1. 编译原理实验－中间代码优化

1.1. 实验内容

在之前实验词法分析，语法分析，语义分析和中间代码生成的基础上使用数据流分析算法等消除效率低下和无法被轻易转化为机器代码的中间代码，从而将 C–源代码翻译成的中间代码转化为语义等价但是更加简洁高效的版本。要求在优化代码的同时保正新生成的代码和原有代码的语义相同。

1.2. 核心优化部分

1.2.1. 无用代码消除

使用活跃变量分析结果，标记dead变量，对相应的死代码进行消除。

```
bool updated = false;
Set_IR_var *blk_out_fact = VCALL(*t, getOutFact, blk);
Set_IR_var *new_out_fact = LiveVariableAnalysis_newInitialFact(t);
LiveVariableAnalysis_meetInto(t, blk_out_fact, new_out_fact);
rfor_list(IR_stmt_ptr, i, blk->stmts) {
    IR_stmt *stmt = i->val;
    if(stmt->stmt_type == IR_OP_STMT || stmt->stmt_type == IR_ASSIGN_STMT) {
        IR_var def = VCALL(*stmt, get_def);
        if(def == IR_VAR_NONE) continue;
        if(VCALL(*new_out_fact, exist, def) == false) {
             stmt->dead = true;
             updated = true;
        }
    }
    LiveVariableAnalysis_transferStmt(t, stmt, new_out_fact);
}
DELETE(new_out_fact);
remove_dead_stmt(blk); // 删除标记为 dead 的变量赋值，完成死代码消除工作
return updated;
```

1.2.2. 公共表达式消除

使用可用表达式分析的结果，对公共表达式进行消除。

```
Fact_set_var *blk_in_fact = VCALL(*t, getInFact, blk);
Fact_set_var *new_in_fact = AvailableExpressionsAnalysis_newInitialFact(t);
AvailableExpressionsAnalysis_meetInto(t, blk_in_fact, new_in_fact);
for_list(IR_stmt_ptr, i, blk->stmts) {
    IR_stmt *stmt = i->val;
    if(stmt->stmt_type == IR_OP_STMT) {
        IR_op_stmt *op_stmt = (IR_op_stmt*)stmt;
        IR_var expr_var = op_stmt->rd;
        if(VCALL(new_in_fact->set, exist, expr_var)) // available
            stmt->dead = true;
    }
    AvailableExpressionsAnalysis_transferStmt(t, stmt, new_in_fact);
```

```c
}
RDELETE(Fact_set_var, new_in_fact);
remove_dead_stmt(blk); // 删除标记为 dead 的可用表达式赋值
```

1.2.3. 常量折叠

使用常量传播分析的结果，识别常量并进行替换。

```c
Map_IR_var_CPValue *blk_in_fact = VCALL(*t, getInFact, blk);
Map_IR_var_CPValue *new_in_fact = ConstantPropagation_newInitialFact(t);
ConstantPropagation_meetInto(t, blk_in_fact, new_in_fact);
for_list(IR_stmt_ptr, i, blk->stmts) {
    IR_stmt *stmt = i->val;
    IR_use use = VCALL(*stmt, get_use_vec);
    for(int j = 0; j < use.use_cnt; j++)
        if(!use.use_vec[j].is_const) {
            IR_var use_var = use.use_vec[j].var;
            CPValue use_CPVal = Fact_get_value_from_IR_var(new_in_fact, use_var);
            if(use_CPVal.kind == CONST)
                use.use_vec[j] = (IR_val){.is_const = true, .const_val = use_CPVal.const_
        }
    ConstantPropagation_transferStmt(t, stmt, new_in_fact);
}
RDELETE(Map_IR_var_CPValue, new_in_fact);
```

1.2.4. 复制传播

```c
Fact_def_use *blk_in_fact = VCALL(*t, getInFact, blk);
Fact_def_use *new_in_fact = CopyPropagation_newInitialFact(t);
CopyPropagation_meetInto(t, blk_in_fact, new_in_fact);
for_list(IR_stmt_ptr, i, blk->stmts) {
    IR_stmt *stmt = i->val;
    IR_use use = VCALL(*stmt, get_use_vec);
    for(int j = 0; j < use.use_cnt; j++)
        if(!use.use_vec[j].is_const) {
            IR_var use_var = use.use_vec[j].var;
            if(VCALL(new_in_fact->def_to_use, exist, use_var))
                use.use_vec[j].var = VCALL(new_in_fact->def_to_use, get, use_var);
        }
    CopyPropagation_transferStmt(t, stmt, new_in_fact);
}
RDELETE(Fact_def_use, new_in_fact);
```

1.3. 数据流分析

在提供的框架代码的基础上完善数据流分析的代码，以支持后续代码优化。根据下面方法对框架代码进行补充。

1.3.1. 常量传播

- 方向：forward
- 边界：Out[Entry] = NAC

- 初始化：In[blk] = NAC; Out[blk] = NAC;
- meet：In[blk] = union(all Out[pred]) // pred 为 blk 的前驱
- transfer：Out[blk] = union(gen[blk], (In[blk] - kill[blk]))

1.3.2. 可用表达分析

- 方向：forward
- 边界：Out[Entry] = empty
- 初始化：In[blk] = Uni(全集)，Out[blk] = Uni(全集)
- meet：In[blk] = intersection(all Out[pred])
- transfer：Out[blk] = union(gen[blk], (In[blk] - kill[blk]))

1.3.3. 活跃变量分析

- 方向：backward
- 边界：In[Exit] = empty
- 初始化：In[blk] = empty，Out[blk] = empty
- meet：Out[blk] = union(all In[succ])
- transfer：In[blk] = union(gen[blk], (Out[blk] - kill[blk]))

1.3.4. 复制传播

- 方向：forward
- 边界：Out[Entry] = Uni
- 初始化：In[blk] = empty, Out[blk] = empty
- meet：In[blk] = intersection(all Out[pred])
- transfer：

```
if(VCALL(fact->def_to_use, exist, new_def)) {
    IR_var use = VCALL(fact->def_to_use, get, new_def);
    VCALL(fact->def_to_use, delete, new_def);
    VCALL(fact->use_to_def, delete, use);
}
if(VCALL(fact->use_to_def, exist, new_def)) {
    IR_var def = VCALL(fact->use_to_def, get, new_def);
    VCALL(fact->use_to_def, delete, new_def);
    VCALL(fact->def_to_use, delete, def);
}
VCALL(fact->def_to_use, set, def, use);
VCALL(fact->use_to_def, set, use, def);
```

1.4. 总结

本次实验主要的难点在于根据数据流分析算法设计，读懂框架代码，完成算法的代码实现。之后使用数据流分析结果进行代码优化的部分实现就非常灵活，可以使用多种优化手段。我的代码只使用了死代码消除优化，但也取得了不错的效果。若想继续提高优化效率，可以尝试其他全局优化和循环不变代码外提。