci->leader_->get_name();
        auto name_j=Cj->leader_->get_name();
        if(name_i==name_j){
```

```
            Cnew->leader_=Ci->leader_;
        }
        if(Ci->value_==Cj->value_){
            Cnew->value_=Cj->value_;
        }
        for(auto i:Ci->members_){
            for(auto j:Cj->members_){
                if(i->get_name()==j->get_name()){
                    Cnew->members_.insert(i);
                }
            }
        }//both own
        Cnew->index_=(Cj->index_!=Ci->index_)?next_cc_number_++:Cnew-
>index_;
        Cnew->leader_=(Ci->leader_->get_name()!=Cj->leader_-
>get_name())?*(Cnew->members_.begin()):Cnew->leader_;
        Cnew->value_ =(Cj->value_!=Ci->value_)?
ValueExpression::create(Cnew->leader_):Cnew->value_ ;
        if(Cnew->members_.empty()){
            return nullptr;
        }
        if(Ci->value_expr_&&Cj->value_expr_){
            if(*(Ci->value_expr_)==*(Cj->value_expr_)){
                Cnew->value_expr_=(Ci->index_<Cj->index_)?Ci-
>value_expr_:Cj->value_expr_;
            }
            else{
                Cnew->value_expr_=PhiExpression::create(Ci->value_,Cj-
>value_);
            }
        }
        return Cnew;
    }
    return Ci;
}
```

valueExpr

代码结构非常复杂，需要通过头文件先对Expression进行一个划分，再在valueExpr中的分支中处理。

在处理时，先判断是否为 `const` 变量，若为 `const` 则通过 `ConstantExpression` 处理，否则不断细分到所属 Expression 类型。

```
shared_ptr<Expression> GVN::valueExpr(Instruction *instr, partitions
pin) {
    // TODO
    //LOG_INFO<<"func: value expression";
    auto instr_inst = dynamic_cast<Instruction *>(instr);
    bool is_phi=false;
    bool is_binary=false;
    bool is_cmp=false;
    bool is_fcmp=false;
    bool is_gep=false;
    bool is_si2fp=false;
    bool is_fp2si=false;
    bool is_zext=false;
    bool is_call=false;
    if(instr){
        is_phi=instr->is_phi();
        is_binary=instr->isBinary();
        is_cmp=instr->is_cmp();
        is_fcmp=instr->is_fcmp();
        is_gep=instr->is_gep();
        is_si2fp=instr->is_si2fp();
        is_fp2si=instr->is_fp2si();
        is_zext=instr->is_zext();
        is_call=instr->is_call();
    }
    if (dynamic_cast<Constant *>(instr)) {
        return ConstantExpression::create(dynamic_cast<Constant *>
(instr));
    }
    else if (!instr) {
        return nullptr;
    }
    // if(instr->is_phi()){
    //     LOG_INFO<<"phi_ve: "+instr->get_name()+" "+instr->print();
    // }
    else{
    for (auto it = pin.begin(); it != pin.end(); it++){
        for (auto it=pin.begin();it!=pin.end();it++) {
            for (auto m:(*it)->members_) {
                if (instr->get_name()==m->get_name()) {
                    // LOG_INFO<<"for x: "<<x->print()<<"find ve: "<<e-
>print();
                    return (*it)->value_expr_;
                }
            }
```

