Lab2 数据流分析实验报告

1 实验要求

对 Lab1 提供的编译器生成的三地址码做数据流分析并实现优化，具体为：
1)构建控制流图（CFG）
2)做到达定值的数据流分析，并在此基础上实现常数传播优化
3)做活跃变量的数据流分析，并在此基础上实现死代码消除优化

2 实验环境

Ubuntu 16.04
gcc 5.4.0
boost 库 1.58.0

3 实验过程

3.1 准备工作

3.1.1 三地址码分析

在着手优化工作之前首先需要分析目标三地址码的结构，通过对照 lab2 提供的实例程序源码及其生成的三地址码，将目标三地址码的指令分类整理如下：

```
map<string, int> op_num = {
    {"wrl",0}, {"entrypc",0}, {"nop",0},
    {"neg",1}, {"br",1}, {"call",1}, {"load",1},  {"enter",1},
    {"ret",1}, {"param",1}, {"read",1}, {"write",1},
    {"store",2}, {"add",2}, {"sub",2}, {"mul",2}, {"div",2},
    {"mod",2}, {"cmpeq",2},  {"cmple",2}, {"cmplt",2},
    {"blbc",2}, {"blbs",2},{"move",2}
};

//NO_USE_DEF
#define NUD 0
#define USE 1
#define DEF 2
map<string, int>ud1_table={
    {"neg",USE}, {"br",NUD}, {"call",NUD}, {"load",USE},  {"enter",NUD},
    {"ret",NUD}, {"param",USE},{"read",DEF}, {"write",USE}
};
map<string, pair<int,int> >ud2_table={
    {"store",make_pair(USE,DEF)},   {"add",make_pair(USE,USE)},
    {"sub",make_pair(USE,USE)},     {"mul",make_pair(USE,USE)},
    {"div",make_pair(USE,USE)},     {"mod",make_pair(USE,USE)},
    {"cmpeq",make_pair(USE,USE)},   {"cmple",make_pair(USE,USE)},
    {"cmplt",make_pair(USE,USE)},   {"blbc",make_pair(USE,USE)},
    {"blbs",make_pair(USE,USE)},    {"move",make_pair(USE,DEF)}
};
```

其中：

wrl，entrypc，nop 等 3 条指令没有操作数；neg，br，call，load，enter，ret，param，read，write 等 9 条指令有一个操作数；store，add，sub，mul，div，mod，cmpeq，cmple，cmplt，blbc，blbs，move 等 12 条指令有两个操作数。各条指令对其操作数的 use 和 def 属性记录在表中供之后使用。

指令的操作数共有 4 种形式：

1） 全局指针，表示形式为 GP

2） 常数，表示形式为常数本身；

3） 临时寄存器，表示形式为用小括号括起的临时寄存器号

4） 变量，表示形式为变量名+“#”+地址偏移，全局变量会在变量名之后加后缀“_base”

5） 转移目标地址，表示形式为用中括号括起的指令编号

（Lab1 提供的说明中还包括 frame pointer 和以“_offset”为后缀的变量作为操作数，用以支持结构体类型的运算。为简略起见**本实验未实现对结构体访问以及数组访问的支持**）

3.1.2 数据结构抽象

为了更好的表示三地址码代表的结构，需要对代码的各个结构进行抽象，分别为：

1） 指令，成员变量包括 opcode，操作数数量，各个操作数，每个操作数的 def 和 use 属性等

2） 标识符，成员变量包括变量名，类型（局部变量还是全局变量），地址等

```
struct instruction
{
    int id;
    string op_code;
    int op_num;
    string op1;
    string op2;
    int use1=-1;//symbol_table index
    int use2=-1;
    int def=-1;
    instruction(){
        id = 0;
        op_code="nop";
        op_num = 0;
    }
    instruction(char *op_code){
        this->op_code = string(op_code);
    }
};
```

```
struct symbol
{
    string name;
    int type;
    int address;
    bool operator<(const symbol a) const{
        return
        (this->name < a.name) ||
        (this->name == a.name && this->type < a.type);
    }
    bool operator==(const symbol a) const{
        return (this->name == a.name && this->type == a.type);
    }
    symbol(){}
    symbol(string &n,int addr,int t){
        if(t == GLOBAL){
            size_t index = n.rfind("_base");
            // if(index != std::string::npos){
                n.replace(index, 5, "");
            // }
        }
        type = t;
        name = n;
        address = addr;
    }
};
```

