IC637 Program Analysis

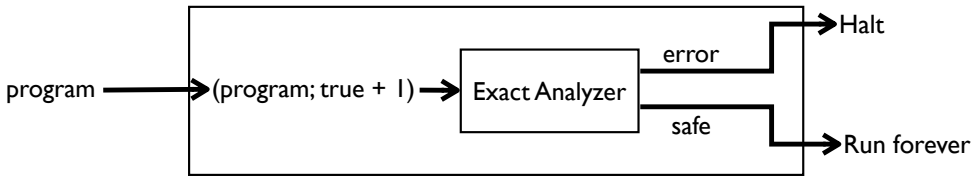
Lecture 1 Review

Minseok Jeon

2025 Fall

Fundamental Limitation

- It is **impossible** to develop an exact program analyzer.
- **Proof**
  1. The Halting problem[1] is not computable (i.e., undecidable).
  2. If we have an exact analyzer that soundly and completely finds error, we can solve the Halting problem with the analyzer.



- **Question:** specification also should be provided along with program!

[1] https://en.wikipedia.org/wiki/List_of_undecidable_problems

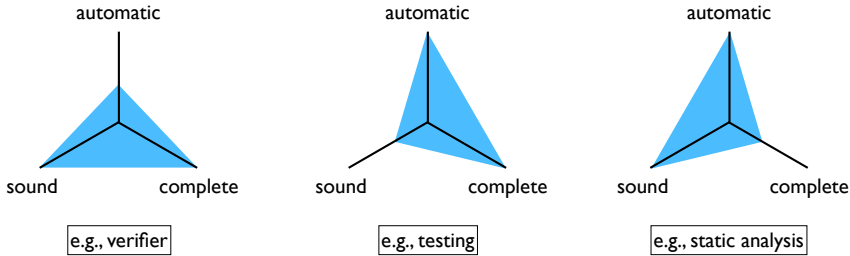
Question 1

- **Question:** specification also should be provided along with program!

```
1 void main() {
2     int i = input(); //Specification matters!
3     if (i > 0){
4         while(true){}
5     }
6     else{
7         return;
8     }
9 }
```

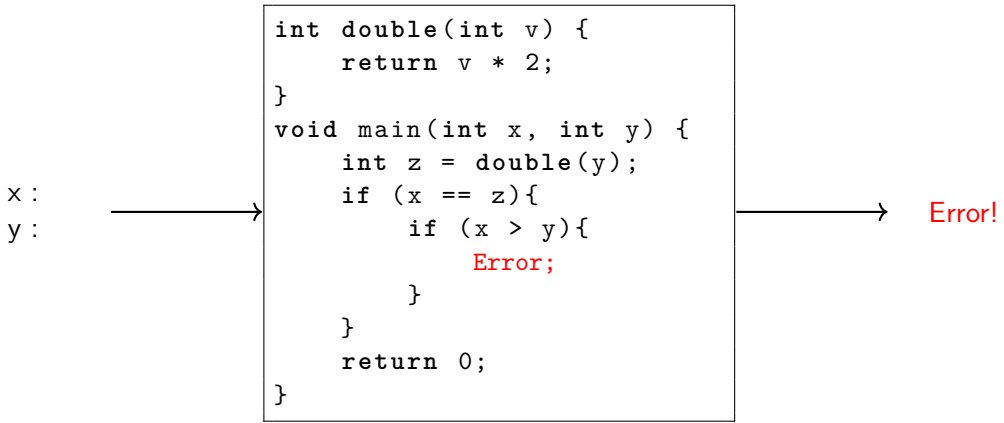
Tradeoff

- Three desirable properties
  - **Soundness :** all program behaviors (e.g., bugs) are captured.
    (If a program has a bug, a analyzer reports it.)
  - **Completeness :** only (possible) program behaviors are captured.
    (If a complete analyzer reports a bug, the program has the bug.)
  - **Automation :** the analyzer can be run automatically without human intervention.
- Achieving all the three properties is generally infeasible:



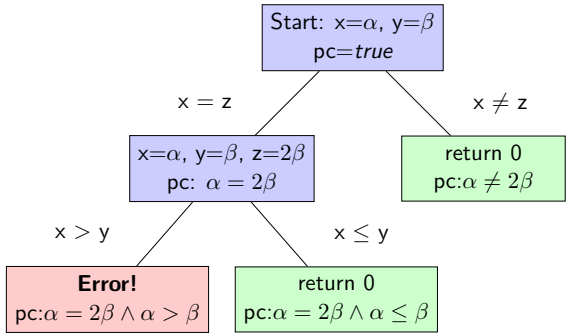
Random Testing/Fuzzing

- **Basic concept:** execute the program with randomly generated **concrete inputs**, analyzing individual program behavior separately.