```
        }
    //LOG_INFO<<"value expression is from instr";
        if(is_binary){
            auto l=instr->get_operand(0);
            auto r=instr->get_operand(1);
            auto l_con = get_binary_Con(pin,l,instr);
            auto r_con = get_binary_Con(pin,r,instr);
            if(l_con&&r_con){
                return ConstantExpression::create(folder_-
>compute(instr,l_con,r_con));
            }
            auto bin_lhs = get_c(pin,l,instr);
            auto bin_rhs = get_c(pin,r,instr);
            auto op = instr->get_instr_type();
            return BinaryExpression::create(op,bin_lhs,bin_rhs);
        }
    else if(is_phi){
        LOG_INFO<<"phi instr: "<<instr->print();
        auto l=instr->get_operand(0);
        auto r=instr->get_operand(2);
        auto lbb=instr->get_operand(1);
        auto l_bb=dynamic_cast<BasicBlock *>(lbb);
        auto rbb=instr->get_operand(3);
        auto r_bb=dynamic_cast<BasicBlock *>(rbb);
        std::shared_ptr<GVNExpression::Expression> phi_l;
        std::shared_ptr<GVNExpression::Expression> phi_r;
        phi_l=get_c(pout_[l_bb],l,instr);
        phi_r=get_c(pout_[r_bb],r,instr);
        auto it = pout_[l_bb].begin();
        auto ve_l=(*it)->value_expr_;
        auto ve_r=(*it)->value_expr_;
        ve_l=nullptr;
        ve_r=nullptr;
            for (auto it = pout_[l_bb].begin(); it != pout_[l_bb].end();
it++) {
                if ((*it)->value_ && *phi_l == *(*it)->value_) {
                    ve_l=(*it)->value_expr_;
                }
            }
            for (auto it = pout_[r_bb].begin(); it != pout_[r_bb].end();
it++) {
                if ((*it)->value_ && *phi_r== *(*it)->value_) {
                    ve_r=(*it)->value_expr_;
                }
            }
```

```
            if((ve_l&&ve_r)&& *ve_l==*ve_r){
                auto phi_con =
std::dynamic_pointer_cast<ConstantExpression>(phi_l);
                if(phi_con){
                    return phi_con;
                }
                //LOG_INFO<<"same, return "<< ve_l;
                return ve_l;
            }
            return PhiExpression::create(phi_l,phi_r);
    }
    else if(is_cmp){
            auto l=instr->get_operand(0);
            auto r=instr->get_operand(1);
            auto l_con=dynamic_cast<Constant *>(l);
            auto r_con=dynamic_cast<Constant *>(r);
            shared_ptr<Expression> bin_l=nullptr;
            shared_ptr<Expression> bin_r=nullptr;
            if(l_con&&r_con){
                return ConstantExpression::create(folder_-
>compute(instr,l_con,r_con));
            }
            else if(!l_con){
                bin_l=get_c(pin,l,instr);
                bin_r=get_c(pin,r,instr);
            }
            else if(!r_con){
                bin_l=get_c(pin,l,instr);
                bin_r=get_c(pin,r,instr);
            }
            else{
            }
            auto op=instr->get_instr_type();
            return CmpExpression::create(op,bin_l,bin_r);
        }

        else if(is_fcmp){
            auto l=instr->get_operand(0);
            auto r=instr->get_operand(1);
            auto l_con=dynamic_cast<Constant *>(l);
            auto r_con=dynamic_cast<Constant *>(r);
            shared_ptr<Expression> bin_l=nullptr;
            shared_ptr<Expression> bin_r=nullptr;
            if(l_con&&r_con){
```

```
                return ConstantExpression::create(folder_-
>compute(instr,l_con,r_con));
            }
            else if(!l_con){
                bin_l=get_c(pin,l,instr);
                bin_r=get_c(pin,r,instr);
            }
            else if(!r_con){
                bin_l=get_c(pin,l,instr);
                bin_r=get_c(pin,r,instr);
            }
            else{
            }
            auto op=instr->get_instr_type();
            return CmpExpression::create(op,bin_l,bin_r);
        }
    else if(is_gep){
            std::list<std::shared_ptr<Expression>> args;
            for(auto op:instr->get_operands()){
                auto e = get_c(pin,op,instr);
                args.push_back(e);
            }
            auto gep=GepExpression::create(args);
            return gep;
        }
    else if(is_si2fp||is_fp2si||is_zext){
            auto val = instr->get_operand(0);
            auto con =dynamic_cast<Constant *>(val);
            if(!con){
                std::string name;
                name=val->get_name();
                auto valE=(*pin.begin())->value_expr_;
                valE=nullptr;
                for (auto it=pin.begin();it!=pin.end();it++){
                    for(auto mem:(*it)->members_){
                        if(name==mem->get_name()){
                            valE = (*it)->value_expr_;
                            break;
                        }
                    }
                    if(valE) break;
                }
                if(valE && valE-
>get_expr_type()==Expression::e_constant)
```

