Mini optimizer document

前言

本项目为《程序设计语言与编译》(余盛季老师，2023)课程设计，完成了面向mini语言编译器工具链的优化器设计与实现。

项目完成的优化器包括：

1. 常量折叠优化
2. 复制传播优化
3. 死代码优化
4. 无效赋值语句优化
5. 公共子表达式优化
6. 循环优化

优化器运行方式：

```
mini filename -o
```

项目成员：江世昊，罗向阳，李涤非。

> PS: 项目可能存在代码实现冗余/存在bug/注释难以理解等等问题，托付给更加优秀的学弟学妹们进一步完善:)

项目整体设计思路

mini编译器通过yyparse完成语法分析，生成存储三地址码的链表。mini优化器的优化在TAC级别进行，因此应在yyparse后调用优化器API对当前的全局TAC进行优化。我们设计的优化器顶层API为tac_optimizer，该函数实现了优化的主要逻辑，会针对针对当前的TAC循环执行一系列的子优化器，完成各类优化。顶层API的调用如图：

```
yyparse();

tac_dump();

if(opt_flag){
    tac_optimizer();
    /* printf(" --------------- After Optimization: ---------------\n");  */
    /* tac_dump(); */
}
```

该函数在构建TAC对应的CFG（）之后即进入优化循环：

```
// All things done, now we begin to do optimize!
int opt_cnt;
int round = 0;
do{
    opt_cnt = 0;
    // 简单优化
    // X = X
    opt_cnt += do_opt.simple_opt(tac_first);
    // 常量传播+常量折叠优化
    opt_cnt += do_opt.const_opt(BB_array, BB_num, tac_first);
    // 全局和局部复制传播优化
    opt_cnt += do_opt.copy_propagaton_opt(BB_array, BB_num, tac_first);
    // 死代码消除
    opt_cnt += do_opt.deadcode_opt(BB_array,BB_num,tac_first);
    // deadassign消除
    opt_cnt += do_opt.deadvar_opt(BB_array,BB_num,tac_first);
    // 公共子表达式消除
    opt_cnt += do_opt.local_comsub_opt(BB_array, BB_num);

    opt_cnt += do_opt.global_comsub_opt(BB_array,BB_num,tac_first);

    round ++;
}while(opt_cnt > 0 && round <= MAX_ROUND);
```

在优化循环中会反复按顺序执行各个优化器（在这里被封装为一个函数，通过函数指针调用），同时统计优化器执行后优化掉TAC的条数，直到达到执行次数上限或已经不再有优化空间。

上述内容即为mini优化器设计的整体思路。然而为了完成具体的优化任务，需要解决的关键问题包括：

1. 设计什么样的数据结构存储CFG?
2. 为了实现优化，我们需要进行一系列的静态分析获得关键信息，如何实现数据流分析？如何设计数据结构存储数据流分析结果?
3. 如何基于数据流分析结果实现各个子优化器?

CFG设计

CFG对于静态分析、编译优化而言至关重要。

本项目存储CFG的数据结构设计为一个verctor，其中的每一项为一个BB。全局变量BB_array存储了当前的CFG。一个BB的内容如下：

```
typedef struct basic_block{
    // in and out TAC pointer of a BB
    TAC *in;
    TAC *out;
    // ID start from 0
    int id;
    // use Adjacency List to store the CFG, this is the pointer to successors
    A_node * suc;
    // prev BBs
    A_node * prev;

    //live sign
    int live;
}BB;
```

其中in和out指针指向了对应的TAC链表中该BB的入口和出口指令。suc和prev指针分别指向了存储后继BB和前驱BB的邻接表项。邻接表的每一项代表一个BB，用BB的id进行索引。

通过TAC构建CFG的函数为build_CFG，CFG的构建方法参考这里，代码实现较为简单不再赘述。

数据流分析API实现

数据流分析(Data Flow Analysis)是实现编译优化的基础。本项目实现了四种基本的数据流分析API，用于为优化器提供程序的静态特征。数据流分析的相关实现位于DFA.c文件中。

本项目的数据流分析原理部分请参考这里，在查看代码实现前强烈推荐先学习该课程。

数据流分析的结果使用数据结构R_node进行存储。R_node会使用bit vector存储每一个基本快的当前IN和OUT状态。值得注意的是，bit vector的长度以及每一位的含义在不同的数据流分析中都有所不同，请在不同的数据流分析API中进行具体分析。完成分析后，一个R_node对应一个基本块：

```
// output of DFA: each node representing the IN and OUT state of a BB
// one node <--> one BB
typedef struct res_node
{
    // n index the BB
    int n;
    // number of elems in vector
    int count;
    // IN and OUT vector for DFA
    __int8_t * in_vector;
    __int8_t * out_vector;

    // this is used ONLY for constant propagation!
    int * constant_status;
    int * status_in;

}R_node;
```

注意，虽然众多数据流分析是在BB级别进行，但是为了实现指令级别的优化，本项目基本实现了在一条TAC指令级别进行的分析。

1. Reaching definition分析

Reaching definition是用来分析变量从程序中的一点p的定义是否可以到达程序中的另一点q，复制传播优化器即使用了该数据流分析的结果。该数据流分析是前向/may类型的分析，bit vector中的每一位代表一个definition。

