编译原理lab5：中间代码优化

221220070 刁涟承

一、功能实现

- 在词法分析、语法分析、语义分析和中间代码生成程序的基础上，使用数据流分析等算法消除冗余；
- 实现可用表达式分析
- 实现常量传播分析
- 实现复制传播分析
- 实现活跃变量分析
- 实现迭代和工作表(`worklist`)驱动的后向分析求解器

二、实现思路

可用表达式分析

可用表达式分析为Forward的Must分析：

- 在`newBoundaryFact`&`newInitialFact`中完成初始化：将边界Fact初始化为空集，内部节点的Fact初始化为全集；
- 在函数`meetInto`中实现meet操作，由于为Must分析，因此采取交集操作。

常量传播分析

首先根据文档介绍的格理论，在`meetValue`函数中实现对不同值的meet处理，分为不同情况：

1. 如果二者有一个为NAC，那么meet得到的值也是NAC；
2. 如果二者有一个为UNDEF，那么返回另一个值（另一个值为UNDEF也成立）
3. 如果二者都为常数，相等则返回原值，不等则meet得到NAC。

处理`calculateValue`函数：

1. 如果两个变量都为常数，返回计算的结果；
2. 如果两个变量其中有一个UNDEF，返回UNDEF；
3. 如果两个变量其中有一个NAC，返回NAC；
4. 检查除0操作，这种情况返回UNDEF

常量传播分析为Forward的May分析：

- 初始化函数中，在处理边界(Entry)时，函数参数初始化为NAC；
- 在`transferStmt`中进行常量值的更新：如果为赋值语句，直接更新（`update_value`）为右值；如果为计算语句，将右边结果计算好后更新；其他情况更新为NAC；

复制传播分析

复制传播分析为Backward的Must分析：

- 在`newBoundaryFact`&`newInitialFact`中完成初始化：将边界Fact初始化为空集，内部节点的Fact初始化为全集；
- 在`transferStmt`中处理use-to-def链和def-to-use链的删除和添加：
  - 在每出现新的复制语句时，在`use_to_def`和`def_to_use`中分别删除对应的无效映射
  - 出现新的复制语句时，在`use_to_def`和`def_to_use`中添加新的`use`和`def`的映射

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

活跃变量分析

活跃变量分析为Backward的May分析：

- 在`meetInto`中实现meet操作，由于为May分析，因此采取并集操作。
- 在`transferStmt`中实现基本块中的状态转移方程 $use_B U(OUT[B] - def_B)$，先在`fact`中减去基本块中新定义的变量`def`，再逐个插入使用变量`use`：

```
// OUT[B] - def_B
if(def != IR_VAR_NONE) {
    VCALL(*fact, delete, def);
}
// use_B U (OUT[B] - def_B)
for(unsigned i = 0; i < use.use_cnt; i++) {
    IR_val use_val = use.use_vec[i];
    if(!use_val.is_const) {
        IR_var use = use_val.var;
        VCALL(*fact, insert, use);
    }
}
```

- 如果检测到变量在语句的出口不活跃（`! VCALL(*new_out_fact, exist, def)`），那么可以标记为死代码

实现后向分析的求解器

在`solver.c`里仿照前向分析的算法框架，实现后向分析求解器的算法框架。

1. 在`initializeBackward`函数中实现初始化，遍历每个基本块，调用不同分析的初始化函数进行IN和OUT的初始化。注意后向分析中Exit为Boundary，需要特殊处理；

2. 在`iterativeDoSolveBackward`函数中实现迭代式分析算法，遍历每个基本块，获取IN和OUT，调用相应函数进行meet操作和状态转移。注意后向分析中，meet操作为$OUT[blk] = meetAll(IN[succ]\ for\ succ \in Allsucc[blk])$，与前向分析相反。
3. 在`worklistDoSolveBackward`函数中实现worklist式分析算法，将每个基本块加入处理队列并进行处理。meet操作仍为$OUT[blk] = meetAll(IN[succ]\ for\ succ \in Allsucc[blk])$。注意，在后向分析中，若IN[blk]发生update，则将其前驱全部加入 worklist，与前向分析相反。

三、编译环境

操作系统：GNU Linux Release: Ubuntu 22.04 LTS, Kernel version 5.15.153.1-2；
编译器：GCC version 7.5.0；
词法分析工具：GNU Flex version 2.6.4；
语法分析工具：GNU Bison version 3.0.4。

四、文件结构

```
221220070.zip
├── Code
│   ├── compile_commands.json
│   ├── include
│   │   ├── config.h
│   │   ├── container
│   │   │   ├── list.h
│   │   │   ├── treap.h
│   │   │   └── vector.h
│   │   ├── IR.h
│   │   ├── macro.h
│   │   └── object.h
│   ├── makefile
│   ├── README.md
│   └── src
│       ├── container
│       │   └── treap.c
│       ├── IR
│       │   ├── IR.c
│       │   ├── IR_display.c
│       │   ├── IR_function_append_sentence.c
│       │   ├── IR_function_build_graph.c
│       │   ├── IR_stmt.c
│       │   └── IR_symbol_allocator.c
│       ├── IR_optimize
│       │   ├── available_expressions_analysis.c
│       │   ├── constant_propagation.c
│       │   ├── copy_propagation.c
│       │   ├── include
│       │   │   ├── available_expressions_analysis.h
│       │   │   ├── constant_propagation.h
│       │   │   ├── copy_propagation.h
│       │   │   ├── dataflow_analysis.h
│       │   │   ├── IR_optimize.h
```

```
│   │   │   └── live_variable_analysis.h
│   │   ├── IR_optimize.c
│   │   ├── live_variable_analysis.c
│   │   └── solver.c
│   ├── IR_parse
│   │   ├── include
│   │   │   └── IR_parse.h
│   │   ├── IR_lexical.l
│   │   ├── IR_parse.c
│   │   └── IR_syntax.y
│   └── main.c
├── report.pdf
```