3） 基本块，成员变量包括基本块的起止指令编号，基本块中 use，def，gen，kill 信息和数据流分析时的 in 和 out 结果，以及基本块的前驱和后继基本块的下标

4） def 指令，为了方便做到达定值分析，将含有定值操作（move 和 store 指令）的高层次的指令单独抽象出来，成员变量包括该高层次指令的起止指令编号，其 def 和 use 的变量，以及如果是将变量定值为常数时，将常数数值记录。

5） 函数，成员变量包括函数的起止指令编号，函数中包含的基本块，符号表以及定值指令表

```cpp
//instruction that performs a def,
//move and store are considered
//read is not considered
struct def_instruction
{
    int start_ins;
    int end_ins;
    int def;//symbol_table index
    vector<int> use;//symbol_table index
    long long value;
    def_instruction(){}
};
```

```cpp
class basic_block
{
public:
    int start_ins;
    int end_ins;
    boost::dynamic_bitset<> use;
    boost::dynamic_bitset<> def;
    boost::dynamic_bitset<> gen;
    boost::dynamic_bitset<> kill;
    boost::dynamic_bitset<> in;
    boost::dynamic_bitset<> out;
    vector<int> pre;//存储在function中bbs的下标
    vector<int> suc;//存储在function中bbs的下标
    basic_block(int start_ins, int end_ins){
        this->start_ins = start_ins;
        this->end_ins = end_ins;
    }
};
```

```cpp
class Function
{
public:
// private:
    int start_ins;
    int end_ins;
    vector<basic_block> bbs;
    map<int, int> bb_map;
    vector<symbol> symbol_table;
    vector<def_instruction> def_ins_table;
// public:
    Function(){}
    Function(int start_ins, int end_ins){
        this->start_ins = start_ins;
        this->end_ins = end_ins;
    }
    ~Function(){}
};
```

使用全局变量保存程序的全部指令和函数。

```cpp
vector<instruction> insts;
vector<Function> funcs;
```

3.2 划分基本块

构建 CFG 的第一步是划分基本块。首先读取三地址码文件，读取每条指令并将其存入 insts 数组中，记录基本块的第一条指令的编号，用以划分基本块。基本块的入口为：每个函数的第一条指令（entry 指令）；跳转指令（call 和 br 指令）的目标地址；转移指令（blbc 和 blbs 指令）的目标地址和下一条指令。找到所有基本块的入口之后就完成了基本块的划分。本部分功能由函数 parse()完成。

3.3 构建 CFG

构建 CFG 的关键在于找到基本块之间的边，即控制流的转移方向。在 parse()函数中除了寻找基本块的入口，也记录了控制流信息，比如跳转指令所在基本块一定有一条指向目标地址所在基本块的边；转移指令所在基本块一定有一条指向目标地址所在基本块的边以及一条指向下一条指令所在基本块的边。此外，目标地址的上一条指令所在基本块可能有一条指向目标地址所在基本块的边，因为目标地址一定是基本块的入口，例外情况是目标地址的上一条指令是一条跳转指令，此时这条边就不应该存在。根据以上原则可以得出 CFG 的所有边，然后即可构建出 CFG。构建 CFG 的功能由函数 construct_cfg()完成。

3.4 构建符号表和定值指令表

构建符号表需要重新遍历 insts 指令数组，记录每条指令中可能包含的变量，要记录的信息包括变量名，变量地址偏移，以及变量的类型（是全局变量还是局部变量），同时也会更新 insts 指令数组中指令对操作数的 use 和 def 信息，此时该信息是不完全的，当操作数为临时寄存器时，并没有对该寄存器做追溯确定其对应的变量。而构建定值指令表的过程就是对定值指令做追溯的过程。该过程会再次遍历 insts 指令数组，对 move 和 store 指令所定值以及使用的变量使用递归的方法进行追溯。这两个表均以函数为单位保存在 Function 类中。以上工作分别由函数 construct_symbol_table ()和 construct_def_ins_table()完成。

3.5 到达定值分析以及常数传播优化

3.5.1 提取基本块的 gen 和 kill 信息

进行到达定值分析首先需要获得基本块的 gen 和 kill 信息，生成对应的位向量。位向量的每一位对应一条定值指令。这里使用了 boost 库提供的 dynamic_bitset 数据结构来表示位向量。该过程会遍历每个基本块以及定制指令表，对每条定值指令，gen 位向量对应的位设为 1，同时对同一变量进行定值的其他定值指令在 kill 位向量中的位设为 1。以上工作由函数 generate_gen_kill_bit_vec_of_bb ()完成。