具体而言，该数据流分析实现与函数Reaching_definition中。首先，该函数遍历搜索TAC中的definition语句，初始化相关数据结构如R_node。之后，该函数会使用Working List算法循环对一个基本快进行分析。所以实际上对一个基本快进行分析的功能实现于RD_for_one_BB函数中。该函数的调用参数及含义见代码注释：

```
// Realize the Meet operation and Transfer Function, return 1 if OUT has changed
// BB_res is the current BB, and res is the globle res for all BBs
/*
    input:
    BB_array : global CFG
    BB : BB index which will be analysis this time
    BB_res : reaching definition analysis result of this BB
    res : reaching definition analysis result of all the BBs
    def_index : how many Definitions in this program
    def_vars : Var of each definition, using the same index with def_tac

    output:
    flag : whether OUT change of this BB. 1 => OUT changed
*/
```

另外，本项目的reaching definition分析与课程中介绍的原理稍有差别。为了实现复制传播优化，我们认为在次应用场景下应该采用must ananlysis。所以设置了一个control bit作为区分，从而可以进行may与must两种分析（实际上只用到了control bit = 0的情景）。

2. Costant Propagation分析

常量传播分析判断在每一个程序点处的某个变量是否为一个常量值，该数据流分析用于常量传播/常量折叠优化，是一个前向/must类型的数据流分析。

Constant_Propagation实现了常量传播分析。该函数的执行逻辑类似前文所述的Reaching_definition，在初始化相关数据结构后使用Working List算法针对每一个基本快进行分析。

常量传播的分析较为复杂，因为需要讨论TAC的类型以判断能否确定某个程序变量的取值，该部分逻辑位于CP_for_one_BB中。

3. live_var 分析

live var分析可以分析当前变量的值后续是否会用到。其可以用于死代码消除，对于无效的赋值语句，我们可以直接将其删除。

参考南京大学软件分析课程。活跃变量分析采用的是一种前向分析方式，我们需要维护一个变量名到比特向量的映射。对每一个block都初始化为全0。然后对每一个block采用前向分析，若在某一个tac中该变量被使用，则将其对应bit置1，若该变量被定义或者赋值，则将其置0。因为这是一个may analysis，所以block的输出为其后继block的输入取并集。

为了防止声明语句被赋值语句覆盖，我们新增了2号状态。当前向分析遇到赋值语句时，会将等号左边变量状态置为2。在优化阶段，对于赋值语句，若其等号左边的变量对应的状态不为1则将其删除，对于定义var语句，若其左边等号变量对应的状态为0才能将其删除。

4.

优化器实现

完成数据流分析后，即可以应用分析结果实现相应的优化器。

1. 简单优化器 simple_opt

针对场景：

顾名思义非常简单，该优化器仅针对COPY TAC的源操作数与目标操作数相同的情况进行优化，比如x = x。

优化器实现：

遍历TAC列表，搜索x = x的情况，搜索到之后删除该TAC。

2. 常量传播+常量折叠优化器 const_opt

针对场景：

对于各类赋值语句（包括运算/COPY）与IF语句，如果我们能够确定此时目标操作数的取值，即可以省去该条指令的操作，直接将其优化为一条var = INT或IFZ INT的TAC。在此步骤中可能难以看出对整体优化的贡献，但是常量传播与常量优化为其他优化器如死代码消除/无效赋值消除等优化创造了条件。

优化器实现：

首先进行常量传播分析Constant_Propagation。获得分析结果后遍历TAC列表搜索可优化的语句。正如上文所述，找到目标操作数的值可确定的赋值语句后，直接将该条语句的操作数优化为INT。

3. 复制传播优化 copy_propagaton_opt

针对场景：

对于：

```
x = a;
y = x;
z = x;
```

的情景，在后续x=a这条语句能达到的程序位置上即可以使用a代替x。如经过复制传播优化后，上述TAC可以优化为：

```
x = a;
y = a;
z = a;
```

优化器实现：

首先进行可达分析Reaching_definition，注意这里使用must的分析方式，因为如果无法确定x=a一定能够到达某个位置，可能进行语义错误的优化。

完成可达分析后，遍历TAC语句寻找变量赋值，形如x=a。找到这样的语句后，从该位置向后遍历，寻找到使用x变量的位置，根据可达分析结果，如果使用x变量的位置处x=a可达，则使用a替换变量x。

4. deadvar优化器

该优化器实现在opt.c中，函数名为deadvar_opt

优化器实现：

根据live_var分析得到每个tac的out_vector，对赋值语句和变量声明语句进行删除。对于赋值语句，若状态不为1则删除，若为变量声明语句，则状态为0时删除。

5. 控制流不可达优化器

针对场景：

控制流不可达（Unreachable Control Flow）是指在程序中存在一些代码路径，它们永远不会被执行到。也就是说，这些代码路径无法通过程序的控制流来到达。其通常是由于分支不可达到造成的。

优化器实现：
根据basic block的定义可知一个basic block要么全是不可达代码要么全是可达代码。所以我们的思路是遍历basic block，删除那些没有被遍历的block。为了让永远不可达的分支不被遍历，所以需要判断每个if的情况。

先将所有basic block的flag置零，然后检测分支，删除不会到达分支的联系(dead_brunch)。深度遍历basic block(dfs)并且将对应basic block的flag置1，最后删除所有flag为0的basic block(kill_BB)。