# Lab4.2 实验报告

姓名 吕凯盛
学号 PB20081590

## 实验要求

1. 实现一个数据流分析，分析程序中的值关系，并得到冗余指令的信息，实现冗余指令的减少
2. 补充GVN.h和GVN.cpp两个程序，补充相关子类和GVN算法内容

## 实验难点

1. c++基础很薄弱，看代码和写代码时遇到了很多问题
2. 算法伪代码和实际实现起来差别很大，有很多东西需要自己设计
3. debug很复杂，段错误一直出现
4. 方向走错了重做代价过大

## 实验设计

1. detect

```
detectEquivalences(G)
    PIN1 = {} // “1” is the first statement in the program
    POUT1 = transferFunction(PIN1)
    for each statement s other than the first statement in the program
        POUTs = Top
    while changes to any POUT occur // i.e. changes in equivalences
        for each statement s other than the first statement in the
program
            if s appears in block b that has two predecessors
            then
                PINs = Join(POUTs1, POUTs2) // s1 and s2 are last
statements in respective predecessors
            else
                PINs = POUTs3    // s3 the statement just before s
            POUTs = transferFunction(PINs) // apply transferFunction on
each statement in the block
```

设计1：顶元是用一个空的partition，然后再join处特判

设计2：然后pins只需要在每一个bb前进行修改即可，不需要每一条语句都进行一个join

设计3：扫描完一个bb后，扫描其所有的后继bb，把相应的phi指令加到对应的等价类，即 `x3=x1`

```
if(lbb == &bb)
                        {
                            //加到相同的member
                            for(auto &lbbInst : dynamic_cast<BasicBlock
*>(lbb)->get_instructions())
                            {

                                for(auto eachCC : p)
                                {
```

```
                                    if(eachCC-
>members_.find(lphi)!=eachCC->members_.end())
                                    {
                                        eachCC->members_.emplace(&instr);
                                        break;
                                    }

                                }

                            }
                            if(dynamic_cast<Constant * >(lphi))
                            {
                                auto C
=ConstantExpression::create(dynamic_cast<Constant * >(lphi));
                                bool same = false;
                                for(auto eachCC : p)
                                {
                                    if(*(eachCC->value_expr_)==*C)
                                    {
                                        eachCC->members_.emplace(&instr);
                                        same = true;
                                        break;
                                    }
                                }
                                if(!same)
                                {
                                    auto newCC =
createCongruenceClass(next_value_number_++);
                                    newCC->leader_ = lphi;
                                    newCC->value_expr_ =
ConstantExpression::create(dynamic_cast<Constant * >(lphi));
                                    newCC->value_phi_ = nullptr;
                                    newCC->members_.emplace(&instr);
                                    p.emplace(newCC);
                                    continue;
                                }

                            }
                        }
```

2. value expr

   这个函数只要实现几种指令的值表达式构建

   类型1：叶子结点

   如load/store/alloca/args/input，这一类都是没有任何指令与其相等的，只能与作为一个底层的叶子结点存在，仿照了constant等expr的设计

```
class LeafExpression : public Expression{
  public:
    static std::shared_ptr<LeafExpression>create(Value* operand)
    {
      return std::make_shared<LeafExpression>(operand);
    }
    virtual std::string print() { return "(Leaf: "+rhs->print()+")"; }
    LeafExpression(Value *operand)
        : Expression(e_leaf),rhs(operand) {}
    bool equiv(const LeafExpression* other) const
    {//???不知道对不对
      return rhs==other->rhs;
    }
```

```
    Value* getRhs() {return rhs; }
  private:
    Value* rhs;
```

类型2：比较值

如fcmp和cmp两种，也仿照了前面的设计填写的。

```
  class CmpExpression : public Expression{
    public:
      static std::shared_ptr<CmpExpression>create(CmpInst::CmpOp
op,std::shared_ptr<Expression>lhs,std::shared_ptr<Expression>rhs)
      {return std::make_shared<CmpExpression>(op,lhs,rhs); }
      CmpExpression(CmpInst::CmpOp
op,std::shared_ptr<Expression>lhs,std::shared_ptr<Expression>rhs):
                          Expression(e_cmp),lhs_(lhs),rhs_(rhs),op_(op){}
      virtual std::string  print()
      {
        return "(INT:"+lhs_->print()+" "+print_cmp_type(op_)+" "+rhs_-
>print()+")";
      }
      bool equiv(const CmpExpression* other) const
      {
        if((this->op_==other->op_)&&(this->lhs_==other->lhs_)&&(this-
>rhs_==other->rhs_))
        {
          return true;
        }
        return false;
      }
    private:
      CmpInst::CmpOp op_;
      std::shared_ptr<Expression>lhs_,rhs_;
  };
```