Symbolic Execution

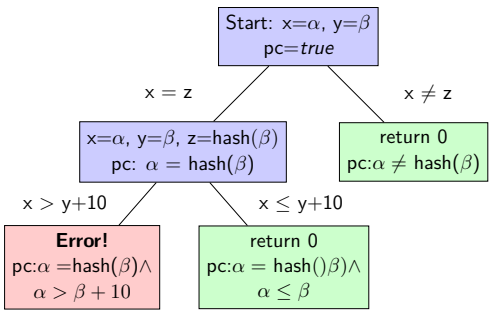
```
1  int double(int v) {
2      return v * 2;
3  }
4  void main(int x, int y) {
5      int z = double(y);
6      if (x == z){
7          if (x > y){
8              Error;
9          }
10     }
11     return 0;
12 }
```



Concolic Testing Example

- Limitation of symbolic execution

```
1  int foo(int v) {
2      return hash(v);
3  }
4  void main(int x, int y) {
5      int z = foo(y);
6      if (x == z){
7          if (x > y + 10){
8              Error;
9          }
10     }
11     return 0;
12 }
```



Concolic Testing Example

```
1  int foo(int v) {
2      return hash(v);
3  }
4  void main(int x, int y) {
5      int z = foo(y);
6      if (x == z){
7          if (x > y + 10){
8              Error;
9          }
10     }
11     return 0;
12 }
```

Problem with symbolic execution:

- Cannot reason about `hash(v)`
- Path explosion (usually)
- Complex constraints (usually)

Concolic solution:

- Execute with concrete values
- Track symbolic constraints
- Generate new inputs systematically

Symbolic Verification

```
1  int f(bool a) {
2      x = false; y = false;
3      if (a) {
4          x = true;
5      }
6      if (a){
7          y = true;
8      }
9      assert(x == y);
10 }
```

Verification condition:

$$((a \wedge x) \vee (\neg a \wedge \neg x)) \wedge$$
$$((a \wedge y) \vee (\neg a \wedge \neg y)) \wedge$$
$$\neg(x == y)$$

SMT solver: unsatisfiable!

Challenge: Loop Invariant

- Property that holds at the beginning of every loop iterations

```
1 i = 0;
2 j = 0;
3 while @(i == j)
4 (i < 10){
5     i++;
6     j++;
7 }
8 assert(i == j);
```

- **Question 2**: I guess the loop invariant in the example can be easily generated automatically.

Question 2

```
1  @pre :  0 ≤ lb ∧ ub < |a|
2  @post :  rv ⟺ ∃ i.lb ≤ i ≤ ub ∧ a[i] = e
3  bool LinearSearch (int[] a, int lb, int ub, int e) {
4    int i = lb;
5    while @?
6    (i <= ub){
7      if(a[i] == e) {return true;}
8      i = i + 1;
9    }
10   return false;
11 }
```

Question 2

```
1  @pre : 0 ≤ lb ∧ ub < |a|
2  @post : rv ⟺ ∃ i.lb ≤ i ≤ ub ∧ a[i] = e
3  bool LinearSearch (int[] a, int lb, int ub, int e) {
4    int i = lb;
5    while @L : lb ≤ i ∧ (∀j. lb ≤ j < i → a[j] ≠ e)
6    (i <= ub){
7      if(a[i] == e) {return true;}
8      i = i + 1;
9    }
10   return false;
11 }
```

Question 2

```
1  @pre :  |a0| ≥ 0
2  @post :  sorted(rv, 0, |rv| - 1)
3  int[] BubbleSort (int[] a0) {
4      int[] a = a0;
5      for (int i := |a| - 1; i > 0; i := i - 1) { @?
6          for (int j := 0; j < i; j := j + 1) { @?
7              if (a[j] > a[j + 1]) {
8                  int tmp = a[j];
9                  a[j] = a[j + 1];
10                 a[j + 1] = tmp;
11             }
12         }
13     }
14     return a;
15 }
```

Question 2

```
1  @pre :  |a0| ≥ 0
2  @post :  sorted(rv, 0, |rv| - 1)
3  int[] BubbleSort (int[] a0) {
4      int[] a = a0;
5      @L1 : −1 ≤ i < |a| ∧ partitioned(a, 0, i, i + 1, |a| − 1) ∧ sorted(a, i, |a| − 1)
6      for (int i := |a| - 1; i > 0; i := i - 1) {
7          @L2 : 1 ≤ i < |a| ∧ 0 ≤ j ≤ i ∧ partitioned(a, 0, i, i + 1, |a| − 1)∧
8           partitioned(a, 0, j − 1, j, j) ∧ sorted(a, i, |a| − 1)
9          for (int j := 0; j < i; j := j + 1) {
10             if (a[j] > a[j + 1]) {
11                 int tmp = a[j];
12                 a[j] = a[j + 1];
13                 a[j + 1] = tmp;
14             }
15         }
16     }
17     return a;}
18 partitioned(a, l1, u1, l2, u2) ⟺ ∀i, j.(l1 ≤ i ∧ i ≤ u1 ∧ l2 ≤ j ∧ j ≤ u2) → a[i] ≤ a[j]
```