3.5.2 进行到达定值的迭代数据流分析

然后进行到达定值的迭代数据流分析。首先将每个基本块的 in 和 out 初始化为 0，然后根据到达定值的数据流分析方程对 in 和 out 进行计算。关键迭代代码如下：

```
do{
    flag = false;
    for(int i=0;i<f->bbs.size();i++){
        if(f->bbs[i].pre.size() == 0){
            in[i].reset();
        }
        else{
            // IN[B] = UNION{P,pre(B)}OUT[P]
            in[i] = out[f->bbs[i].pre[0]];
            for(int j = 1;j < f->bbs[i].pre.size();j++){
                in[i] |= out[f->bbs[i].pre[j]];
            }
        }
        boost::dynamic_bitset<> tmp(out[i]);

        // OUT[B] = GEN{B} UNION (IN[B] - KILL{B})
        out[i] = f->bbs[i].gen | (in[i] & ~(f->bbs[i].kill));

        if(out[i] != tmp){
            flag = true;
        }
    }
}
while(flag);
```

可见使用位向量以及 dynamic_bitset 类进行迭代运算的过程十分清晰。以上工作由函数 dfa_reaching_definitions ()完成。

3.5.3 常数传播优化

根据 dfa 的结果进行常数传播优化，核心思想是：对每一个基本块的 IN，如果变量 x 只有一个定值并且该定值是常数，或者 x 有多个定值并且所有定值都是常数且相等，则可以把 x 的所有使用都替换成该常数。以上工作由函数 contstant_propagation ()完成。

3.6 活跃变量分析以及死代码消除优化

3.6.1 提取基本块的 use 和 def 信息

同到达定值分析相同，首先需要获得基本块的 use 和 def 信息，生成对应的位向量，不同的是这里的位向量的每一位对应的不是定值指令而是变量。该过程会遍历每个基本块，对每个变量的 use 和 def 信息进行记录（def 信息已经存在定值指令表之中了，此处直接读取；对 use 信息进行递归回溯收集）。需要注意的是如果在一个基本块中一个变量的 use 在 def 之前，则只记录其 use 信息而不记录 def 信息。如果 def 在 use 之前，只记录其 def 信息而不记录 use 信息。以上工作由函数 generate_use_def_bit_vec_of_bb ()完成。

3.6.2 进行活跃变量的迭代数据流分析

同到达定值分析相同，此处根据活跃变量分析的数据流分析方程对 in 和 out 进行计算。关键迭代代码如下：

```
do{
    flag = false;
    for(int i=0;i<f->bbs.size();i++){
        if(f->bbs[i].pre.size() == 0){
            in[i].reset();
        }
        else{
            // IN[B] = UNION{P,pre(B)}OUT[P]
            in[i] = out[f->bbs[i].pre[0]];
            for(int j = 1;j < f->bbs[i].pre.size();j++){
                in[i] |= out[f->bbs[i].pre[j]];
            }
        }
        boost::dynamic_bitset<> tmp(out[i]);

        // OUT[B] = GEN{B} UNION (IN[B] - KILL{B})
        out[i] = f->bbs[i].gen | (in[i] & ~(f->bbs[i].kill));

        if(out[i] != tmp){
            flag = true;
        }
    }
}
while(flag);
```

以上工作由函数 dfa_living_variables ()完成。

3.6.3 死代码消除优化

根据 dfa 的结果进行死代码消除优化，核心思想是：对每一个基本块的 OUT，如果变量 x 不是 alive 的，则基本块内对 x 的定值可以删除。以上工作由函数 dead_code_elimination() 完成。

4 实验结果

4.1 到达定值与常数传播

首先需要准备待优化的代码并使用 Lab1 提供的编译器生成三地址码。待优化的代码使用了课件上的例子，源码如下：

```
 8  void main()
 9  {
10      long s, a, i, k, b, n;
11      s = 0;
12      a = 4;
13      i = 0;
14      if(k == 0){
15          b = 1;
16      }
17      else{
18          b = 2;
19      }
20      while(i < n){
21          s = s + a*b;
22          i = i + 1;
23      }
24      WriteLong(s);
25  }
26
```