类型3：单值指令

如各种zext/sitofp等指令

```
class SingleExpression : public Expression
  {
    public:
      SingleExpression(Instruction::OpID
op,std::shared_ptr<Expression>lhs):Expression(e_single),op_(op),lhs_(lhs)
{}
      virtual std::string print()
      {
        return "("+Instruction::get_instr_op_name(op_)+":"+lhs_-
>print()+")";
      }
      static std::shared_ptr<SingleExpression>create(Instruction::OpID
op,std::shared_ptr<Expression>lhs)
      { return std::make_shared<SingleExpression>(op,lhs);}
      bool equiv(const SingleExpression* other) const
      {
        if(this->op_==other->op_ && this->lhs_==other->lhs_)
        {
          return true;
        }
        return false;
```

```
    }
  private:
    Instruction::OpID op_;
    std::shared_ptr<Expression>lhs_;
};
```

valueexpr按相关的指令生成expr

```
    if(instr->is_call()||instr->is_load()||instr->is_alloca()||instr-
>is_gep()||instr->is_store())
    {
        return LeafExpression::create(instr);
    }
    else if(instr->get_num_operand()==1)
    {

        //`zext,fptosi,sitofp`
        auto operand = instr->get_operand(0);
        if(dynamic_cast<Constant*>(instr->get_operand(0)))
        {
            return ConstantExpression::create(folder_-
>compute(instr,dynamic_cast<Constant*>(instr->get_operand(0))));
        }
        else
        {
            //从pin中找到值表达式
            for(auto i:pin)
            {
                if(i->members_.find(instr->get_operand(0))!=i-
>members_.end())
                {
                    //find value expression
                    return SingleExpression::create(instr-
>get_instr_type(),i->value_expr_);
                }
            }
            //return
VarExpression::create(valueExpr(static_cast<Instruction*>(instr-
>get_operand(0)),pin));
        }
    }
    if(instr->isBinary()||instr->is_cmp()||instr->is_fcmp())
    {
        auto lhs = instr->get_operand(0);
        auto rhs = instr->get_operand(1);
        bool lhs_c = false;
        bool rhs_c = false;
        std::shared_ptr<GVNExpression::Expression>lhs_ve=nullptr,rhs_ve =
nullptr;
        //是instruction,则为值表达式
        //是constant,则为常量

 std::shared_ptr<GVNExpression::ConstantExpression>l_const=nullptr,r_cons
t=nullptr;
        if(dynamic_cast<Constant*>(lhs)!=nullptr)
        {
            lhs_c = true;
            //lhs_i = false;
```

```
            l_const = ConstantExpression::create(dynamic_cast<Constant*>
(lhs));
        }
        else
        {
            for(auto i:pin)
            {
                if(i->members_.find(instr->get_operand(0))!=i-
>members_.end())
                {
                    lhs_ve = i->value_expr_;
                    break;
                }
            }
        }
        if(dynamic_cast<Constant*>(rhs))
        {
            rhs_c = true;
            //lhs_i = false;
            r_const = ConstantExpression::create(dynamic_cast<Constant*>
(rhs));
        }
        else
        {
            for(auto i:pin)
            {
                if(i->members_.find(instr->get_operand(1))!=i-
>members_.end())
                {
                    rhs_ve = i->value_expr_;
                    break;
                }
            }
        }
        //判断左右是否有常量
        if(instr->isBinary())
        {
            if(lhs_c&&rhs_c)
            {
                //左右都是常量
                auto res =folder_->compute(instr,dynamic_cast<Constant *>
(lhs),dynamic_cast<Constant * >(rhs));
                return ConstantExpression::create(res);
            }
            if(!lhs_c&&rhs_c)
            {
                //从pin中找到它的值表达式
                return BinaryExpression::create(instr-
>get_instr_type(),lhs_ve,r_const);
            }
            if(lhs_c&&!rhs_c)
            {
                return BinaryExpression::create(instr-
>get_instr_type(),l_const,rhs_ve);
            }
            if(!lhs_c&&!rhs_c)
            {
                return BinaryExpression::create(instr-
>get_instr_type(),lhs_ve,rhs_ve);
            }
        }
```

```
        if(instr->is_cmp())
        {
            if(lhs_c&&rhs_c)
            {
                auto res = folder_->compute(instr,dynamic_cast<Constant
*>(lhs),dynamic_cast<Constant * >(rhs));
                return ConstantExpression::create(res);
            }
            if(!lhs_c&&rhs_c)
            {
                return CmpExpression::create(dynamic_cast<CmpInst*>
(instr)->get_cmp_op(),lhs_ve,r_const);
            }
            if(lhs_c&&!rhs_c)
            {
                return CmpExpression::create(dynamic_cast<CmpInst*>
(instr)->get_cmp_op(),l_const,rhs_ve);
            }
            return CmpExpression::create(dynamic_cast<CmpInst*>(instr)-
>get_cmp_op(),lhs_ve,rhs_ve);
        }
        if(instr->is_fcmp())
        {
            if(lhs_c&&rhs_c)
            {
                auto res = folder_->compute(instr,dynamic_cast<Constant
*>(lhs),dynamic_cast<Constant * >(rhs));
                return ConstantExpression::create(res);
            }
            if(!lhs_c&&rhs_c)
            {
                return FcmpExpression::create(dynamic_cast<FCmpInst*>
(instr)->get_cmp_op(),lhs_ve,r_const);
            }
            if(lhs_c&&!rhs_c)
            {
                return FcmpExpression::create(dynamic_cast<FCmpInst*>
(instr)->get_cmp_op(),l_const,rhs_ve);
            }
            return FcmpExpression::create(dynamic_cast<FCmpInst*>(instr)-
>get_cmp_op(),lhs_ve,rhs_ve);
        }
    }
    if(instr->is_phi())
    {
        //遇到phi指令先跳过
        //不会遇到phi的

    }
    return {};
```

