1 Algorithm for program SMT

1.1 Definition

Some definitions are made in order to make the algorithm easier understood.

1. F_i is SMT formula for the basic block i.
2. $r_{i\_j\_k}$ represents the register value, where i is the register ID, j is the basic block ID, k is the version ID (i.e., the number of update times in this block). k starts from 0 and each update in this basic block will increase k by 1. For example, $r_{0\_0\_1}$ is the value that represents r_1 in basic block 0 (i.e., the starting basic block), and r_1 has been updated once in basic block 0. j ensures that each basic block has its own register version, and k ensures that different values of the same register can be differed in the same basic block.
3. f_{pl} is the formula of program logic for the current basic block.
4. $IV = \{iv_0, iv_1, ...\}$ is the initial value set for all basic blocks. Each item $iv_i = \langle iv_{i\_0}, iv_{i\_1}, ...\rangle = \langle r_{0\_\boldsymbol{i}\_k_0}, r_{1\_\boldsymbol{i}\_k_1}, ...\rangle$ represents the initial values of all registers ***generated*** by basic block i.
5. f_{iv} is the formula representing the logic that initial values of register from the last basic block b_1 are fed to the register values in current basic block b_2, that is,
$$\begin{aligned} f_{iv} &= \bigwedge_i r_{i\_b_1\_k_i} == r_{i\_b_2\_0} \\ &= \bigwedge_i iv_{b_1\_i} == r_{i\_b_2\_0} \end{aligned} \tag{1}$$
6. $C = \{c_{b_{i_1}\to b_{j_1}}, c_{b_{i_2}\to b_{j_2}}, ...\}$ is the initial path condition formulas set for all edges. Each item $c_{b_i\to b_j}$ is a path condition formula generated by the basic block b_i for the basic block b_j. The post path condition C_b for the basic block b is
$$C_b = \bigcup_i \{c_{parents} \wedge c_{instEnd\_case_i}\} \tag{2}$$
where $c_{parents} = \bigvee_p c_{p\to b}$

1.2 Target SMT formula

SMT formula for the program (do not take pre-condition or post-condition into consideration here) is

$$\begin{aligned} F &= \bigwedge_i F_i \\ &= \bigwedge_i \bigwedge_j c_{j\to i} \to f_{iv\_i} \wedge f_{pl\_i} \end{aligned} \tag{3}$$

Algorithm 1 SMT_Prog

Input: $instLst$, $instLstLen$
Output: F, $outputRegLst$

// Sort basic blocks by Topological_Sorting
// $B = \langle b_0, b_1, ... \rangle$ is set of basic blocks, b_0 is the start basic block
// $InB = \langle inb_0, inb_1, ... \rangle$ is set of incoming edges for each basic block
// $OutB = \langle outb_0, outb_1, ... \rangle$ is set of outgoing edges for each basic block
// InB and $OutB$ are from cfg
$cfg \Leftarrow Gen\_Cfg(instLst, instLstLen)$
$B \Leftarrow Topological\_Sorting(cfg)$

// Process Basic Block 0
$f_{pl} \Leftarrow Gen\_Block\_Prog\_Logic(0, instLst[start : end])$
$F_0 \Leftarrow f_{pl}$
$iv_0 \Leftarrow Get\_Cur\_Reg\_Val(0)$
$IV \Leftarrow \{iv_0\}$
$C \Leftarrow Gen\_Path\_Cond\_For\_Next\_Blocks(0, instLst[end], inb_0, outb_0)$

// Process Basic Block $1 \rightarrow (len - 1)$
for $i = 1$ to $len(B) - 1$ **do**
    $f_{pl} \Leftarrow Gen\_Block\_Prog\_Logic(i, instLst[start : end])$
    $F_i \Leftarrow true$
    for each last block b in inb_i **do**
        $f_{iv} \Leftarrow \bigwedge_{j=1}^{j=NumOfReg} iv_{b\_j} == r_{j\_i\_0}$
        $F_i \Leftarrow F_i \wedge (c_{b \rightarrow i} \rightarrow (f_{iv} \wedge f_{pl}))$
    end for
    $iv_i \Leftarrow Get\_Cur\_Reg\_Val(i)$
    $IV \Leftarrow IV \cup \{iv_i\}$
    $C_{out\_i} \Leftarrow Gen\_Path\_Cond\_For\_Next\_Blocks(i, instLst[end], inb_i, outb_i)$
    $C \Leftarrow C \cup C_{out\_i}$
end for

for $i = 0$ to $len(B) - 1$ **do**
    $F \Leftarrow F \wedge F_i$
end for

initialize $outputRegLst$
for $i = 0$ to $NumOfReg - 1$ **do**
    $outputRegLst_i \Leftarrow iv_{(instLstLen-1)\_i}$
end for
return $outputRegLst_i$

