lab4

PB21051012 刘祥辉

实现方法

支配树的构建：

- 得到基本块的idom：

  首先后序遍历除entry block的所有节点，使用list类型结构存储：`std::list<BasicBlock*> bb_list;`,再使用`bb_list.reverse()`倒置得到逆后序遍历的顺序。
  `*BasicBlock** *Dominators*::first_processed_pre_bb(*BasicBlock** *bb*);`函数寻找正在处理的节点中父节点已经被处理的节点，并返回首个即可。
  `void *Dominators*::bb_num_init(*BasicBlock** *bb*,std::map<*BasicBlock**,int> &*bb_num*,int *num*,std::map<*BasicBlock**,bool>& *is_visit*);`该函数记录广度优先遍历的顺序，以方便后续寻找两个节点最近的祖先结点。
  `*BasicBlock** *Dominators*::intersect(*BasicBlock** *b1*, *BasicBlock** *b2*,std::map<*BasicBlock**,int> *bb_num*);`该函数用于更新节点找到最近的父节点，该函数调用了
  `void *Dominators*::renew_list(std::list<*BasicBlock**> &*pre_bb*);`这个函数功能是更新队列实现前向搜索最近的祖先。

```
void Dominators::create_idom(Function *f) {
    // TODO 分析得到 f 中各个基本块的 idom
    BasicBlock* new_idom;
    idom_[f->get_entry_block()] = f->get_entry_block();
    bool changed = true;
    std::map<BasicBlock*,bool> is_visit{};//辅助键对，是否被遍历过
    std::map<BasicBlock*,int> bb_num{};
    std::list<BasicBlock*>  bb_list;  //链表存储后序遍历的顺序,头部为entry_block
    bb_list.clear();
    is_visit.clear();
    bb_num.clear();
    for(auto &bb1 : f->get_basic_blocks()){
        auto bb = &bb1;
        if(bb == f->get_entry_block())
            is_visit[bb] = true;
        else
            is_visit[bb] = false;
    }
    bb_dfs(bb_list,f->get_entry_block(),is_visit);
    bb_list.push_back(f->get_entry_block());
    bb_list.reverse(); //存储逆置后续访问的顺序
    for(auto &bb1 : f->get_basic_blocks()){
        auto bb = &bb1;
        is_visit[bb] = false;
    }
    bb_num_init(f->get_entry_block(),bb_num,1,is_visit);
    while(changed){
        changed = false;
        for (auto element : bb_list) {
            if(element == f->get_entry_block()) continue;
```

```
            else{
                new_idom = first_processed_pre_bb(element);
                for (auto pred : element->get_pre_basic_blocks()) {
                    if (idom_[pred] != nullptr) {
                        new_idom = intersect(pred, new_idom,bb_num);
                    }
                }
                if (idom_[element] != new_idom) {
                    idom_[element] = new_idom;
                    changed = true;
                }

            }
        }
    }
}


void Dominators::create_dominance_frontier(Function* f) {
    // TODO 分析得到 f 中各个基本块的支配边界集合
    BasicBlock* runner = nullptr;
    for (auto it = f->get_basic_blocks().begin();it != f-
>get_basic_blocks().end();it++) {
        if ((&*it)->get_pre_basic_blocks().size() >= 2) {
            for (auto pred : (&*it)->get_pre_basic_blocks()) {
                runner = pred;
                while (runner != idom_[&*it]) {
                    dom_frontier_[runner].insert(&*it);
                    runner = idom_[runner];
                }