3. transferfunction

   这个函数基本上就是按照伪代码的来

```
for(auto cc_ptr:pout)
    {
        auto pos = (*cc_ptr).members_.find(x);
        if(pos!=(*cc_ptr).members_.end())
        {
            (*cc_ptr).members_.erase(x);
```

```
        }
    }//这是找到相同的x

auto ve = valueExpr(dynamic_cast<Instruction*>(x),pin);
    auto vpf = valuePhiFunc(ve,pin,curBB);
    bool findVe = false,findVpf = false;
    // 找到一样的ve/vpf就加到member
    for (auto cc_ptr : pout)
    {
        if(*(cc_ptr->value_expr_)==*ve)
        {
            //cc_ptr->members_.emplace(x);
            findVe = true;
            cc_ptr->members_.emplace(x);
            break;
        }
        if(vpf!=nullptr&&(cc_ptr->value_phi_!=nullptr))
        {
            if(*(cc_ptr->value_phi_)==*vpf)
            {
                findVpf = true;
                cc_ptr->members_.emplace(x);
                break;
            }
        }
    }
    // 没找到就创建新等价类
 auto cc = createCongruenceClass(next_value_number_++);
        cc->leader_ = x;
        cc->members_ = {x};
        cc->value_expr_ = ve;

        cc->value_phi_ = vpf;
        pout.insert(cc);
```

4. value phi func

   这个函数主要是按照伪代码来的，**值得注意的是，在intersect函数中，我将value expr 设置为了value phi**，不过这样会导致循环依赖

```
bool isVB = ve->get_expr_type()==GVNExpression::Expression::e_bin;
    auto VBptr = std::dynamic_pointer_cast<BinaryExpression>(ve);
    shared_ptr<Expression> lhs,rhs;
    if(isVB)
    {
        //std::shared_ptr<GVNExpression::BinaryExpression>
        lhs = VBptr->getVal(0);
        rhs = VBptr->getVal(1);
        //此处有问题
        //应该是查找左右的等价类，然后从等价类中找phi
        bool l1 = (lhs!=nullptr)&&lhs-
>get_expr_type()==Expression::e_phi;
        bool r1 = (rhs!=nullptr)&&rhs-
>get_expr_type()==Expression::e_phi;

        if(l1&&r1)
        {
            int count = 0;
            shared_ptr<Expression> V[2];
            for(auto i : curBB->get_pre_basic_blocks())
            {
```

```
                if(count==0)
                {
                    V[count] = getVN(pout_[i],
                            BinaryExpression::create(VBptr->getOp(),
std::dynamic_pointer_cast<PhiExpression>(lhs)->getLhs()
,std::dynamic_pointer_cast<PhiExpression>(rhs)->getLhs()));
                    if(V[count] == nullptr)
                    {
                        V[count] =
valuePhiFunc(BinaryExpression::create(VBptr->getOp(),
std::dynamic_pointer_cast<PhiExpression>(lhs)->getLhs()
,std::dynamic_pointer_cast<PhiExpression>(rhs)->getLhs()),pout_[i],i);

                    }
                }
                else
                {
                    V[count] = getVN(pout_[i],
                            BinaryExpression::create(VBptr->getOp(),
std::dynamic_pointer_cast<PhiExpression>(lhs)->getRhs()
,std::dynamic_pointer_cast<PhiExpression>(rhs)->getRhs()));
                    if(V[count] == nullptr)
                    {
                        V[count] =
valuePhiFunc(BinaryExpression::create(VBptr->getOp(),
std::dynamic_pointer_cast<PhiExpression>(lhs)->getRhs()
,std::dynamic_pointer_cast<PhiExpression>(rhs)->getRhs()),pout_[i],i);

                    }
                }
                count++;
            }
            if(V[0]!=nullptr&&V[1]!=nullptr)
            {
                return PhiExpression::create(V[0],V[1]);
            }
        }
    }
    return {};
```