可见源码第 21 行对 a 的 use 可以替换为常量 4，同时对 b 的 use 由于控制流的原因不能替换为常量 1 或 2。

对以上源码分析的结果如下：

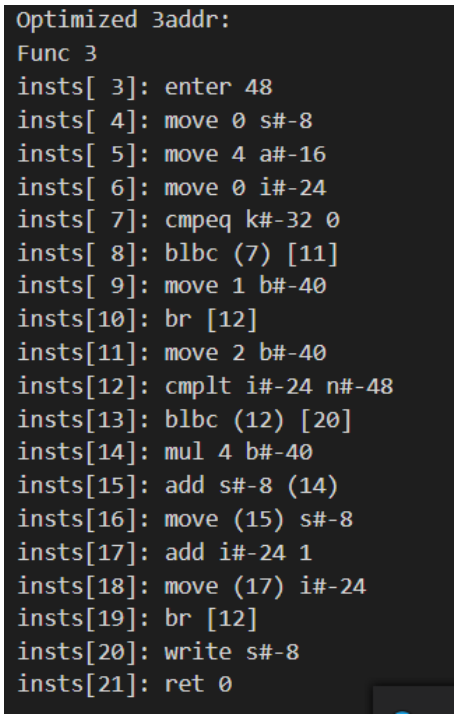
基本块划分，CFG 以及各个基本块的 GEN 和 KILL 信息以及 IN 和 OUT 结果如下：

```
Function: 3
Basic blocks: 3 9 11 12 14 20
CFG:
3 -> 9 11
9 -> 12
11 -> 12
12 -> 14 20
14 -> 12
20 ->
```

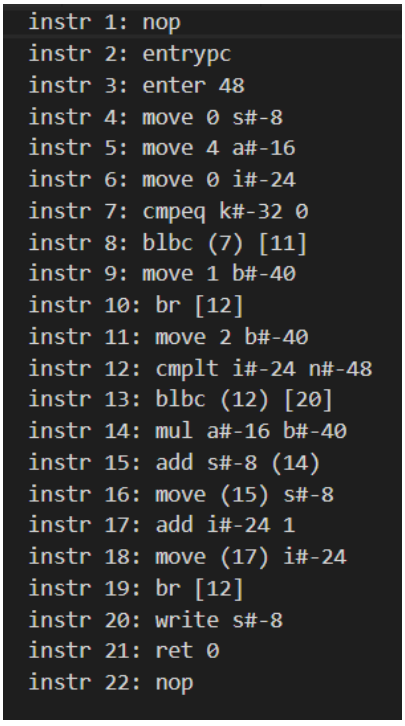
```
bit vector:
Func 3
basic block 3-8: gen:0000111; kill:1100000
basic block 9-10: gen:0001000; kill:0010000
basic block 11-11: gen:0010000; kill:0001000
basic block 12-13: gen:0000000; kill:0000000
basic block 14-19: gen:1100000; kill:0000101
basic block 20-21: gen:0000000; kill:0000000
```

```
IN OUT:
Func 3
basic block 3-8: in:0000000; out:0000111
basic block 9-10: in:0000111; out:0001111
basic block 11-11: in:0000111; out:0010111
basic block 12-13: in:1111111; out:1111111
basic block 14-19: in:1111111; out:1111010
basic block 20-21: in:1111111; out:1111111
```

优化后的三地址码（左图）以及源程序的三地址码（右图）如下：



```
Optimized 3addr:
Func 3
insts[ 3]: enter 48
insts[ 4]: move 0 s#-8
insts[ 5]: move 4 a#-16
insts[ 6]: move 0 i#-24
insts[ 7]: cmpeq k#-32 0
insts[ 8]: blbc (7) [11]
insts[ 9]: move 1 b#-40
insts[10]: br [12]
insts[11]: move 2 b#-40
insts[12]: cmplt i#-24 n#-48
insts[13]: blbc (12) [20]
insts[14]: mul 4 b#-40
insts[15]: add s#-8 (14)
insts[16]: move (15) s#-8
insts[17]: add i#-24 1
insts[18]: move (17) i#-24
insts[19]: br [12]
insts[20]: write s#-8
insts[21]: ret 0
```