Algorithm 2 Gen_Block_Prog_Logic

Input: Basic Block ID b, $instLst$
Output: f_{pl}
1: $f_{pl} \Leftarrow true$
2: **for** $i = 1$ to $len(instLst)$ **do**
3: **if** type of instruction is not JMP or END **then**
4: $f_{pl} \Leftarrow f_{pl} \wedge f_{instruction\_i}$
5: **end if**
6: **end for**
7: **return** f_{pl}

Algorithm 3 Get_Cur_Reg_Val

Input: Basic Block ID b, R_b
Output: iv_b
1: $iv_b \Leftarrow \emptyset$
2: **for** $i = 0$ to $(NumberOfRegisters - 1)$ **do**
3: $//R$ stores the current values of all register for the basic blocks
4: $iv_b \Leftarrow iv_b \cup \{R_{b\_i}\}$
5: **end for**
6: **return** iv_b

Algorithm 4 Gen_Path_Cond_For_Next_Blocks

Input: Basic Block ID b, $instEnd$, $inBlocks$, $outBlocks$
Output: C_{out}
1: $c_{in} \Leftarrow true$
2: **for** each last block b_i in $inBlocks$ **do**
3: $c_{in} \Leftarrow c_{in} \vee c_{b_i \rightarrow b}$
4: **end for**
5: $C_{out} \Leftarrow \emptyset$
6: $c_{instEnd} \Leftarrow ParseInstruction(instEnd)$
7: **for** each last block c in $c_{instEnd}$ **do**
8: $C_{out} \Leftarrow C_{out} \cup \{c_{in} \wedge c\}$
9: **end for**
10: **return** C_{out}

1.3 Example

1. Instructions:

```
inst(MOVXC, 2, 15), // 0 mov r2, 15
inst(JMPGT, 0, 2, 2), // 1 if r0 <= r2 no jmp else jmp to 4
inst(ADDXY, 0, 1), // 2 add r0, r1
inst(RETX, 2), // 3 ret r2
inst(ADDXY, 2, 1), // 4 add r2, r1
inst(RETX, 0), // 5 else ret r0
```

2. Instructions → CFG
   CFG:
   nodes: 0[0:1] 1[2:3] 2[4:5]
   edges: $0 \to 1$ $0 \to 2$

3. List by topological Sorting
   $B = \langle 0, 1, 2 \rangle$
   $InB = \langle \langle\rangle, \langle 0 \rangle, \langle 0 \rangle \rangle$
   $OutB = \langle \langle 1, 2 \rangle, \langle\rangle, \langle\rangle \rangle$

4. Process Node 0
   $f_{pl} = (r_{2\_0\_1} == 15)$
   $iv_0 = \langle r_{0\_0\_0}, r_{1\_0\_0}, r_{2\_0\_1} \rangle$
   $IV = \{iv_0\}$
   $F_0 = (r_{2\_0\_1} == 15)$
   $C_{out\_0} = \{c_{0\to1} = (r_0 > r_2), c_{0\to2} = \neg(r_0 > r_2)\}$
   $C = C_{out\_0}$

5. Process Node 1
   $f_{pl} = (r_{0\_1\_1} == r_{0\_1\_0} + r_{1\_1\_0})$
   $iv_1 = \langle r_{0\_1\_1}, r_{1\_1\_0}, r_{2\_1\_0} \rangle$
   $IV = \{iv_0, iv_1\}$
   $f_{iv} = \bigwedge_{i=1}^{i=3}(iv_{0\_i} == r_{i\_1\_0})$
   $F_1 = c_{0\to1} \to f_{iv} \wedge f_{pl} = (r_0 > r_2) \to (\bigwedge_{i=1}^{i=3}(iv_{0\_i} == r_{i\_1\_0}) \wedge (r_{0\_1\_1} == r_{0\_1\_0} + r_{1\_1\_0}))$
   $C_{out\_1} = \emptyset$
   $C = C_{out\_0}$

6. Process Node 2
   $f_{pl} = (r_{2\_2\_1} == r_{2\_2\_0} + r_{1\_2\_0})$
   $iv_1 = \langle r_{0\_2\_0}, r_{1\_2\_0}, r_{2\_2\_1} \rangle$
   $IV = \{iv_0, iv_1, iv_2\}$
   $f_{iv} = \bigwedge_{i=1}^{i=3}(iv_{0\_i} == r_{i\_2\_0})$
   $F_2 = c_{0\to2} \to f_{iv} \wedge f_{pl} = \neg(r_0 > r_2) \to (\bigwedge_{i=1}^{i=3}(iv_{0\_i} == r_{i\_2\_0}) \wedge (r_{2\_2\_1} == r_{2\_2\_0} + r_{1\_2\_0}))$
   $C_{out\_2} = \emptyset$
   $C = C_{out\_0}$