Question 3

- **Question 3:** Are assertions really used in practice?

```
def sqrt(x):
    if x<0:
        raise ValueError, 'sqrt requires positive numbers'
    root = <do stuff>
    return root

def some_func(x):
    y = float(raw_input('Type a number:'))
    try:
        print 'Square root is %f'%sqrt(y)
    except ValueError:
        # User did not type valid input
        print '%f must be a positive number!'%y
```

Question 3

- **Question 3:** Are assertions really used in practice?

```
elif case == 0:
  #new_to
  node_var_fr = target_edge_var[1]
  node_var_to = target_edge_var[2]
  fr_con = node_var_idx_to_concrete_node[node_var_fr]
  #print(my_maps.succ_node_to_nodes)
  candidate_to_nodes = my_maps.succ_node_to_nodes[fr_con]
  for _, val  in enumerate(candidate_to_nodes):
    (con_edge, to_con) = val
    condition1 = not (to_con in subgraph[0])
    condition2 = concrete_node_belong_node_var(GDL_program.nodeVars[node_var_to], my_maps.A[con_edge][1], my_maps.X_node)
    condition3 = concrete_edge_belong_edge_var(target_edge_var, con_edge, my_maps.X_edge)
    if condition1 and condition2 and condition3:
      new_node_var_idx_to_concrete_node = copy.deepcopy(node_var_idx_to_concrete_node)
      new_node_var_idx_to_concrete_node[target_edge_var[2]] = to_con
      new_edge_var_idx_to_concrete_edge = copy.deepcopy(edge_var_idx_to_concrete_edge)
      new_edge_var_idx_to_concrete_edge[edge_var_idx] = con_edge

      new_subgraph = copy.deepcopy(subgraph)
      new_subgraph[0].append(to_con)
      new_subgraph[1].append(con_edge)
      val = find_subgraph_DFS(new_subgraph, new_sub_GDL_program, GDL_program, graph, edge_var_idx + 1, new_node_var_idx_to_
edge, my_maps)
      if val[0] == 0:
        return val
else:
  raise("Cannot be happened")
return (1, None)
```

Principles of Abstract Interpretation

$$\mathbf{30 \times 12 + 11 \times 9 = ?}$$

- Dynamic analysis (testing): 459
- Static analysis: a variety of answers
  - "integer", "odd integer", "positive integer", "400≤n≤500", "etc"
- Static analysis process:
  - Choose abstract value (domain), e.g., $\hat{V} = \{\top, e, o, \bot\}$
  - Define the program execution in terms of abstract values:

$\hat{\times}$	$\top$	e	o	$\bot$
$\top$				
e				
o				
$\bot$				

$\hat{+}$	$\top$	e	o	$\bot$
$\top$				
e				
o				
$\bot$				

$\hat{/}$	$\top$	e	o	$\bot$
$\top$				
e				
o				
$\bot$				

- Execute the program:

$$\boldsymbol{e} \,\hat{\times}\, \boldsymbol{e} \,\hat{+}\, \boldsymbol{o} \,\hat{\times}\, \boldsymbol{o} \;=\; ?$$

Principles of Abstract Interpretation

- By contrast to testing, static analysis can prove the absence of bugs:

```
void main(int x){
    y = x * 12 + 9 * 11;
    assert (y % 2 == 1);
}
```

- Instead, static analysis may produce false alarms:

```
void main (int x) {
    y = x + x;
    assert (y % 2 == 0);
}
```

Principles of Abstract Interpretation

- **Question 4:** For precisely computing $v_1 + v_2$, we can extend the table like the following. Then, we can prove the query.

$$\top + \top = \begin{cases} e & \text{if } v_1 = v_2 \\ \top & \text{elif } v_1 \neq v_2 \\ \top & \text{else (don't know)} \end{cases}$$

Principles of Abstract Interpretation

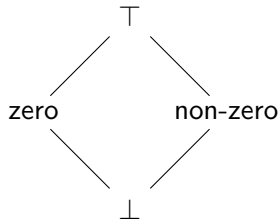
- **Quiz:** is there an abstract domain that can prove the safety of the following program?

```
divide(a, b) {
  return a / b;
}

main(x, y) {

  if (y == 0) {
    return -1;
  }

  z = divide(x, y);
  return 0;
}
```



Principles of Abstract Interpretation

- **Question:** If a correct program is given, does there always exist an abstract domain that can prove its correctness?