5. getVN

   补充了一个返回值，即返回了VN

```
    for (auto it = pout.begin(); it != pout.end(); it++)
        if ((*it)->value_expr_ and *(*it)->value_expr_ == *ve)
            return (*it)->value_expr_;
    return nullptr;
```

6. join

   用顶元判断，如果只有一个元素且index=0，表明是顶元，直接返回另一个。

   是一个$O(mn)$的循环

```
partitions P = {};
    if(!P1.size()||!P2.size())
    {
        return {};
    }
```

```
    if(P1.size()==1)
    {
        //顶元，只有一个且其index=0
        for(auto i : P1)
        {
            if(i->index_==0)
            {
                return P2;
            }
        }
    }
    if(P2.size()==1)
    {
        //顶元，只有一个且其index=0
        for(auto i : P2)
        {
            if(i->index_==0)
            {
                return P1;
            }
        }
    }
    for(auto eachCC1 : P1)
    {
        for(auto eachCC2 :P2)
        {
            auto ck = intersect(eachCC1,eachCC2);
            if(ck!=nullptr)
            {
                P.insert(ck);
            }
        }
    }
    return P;
```

7. intersect

   代码解释见注释

```
auto Ck = createCongruenceClass();
    std::set_intersection(Ci->members_.begin(),Ci->members_.end(),
                          Cj->members_.begin(),Cj->members_.end(),
                          std::inserter(Ck->members_,Ck-
>members_.begin()));
    //auto ve = (Ci->value_expr_==Cj->value_expr_)?Ci-
>value_expr_:nullptr;
    //auto vpf = (Ci->value_phi_==Cj->value_phi_)?Ci->value_phi_:nullptr;
    if(!Ck->members_.empty()&&Ck->index_==0)
    {

        if(*Ci==*Ck)
        {
            //这种情况是两边都有相同的ve,这就表明:在两个前驱都存在相同的ve,就可以把
ve传递下去，leader域可能存在取交集导致原有leader失效
            //因此leader域从member中取,取第一个即可
            //vpf只保留相同的
            Ck->leader_ = *(Ck->members_.begin());
            Ck->value_expr_ = Ci->value_expr_;
            Ck->value_phi_ = Ci->value_phi_;
            Ck->index_ = Ci->index_;
        }
```

```
        else
        {
            //这种情况是两边的ve是不一样的，这时候对应伪代码的无value number
            //value expr在此处是不急着生成的,在后面分析到phi指令时,用instr补上一个
叶子结点即可
            Ck->leader_ = *(Ck->members_.begin());
            Ck->index_ = next_value_number_++;
            Ck->value_phi_ = PhiExpression::create(Ci->value_expr_,Cj-
>value_expr_);
            Ck->value_expr_ = Ck->value_phi_;//后面检测时再加value expr(以叶
子结点形式)
        }
        return Ck;
    }

    return nullptr;
```

## 思考题

1. 请简要分析你的算法复杂度

   设程序中有n个值表达式，每个分区至多有$O(n)$个等价类，每个等价类至多有$O(v)$的大小，v是程序中常数和变量的总量。每一次 `intersect` 则需之多$O(v)$的时间。故一次join复杂度为$O(n^2v)$。

   设有j个join的点，故有$O(n^2 \times v \times j)$的复杂度花费在join上

   一个valueexpr计算的时间为$O(n)$，而一个phi计算的时间为$O(n * j)$，(这是因为至多向上搜索j次就能终止递归)。最后一条语句所需时间为$O(n * j)$，

   有n条语句，故迭代一次(考虑了join)最多要$O(n^2 * v * j)$，最多迭代n次，因此总的复杂度为$O(n^3 * v * j)$

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?

   存在循环引用的，因为我把value expr设置为了phi，而phi之后又会关联其他的expr，其他的expr会关联value expr，这样会形成循环引用(但是最后几天才发现的，来不及重构了)

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

   1. 底层设计有问题，需要重构，因为存在了循环引用导致phi的叠加无法处理，但是因为时间不够了来不及重构了。

# 实验总结

1. 极大地提高了自己的写代码能力，学会了不少现代c++的特性和风格
2. 学会了复现论文的算法
3. 第一次完整地做一个项目

# 实验反馈（可选 不会评分）

希望能给个更加确定的方向，不然做错了真的难顶。