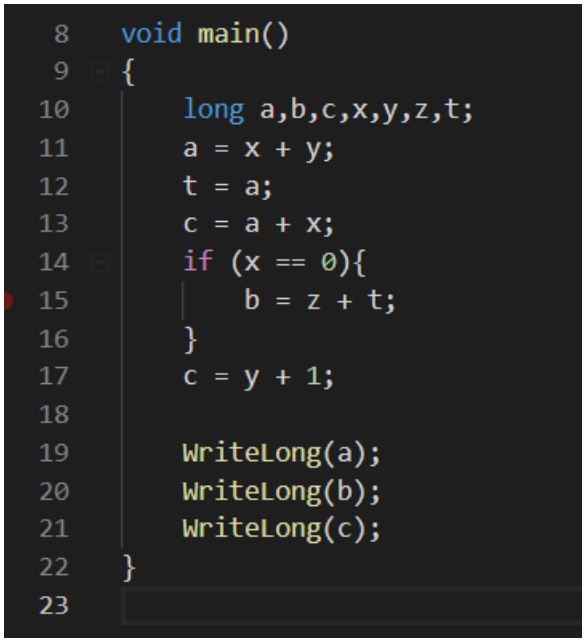


```
instr 1: nop
instr 2: entrypc
instr 3: enter 48
instr 4: move 0 s#-8
instr 5: move 4 a#-16
instr 6: move 0 i#-24
instr 7: cmpeq k#-32 0
instr 8: blbc (7) [11]
instr 9: move 1 b#-40
instr 10: br [12]
instr 11: move 2 b#-40
instr 12: cmplt i#-24 n#-48
instr 13: blbc (12) [20]
instr 14: mul a#-16 b#-40
instr 15: add s#-8 (14)
instr 16: move (15) s#-8
instr 17: add i#-24 1
instr 18: move (17) i#-24
instr 19: br [12]
instr 20: write s#-8
instr 21: ret 0
instr 22: nop
```

可以看到 14 号指令对 a 的 use 已经替换为常量 4，对 b 的 use 仍保持原样。

4.2 活跃变量分析与死代码消除

待优化的代码同样使用了课件上的例子，源码如下：



```
8   void main()
9   {
10      long a,b,c,x,y,z,t;
11      a = x + y;
12      t = a;
13      c = a + x;
14      if (x == 0){
15          b = z + t;
16      }
17      c = y + 1;
18
19      WriteLong(a);
20      WriteLong(b);
21      WriteLong(c);
22  }
23
```

可见源码第 13 行对 c 的定值是无效的，可以删去。

对以上源码分析的结果如下：

基本块划分，CFG 以及各个基本块的 GEN 和 KILL 信息以及 IN 和 OUT 结果如下：

```
Function: 3
Basic blocks: 3 11 13
CFG:
3 -> 11 13
11 -> 13
13 ->
```

```
bit vector:
Func 3
symbol: x y a t c z b
basic block 3-10: use:0000011; def:0011100
basic block 11-12: use:0101000; def:1000000
basic block 13-18: use:1000110; def:0010000
```

```
IN OUT:
Func 3
basic block 3-10: in:1100011; out:1101111
basic block 11-12: in:0101111; out:1101111
basic block 13-18: in:1101111; out:1111111
```

优化后的三地址码（左图）以及源程序的三地址码（右图）如下：

```
Optimized 3addr:
Func 3
insts[ 3]: enter 56
insts[ 4]: add x#-32 y#-40
insts[ 5]: move (4) a#-8
insts[ 6]: move a#-8 t#-56
insts[ 7]: nop
insts[ 8]: nop
insts[ 9]: cmpeq x#-32 0
insts[10]: blbc (9) [13]
insts[11]: add z#-48 t#-56
insts[12]: move (11) b#-16
insts[13]: add y#-40 1
insts[14]: move (13) c#-24
insts[15]: write a#-8
insts[16]: write b#-16
insts[17]: write c#-24
insts[18]: ret 0
```

```
instr 1: nop
instr 2: entrypc
instr 3: enter 56
instr 4: add x#-32 y#-40
instr 5: move (4) a#-8
instr 6: move a#-8 t#-56
instr 7: add a#-8 x#-32
instr 8: move (7) c#-24
instr 9: cmpeq x#-32 0
instr 10: blbc (9) [13]
instr 11: add z#-48 t#-56
instr 12: move (11) b#-16
instr 13: add y#-40 1
instr 14: move (13) c#-24
instr 15: write a#-8
instr 16: write b#-16
instr 17: write c#-24
instr 18: ret 0
instr 19: nop
```

可以看到 7 号和 8 号指令对 c 的定值已经被删去（替换为 nop 指令）。

5 实验总结

本次实验实现了两个基本的迭代数据流分析方法，并在分析结果的基础之上对目标三地址码实现了相应的优化。实验中自我感觉对于中间数据如何存储的决策比较困难，走了不少弯路，对数据结构的抽象也是在不断摸索之中慢慢总结出来的。实现可能有冗余的部分，比如定值指令表的计算等。另外由于时间原因没有实现对数组和结构体的支持，部分测试也做的不够充分，但是就结果来说实现了一个完整的迭代数据流分析过程，让我加深了对数据流分析的认识。