            }
        }
    }
}



void Dominators::create_dom_tree_succ(Function* f) {
    // TODO 分析得到 f 中各个基本块的支配树后继
    for (auto it = f->get_basic_blocks().begin();it != f-
>get_basic_blocks().end();it++) {
        if (&*it != f->get_entry_block())
            dom_tree_succ_blocks_[idom_[&*it]].insert(&*it);
    }
}

void Dominators::bb_dfs(std::list<BasicBlock*>& bb_list,
            BasicBlock* bb,
            std::map<BasicBlock*,bool> &is_visit){
    //后序遍历
    for(auto &it: bb->get_succ_basic_blocks()){
        if(!is_visit[it]){
            //没有遍历过
            is_visit[it] = true;
            bb_dfs(bb_list,it,is_visit);
            bb_list.push_back(it);
        }
    }
```

```
}

BasicBlock* Dominators::first_processed_pre_bb(BasicBlock* bb){
    //不必为第一个，随机一个即可
    for (auto &preb : bb->get_pre_basic_blocks()) {
        if (idom_[preb] != nullptr) {
            return preb;
        }
    }
}


void Dominators::bb_num_init(BasicBlock* bb,
                             std::map<BasicBlock*,int> &bb_num,int num,
                             std::map<BasicBlock*,bool>& is_visit){
    for(auto &succ_bb :bb->get_succ_basic_blocks()){
        if(!is_visit[succ_bb]){
            bb_num[succ_bb] = num;
            is_visit[succ_bb] = true;
            bb_num_init(succ_bb,bb_num,num+1,is_visit);
        }
    }
}


void Dominators::renew_list(std::list<BasicBlock*> &pre_bb){
    int num = pre_bb.size();
    while(num>0){
        BasicBlock* front = pre_bb.front();
        pre_bb.pop_front();
        for(auto &pre: front->get_pre_basic_blocks()){
            pre_bb.push_back(pre);
        }
        num -= 1;
    }
}

BasicBlock* Dominators::intersect(BasicBlock* b1, BasicBlock* b2,
                                  std::map<BasicBlock*,int> bb_num) {
    BasicBlock* finger1 = b1;
    BasicBlock* finger2 = b2;
    std::list<BasicBlock*> pre_b1;
    std::list<BasicBlock*> pre_b2;
    int seq_b1,seq_b2;
    pre_b1.push_back(finger1);
    pre_b2.push_back(finger2);
    seq_b1 = bb_num[pre_b1.front()];
    seq_b2 = bb_num[pre_b2.front()];
    while(seq_b1!=seq_b2){
        if(seq_b1 < seq_b2){
            renew_list(pre_b2);
        }
        else{
            renew_list(pre_b1);
        }
        seq_b1 = bb_num[pre_b1.front()];
        seq_b2 = bb_num[pre_b2.front()];
    }
```

```
    bool flag = true;
    std::list<BasicBlock*> same_bb;
    while(flag){
        same_bb.clear();
        std::set_intersection(
            pre_b1.begin(), pre_b1.end(),
            pre_b2.begin(), pre_b2.end(),
            std::back_inserter(same_bb)
        );
        if(!same_bb.empty())
            flag = false;
        else{
            renew_list(pre_b1);
            renew_list(pre_b2);
        }
    }
    return same_bb.front();
}
```