```
                    con=std::dynamic_pointer_cast<ConstantExpression>
(valE)->get_constant();
            }
            if(con) return ConstantExpression::create(folder_-
>compute(instr,con));
            auto trans_v = get_c(pin,val,instr);
            return TransExpression::create(instr-
>get_instr_type(),trans_v);
        }
    else if(is_call){
        auto fun = dynamic_cast<Function *>(instr->get_operand(0));
        auto name = fun->get_name();
        std::list<std::shared_ptr<Expression>> args;
        for(auto op : instr->get_operands()){
            if(fun == dynamic_cast<Function *>(op)) continue;
            auto e = get_c(pin,op,instr);
            args.push_back(e);
        }
        return FunExpression::create(name,args,fun);
    }
    }
    return ValueExpression::create(instr);
}
}
```

ValuePhiFunc()

先判断ve是否具有φk(vi1, vj1) ⊕ φk(vi2, vj2)的形式，之后找到值编码对应的Value_phi_，如果两边都是Phi指令的话，进行伪代码的如下后续操作：

```
shared_ptr<PhiExpression> GVN::valuePhiFunc(shared_ptr<Expression> ve,
BasicBlock * bb, const partitions &P) {
    // TODO
    shared_ptr<Expression> l_l= NULL;
    shared_ptr<Expression> l_r = NULL;
    shared_ptr<Expression> r_l = NULL;
    shared_ptr<Expression> r_r = NULL;
    auto bin = std::dynamic_pointer_cast<BinaryExpression>(ve);
    if(bin){
        // LOG_INFO<<"bin phi check\n";
        auto m=P.begin();
        auto bin_l=(*m)->value_expr_;
        auto bin_r=(*m)->value_expr_;
        bin_l=nullptr;
```

```
        bin_r=nullptr;
        for (auto it = P.begin(); it != P.end(); it++) {
            if ((*it)->value_ && *(bin->get_lhs()) == *(*it)->value_) {
                bin_l = (*it)->value_expr_;
            }
        }
        for (auto it = P.begin(); it != P.end(); it++) {
            if ((*it)->value_ && *(bin->get_rhs()) == *(*it)->value_) {
                bin_r = (*it)->value_expr_;
            }
        }
        auto l_phi = std::dynamic_pointer_cast<PhiExpression>(bin_l);
        auto r_phi = std::dynamic_pointer_cast<PhiExpression>(bin_r);
        auto l_bb = bb->get_pre_basic_blocks().front();
        auto r_bb = bb->get_pre_basic_blocks().back();
        if(l_phi&&r_phi){
            l_l = l_phi->get_lhs();
            r_l = l_phi->get_rhs();
            l_r = r_phi->get_lhs();
            r_r = r_phi->get_rhs();
            auto l_l_con = std::dynamic_pointer_cast<ConstantExpression>
(l_l);
            auto r_l_con = std::dynamic_pointer_cast<ConstantExpression>
(r_l);
            auto l_r_con = std::dynamic_pointer_cast<ConstantExpression>
(l_r);
            auto r_r_con = std::dynamic_pointer_cast<ConstantExpression>
(r_r);
            if (!(l_l_con)) {
                if (l_l) {
                    for (auto it = pout_[l_bb].begin(); it !=
pout_[l_bb].end(); it++) {
                        if ((*it)->value_ && *(l_l) == *(*it)->value_) {
                            l_l = (*it)->value_expr_;
                        }
                    }
                }
                l_l_con = std::dynamic_pointer_cast<ConstantExpression>
(l_l);
            }
            if (!(r_l_con)) {
                if (r_l) {
                    for (auto it = pout_[r_bb].begin(); it !=
pout_[r_bb].end(); it++) {
                        if ((*it)->value_ && *(r_l) == *(*it)->value_) {
```