7. Generate F
   $F = F_0 \wedge F_1 \wedge F_2$
   where
   $F_0 = true \rightarrow (true \wedge (r_{2\_0\_1} == 15))$
   $F_1 = (r_0 > r_2) \rightarrow (\bigwedge_{i=1}^{i=3}(iv_{0\_i} == r_{i\_1\_0}) \wedge (r_{0\_1\_1} == r_{0\_1\_0} + r_{1\_1\_0}))$
   $F_2 = \neg(r_0 > r_2) \rightarrow (\bigwedge_{i=1}^{i=3}(iv_{0\_i} == r_{i\_2\_0}) \wedge (r_{2\_2\_1} == r_{2\_2\_0} + r_{1\_2\_0}))$

2 Algorithm for equivalence check

2.1 Definition

1. Since there are two programs, a new dimension, program ID, should be added into register value $r_{i\_j\_k}$ to ensure that each program has it own register value version. Then $r_{i\_j\_k}$ extends to $\boldsymbol{r_{i\_j\_k\_p}}$, where p is the program ID.

2. Equivalence check formula

$$F = F_{pre_1} \wedge F_{pre_2} \wedge F_1 \wedge F_2 \rightarrow F_{post} \tag{4}$$

   where
   F_{pre_i}, F_i, F_{post} are the pre-conditon, program logic, post-condition FOL formulas of program i, that is,

$$F_{pre_i} = (r_{0\_*\_*\_i}) \wedge (\bigwedge_{j=1}^{j=NumOfReg-1} r_{j\_*\_*\_i} == 0) \tag{5}$$

$$\begin{aligned} F_i &= (\bigwedge_{j=0}^{j=NumOfInstLst-1} f_{inst\_j}) \wedge (F_{output}) \\ &= (\bigwedge_{j=0}^{j=NumOfInstLst-1} f_{inst\_j}) \wedge (\bigwedge_{r_k=reg[ret_k]} (c \rightarrow r_k == output_i)) \end{aligned} \tag{6}$$

$$F_{post} = (output_1 == output_2) \tag{7}$$

2.2 Algorithm

3 Test set

3.1 Program equivalence

1. no branch

Algorithm 5 Equivalence_Check

Input: $input$, $instLst_1$, len_1, $instLst_2$, len_2
Output: $isEqual$

// pre-condition formula: $input_1 == input_2$
$F_{pre_1} \Leftarrow true$
$F_{pre_2} \Leftarrow true$
for each value $\boldsymbol{rv}$ of register $\boldsymbol{i}$ in $input$ **do**
    $F_{pre_1} \Leftarrow F_{pre_1} \wedge (rv == r_{i\_0\_0\_1})$
    $F_{pre_2} \Leftarrow F_{pre_2} \wedge (rv == r_{i\_0\_0\_2})$
end for
// program logic formula: F_1, F_2
$F_1, output_1 \Leftarrow SMT\_Prog(instLst_1, len_1)$
$F_2, output_2 \Leftarrow SMT\_Prog(instLst_2, len_2)$
// post-condition formula: $output_1 == output_2$
$F_{post} \Leftarrow true$
for $i = 0$ to $len(output_1)$ **do**
    $F_{post} \Leftarrow F_{post} \wedge (output_{1\_i} == output_{2\_i})$
end for
$F \Leftarrow F_{pre_1} \wedge F_{pre_2} \wedge F_1 \wedge F_2 \rightarrow F_{post}$
$isEqual \Leftarrow Is\_Unsat(\neg F)$
return $isEqual$

2. branch: use more arithmetic and jmp instructions

   with RET,

   without RET

3. illegal input

3.2 Specification check

1. Instruction specification: the range of register ID,...;
   done by candidate program generator?
   Now it is done by candidate program generator.

2. Invalid instruction reach: Jmp instruction check; has done
   End instruction check → Is the program is always ended with **RETs**?
   For now, only with registers, if no RET, return r_0

3. No loop

4 Data structure

```
class node {
private:
public:
```

```
    unsigned int _start = 0; // start intruction ID
    unsigned int _end = 0; // end instruction ID
};

class graph {
private:
    vector<node> nodes;
    vector<vector<unsigned int> > nodesIn;
    vector<vector<unsigned int> > nodesOut;
public:
};

class progSmt {
private:
    // f[i] is program logic FOL formula F of basic block i
    vector<expr> f;
    // postRegVal[i] is post register values of basic block i,
    // which are initial values for NEXT basic blocks
    vector<vector<expr> > postRegVal;
    // pathCon[i] stores pre path conditions of basic block i
    // There is a corresponding relationship between pathCon and g.nodesIn
    // more specifically, pathCon[i][j] stores the pre path condition from basic
        block g.nodesIn[i][j] to i
    vector<vector<expr> > pathCon;
    // program FOL formula
    expr smt = stringToExpr("true");
    // program output FOL formula
    expr smtOutput = stringToExpr("true");
    // return the SMT for the given program
    expr smtProg(inst* program, int length, smtVar* sv);
    // return SMT for the given instruction
    expr smtInst(smtVar* sv, inst* in);
public:
};
```