插入phi函数和rename

`void Mem2Reg::generate_phi();`函数按照论文中算法写即可。

`void Mem2Reg::rename(BasicBlock *bb);`中`origin_stack = variable_stacks;`保存栈，方便函数结束时进行恢复。

最后进行删除时删除相应的`StoreInst*`类型指令即可，注意不要边运行变删除，要在统计`StoreInst*`指令位置后再统一删除，否则会报错。

```
void Mem2Reg::generate_phi() {
    // TODO
    // 步骤一：找到活跃在多个 block 的全局名字集合，以及它们所属的 bb 块
    // 步骤二：从支配树获取支配边界信息，并在对应位置插入 phi 指令
    using BBSet = std::set<BasicBlock *>;
    BasicBlock* X;
    for(auto &bb: func_->get_basic_blocks()){
        BasicBlock* bb_ = &bb;
        for (auto &instr : bb_->get_instructions()) {
            if (instr.is_alloca() && !dynamic_cast<AllocaInst*>(&instr)-
>get_alloca_type()->is_array_type()) {
                Variable_bbset[&instr].insert(bb_);
                used_Variable.insert(&instr);
            }
            if(instr.is_store()){
                auto res = used_Variable.find(dynamic_cast<StoreInst*>(&instr)-
>get_lval());
                if(res !=used_Variable.end()){
                    Defs[*res].insert(bb_);
                }
            }
        }
    }
    for(auto variable: used_Variable){
        BBSet  F;
        BBSet  W;
        F.clear();
        W.clear();
        for(auto bb: Defs[variable]){
```

```
            W.insert(bb);
        }
        while(!W.empty()){
            X = *W.begin();
            W.erase(W.begin());
            for(auto Y: dominators_->get_dominance_frontier(X)){
                auto res = F.find(Y);
                if(res == F.end()){
                    Instruction* temp = PhiInst::create_phi(variable-
>get_type()->get_pointer_element_type(), Y);
                    Y->add_instr_begin(temp);
                    phi_to_variable[temp] = variable;
                    F.insert(Y);
                    bool in_or_not = false;
                    for (auto element : Defs[variable]) {
                        if (element == Y) {
                            in_or_not = true;
                            break;
                        }
                    }
                    if (!in_or_not) {
                        W.insert(Y);
                    }
                }
            }
        }
    }

}

void Mem2Reg::rename(BasicBlock *bb) {
    // TODO
    // 步骤三：将 phi 指令作为 lval 的最新定值，lval 即是为局部变量 alloca 出的地址空间
    // 步骤四：用 lval 最新的定值替代对应的load指令
    // 步骤五：将 store 指令的 rval，也即被存入内存的值，作为 lval 的最新定值
    // 步骤六：为 lval 对应的 phi 指令参数补充完整
    // 步骤七：对 bb 在支配树上的所有后继节点，递归执行 re_name 操作
    // 步骤八：pop出 lval 的最新定值
    // 步骤九：清除冗余的指令
    std::map<Value*,std::stack<Value*>> origin_stack;
    origin_stack = variable_stacks;
    for(auto &instr: bb->get_instructions()){
        Instruction* instr_ = &instr;
        if(instr_->is_phi()){
            variable_stacks[phi_to_variable[instr_]].push(instr_);
        }
        if(instr_->is_store()) {
            auto res = used_Variable.find(dynamic_cast<StoreInst*>(instr_)-
>get_lval());
            if(res != used_Variable.end()){
                variable_stacks[dynamic_cast<StoreInst*>(instr_)-
>get_lval()].push(dynamic_cast<StoreInst*>(instr_)->get_rval());
            }
        }
        if(instr_->is_load()) {
            auto res = used_Variable.find(dynamic_cast<LoadInst*>(instr_)-
>get_lval());
```

```
            if(res != used_Variable.end() &&
!variable_stacks[dynamic_cast<LoadInst*>(instr_)->get_lval()].empty()){
                instr_-
>replace_all_use_with(variable_stacks[dynamic_cast<LoadInst*>(instr_)-
>get_lval()].top());
            }
        }
    }
    for(auto &suc_bb: bb->get_succ_basic_blocks()){
        for (auto& instr : suc_bb->get_instructions()) {
            Instruction* instr_ = &instr;
            if(instr_->is_phi() &&
variable_stacks[phi_to_variable[instr_]].size() != 0) {
                dynamic_cast<PhiInst*>(instr_)-
>add_phi_pair_operand(variable_stacks[phi_to_variable[instr_]].top(), bb);
            }
        }
    }
    for (auto &succ_dom : dominators_->get_dom_tree_succ_blocks(bb)) {
        rename(succ_dom);
    }
    std::set<Instruction*>to_be_delete_ins;
    for (auto & instr : bb->get_instructions()) {
        Instruction* instr_ = &instr;
        if(instr_->is_store()){
            auto res = used_Variable.find(dynamic_cast<StoreInst*>(instr_)-
>get_lval());
            if (res !=used_Variable.end()) {
                // bb->get_instructions().erase(&instr);  //在这删除会报错！！！
                to_be_delete_ins.insert(&instr);
            }
        }
    }
    for (auto instr : to_be_delete_ins) {
        bb->get_instructions().erase(instr);
    }
    variable_stacks = origin_stack;
}
```

正确性验证

`functional-cases:`

```
fcmwf@LAPTOP-126PVBV4:~/2023ustc-jianmu-compiler/tests/4-me
m2reg$ ./eval_lab4.sh functional-cases test
[info] Start testing, using testcase dir: functional-cases
0-io...OK
1-return...OK
2-calculate...OK
3-output...OK
4-if...OK
5-while...OK
6-array...OK
7-function...OK
8-store...OK
9-fibonacci...OK
10-float...OK
11-floatcall...OK
12-global...OK
13-complex...OK
```