```
                            r_l = (*it)->value_expr_;
                        }
                    }
                }
                r_l_con = std::dynamic_pointer_cast<ConstantExpression>
(r_l);
            }
            if (!(l_r_con)) {
                if (l_r) {
                    for (auto it = pout_[l_bb].begin(); it !=
pout_[l_bb].end(); it++) {
                        if ((*it)->value_ && *(l_r) == *(*it)->value_) {
                            l_r = (*it)->value_expr_;
                        }
                    }
                }
                l_r_con = std::dynamic_pointer_cast<ConstantExpression>
(l_r);
            }
            if (!(r_r_con)) {
                if (r_r) {
                    for (auto it = pout_[r_bb].begin(); it !=
pout_[r_bb].end(); it++) {
                        if ((*it)->value_ && *(r_r) == *(*it)->value_) {
                            r_r = (*it)->value_expr_;
                        }
                    }
                }
                r_r_con = std::dynamic_pointer_cast<ConstantExpression>
(r_r);
            }
            std::shared_ptr<Expression> l_mge;
            std::shared_ptr<Expression> r_mge;
            if (l_l_con && l_r_con) {
                LOG_INFO << "con";
                l_mge=ConstantExpression::create(folder_->compute(bin-
>get_op(),l_l_con->get_constant(),l_r_con->get_constant()));
            }
            if (r_l_con && r_r_con) {
                r_mge=ConstantExpression::create(folder_->compute(bin-
>get_op(),r_l_con->get_constant(),r_r_con->get_constant()));
            }
            if (!l_mge)
                l_mge = BinaryExpression::create(bin->get_op(), l_l,
l_r);
```

```
            if (!r_mge)
                r_mge = BinaryExpression::create(bin->get_op(), r_l,
r_r);
            auto l_v = getVN(pout_[l_bb], l_mge);
            auto r_v = getVN(pout_[r_bb], r_mge);
            if (l_v && r_v) {
                return PhiExpression::create(l_v, r_v);
            }
        }
    }
    return nullptr;
}
```

优化前后的IR对比：

下面我们通过bin.cminus在开启优化和不开启优化的情况下进行比较：

bin.cminus

```
int main(void) {
    int a;
    int b;
    int c;
    int d;
    if (input() > input()) {
        a = 33 + 33;
        b = 44 + 44;
        c = a + b;
    } else {
        a = 55 + 55;
        b = 66 + 66;
        c = a + b;
    }
    output(c);
    d = a + b;
    output(d);
}
```

Before Pass

```
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
}
```

After Pass

```
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
}
```

可以清楚看到其中冗余的操作数（op6 、op7、op 8、op 11、op 12、op 13、op 15、op 16、op 17）在开启Pass优化后被删除，同时对d的计算也因为之前计算的c=a+b而简化，其余的被删去的指令转化为第三个基本块的phi函数，也有常量传播。

思考题

1. 请简要分析你的算法复杂度：

   根据论文中的时间复杂度分析和算法导论，原英文如下

   ```
   Let there be n number of expressions in a program. By definitions
   of Join and transferFunction a partition can have O(n) classes with
   each class of O(v) size, where v is the number of variables and
   constants in the program. The join operation is class-wise
   intersection of partitions. With efficient data structure that
   supports lookup, intersection of each class takes O(v) time. With a
   total of n2 such intersections, a join takes O(n2.v) time. If there
   are j join points, the total time taken by all the join operations
   in an iteration is O(n2.v.j). The transfer function involves
   construction and lookup of value expression or value φ-function in
   the input partition. A value expression is computed and searched
   for in O(n) time. Computation of value φ-function for an expression
   x+y essentially involves lookup of value expressions, recursively,
   in partitions at left and right predecessors of a join block. If a
   lookup table is maintained to map value expressions to value φ-
   functions (or NULL when a value expression does not have a value φ-
   function), then computation of a value φ-function can be done in
   O(n.j) time. Thus transfer function of a statement x = e takes
   O(n.j) time. In a program with n expressions total time taken by
   all the transfer functions in an iteration is O(n2.j). Thus the
   time taken by all the joins and transfer functions in an iteration
   is O(n2.v.j). As shown in [4], in the worst case the iterative
   analysis takes n iterations and hence the total time taken by the
   analysis is O(n3.v.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?

   答:存在，例子如下：

   ```
   %op1 = phi %op2,%op3

      %op2 = add %op1, 1
   ```

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

   可以对GVN实现进行改进，首先Expression类别太多，判断耗时过长，降低性能。可以考虑采用方法内连等方法进行优化。在自己的代码中，可以考虑将常量折叠转移到函数中解决，减少ValueExpr逻辑判断耗时。

实验总结

本次实验我了解到了GVN的原理、算法结构，同时在实现过程中使用了重载、智能指针和模版类等C++特性，对我后续使用C++这一工具有了引导作用，也让我认识到了C++的强大之处。

非常遗憾，本次实验的分数没有拿满，很感谢辛勤付出的主角和群里无私解答问题的同学，收获很大，也学到了很多别人的优点。

感谢老师和助教!