`testcases:`

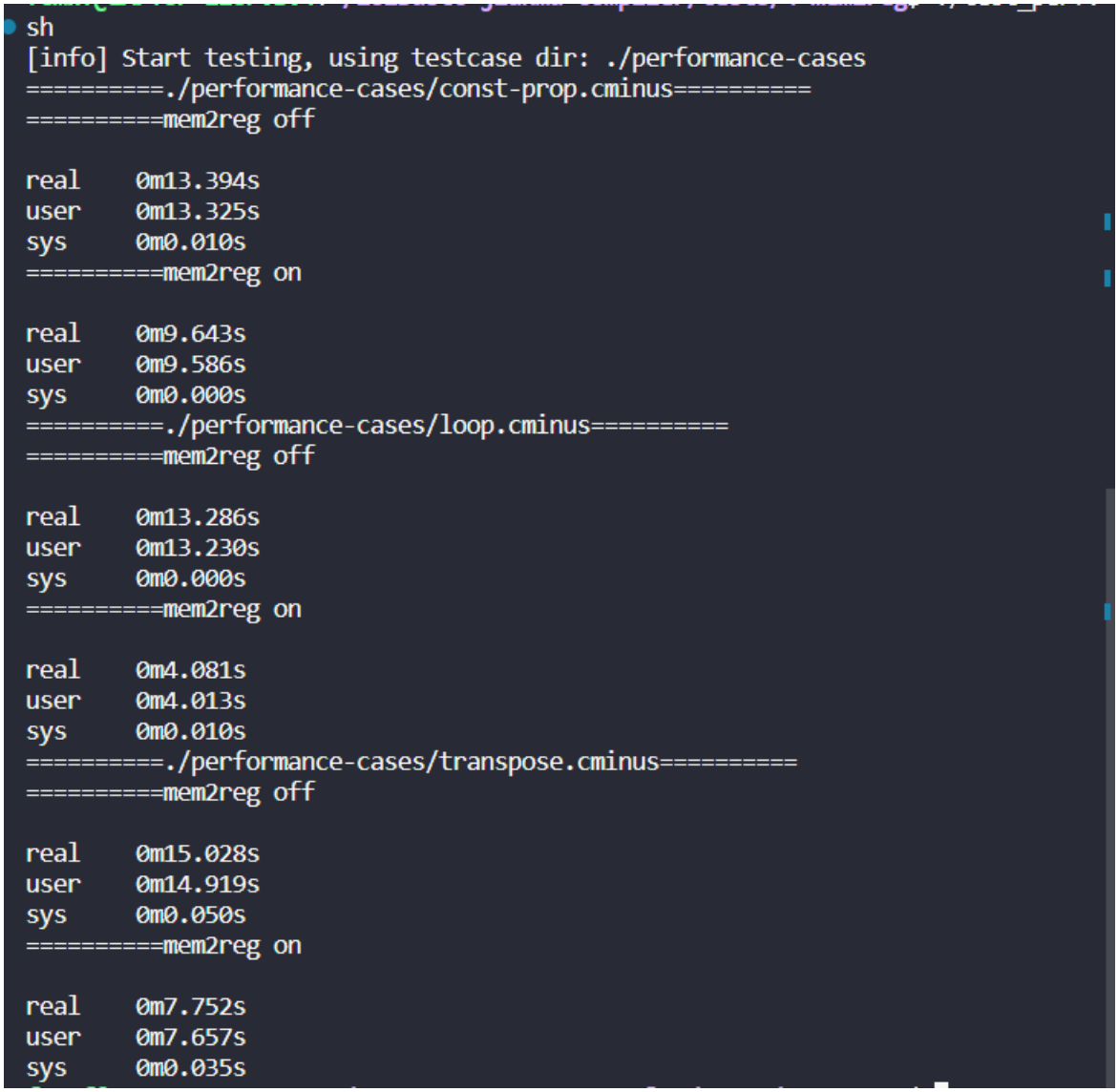
```
fcmwf@LAPTOP-126PVBV4:~/2023ustc-jianmu-compiler/tests/4-me
m2reg$ ./eval_lab4.sh testcases test
[info] Start testing, using testcase dir: testcases
0-io...OK
1-return...OK
2-calculate...OK
3-output...OK
4-if...OK
5-while...OK
6-array...OK
7-function...OK
8-store...OK
9-fibonacci...OK
10-float...OK
11-floatcall...OK
12-global...OK
13-complex...OK
```

`testcases-general:`

```
fcmwf@LAPTOP-126PVBV4:~/2023ustc-jianmu-compiler/tests/4-me
m2reg$ ./eval_lab4.sh testcases_general test
[info] Start testing, using testcase dir: testcases_general
1-return...OK
2-decl_int...OK
3-decl_float...OK
4-decl_int_array...OK
5-decl_float_array...OK
6-num_add_int...OK
7-assign_int_var_local...OK
8-assign_int_array_local...OK
9-assign_cast...OK
10-funcall...OK
11-funcall_chain...OK
12-funcall_recursion...OK
13-if_stmt...OK
14-while_stmt...OK
15-if_while...OK
16-if_chain...OK
17-while_chain...OK
18-global_var...OK
19-global_local_var...OK
20-gcd_array...OK
21-comment...OK
```

均已通过

性能验证等


sh
[info] Start testing, using testcase dir: ./performance-cases
==========./performance-cases/const-prop.cminus==========
==========mem2reg off
real 0m13.394s
user 0m13.325s
sys 0m0.010s
==========mem2reg on
real 0m9.643s
user 0m9.586s
sys 0m0.000s
==========./performance-cases/loop.cminus==========
==========mem2reg off
real 0m13.286s
user 0m13.230s
sys 0m0.000s
==========mem2reg on
real 0m4.081s
user 0m4.013s
sys 0m0.010s
==========./performance-cases/transpose.cminus==========
==========mem2reg off
real 0m15.028s
user 0m14.919s
sys 0m0.050s
==========mem2reg on
real 0m7.752s
user 0m7.657s
sys 0m0.035s

性能有很大的提高