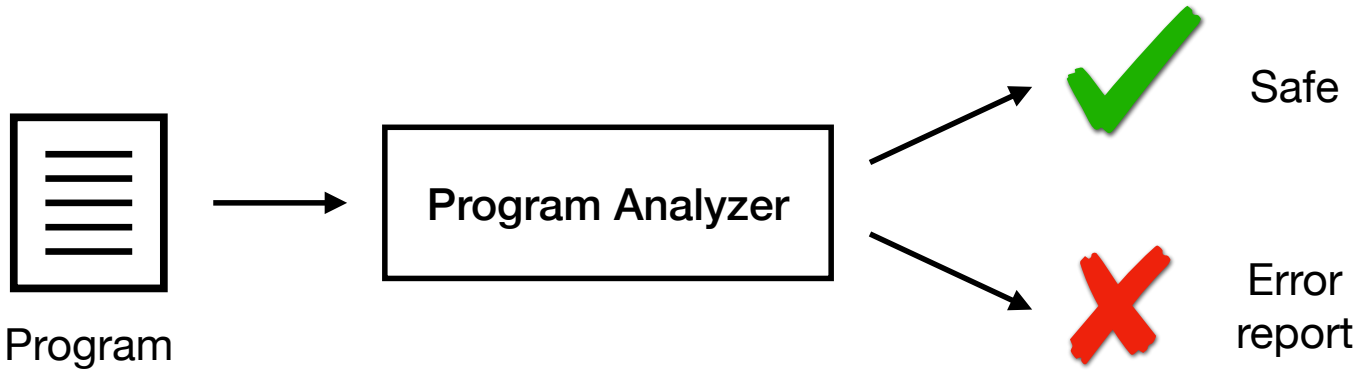
EC3204: Programming Languages and Compilers

Lecture 10 — Semantic Analysis (1)
Introduction

Sunbeom So
Fall 2024

Semantic Program Analysis



- Program analysis is a technology for discovering bugs or proving safety.
- Widely used to find SW vulnerabilities and bugs in the industry.
  - VCC (Microsoft), Danfy (AWS), SAGE (Microsoft), SMTChecker (Ethereum Foundation), JavaPathFinder (NASA), and many others.

Google Microsoft aws ethereum foundation NASA

An Ideal Analyzer: Sound and Complete

An automatic analyzer A is ideal iff it is **sound** and **complete**.

- **Soundness**:

  $$\text{for every program } p,\ A(p, \phi) = \texttt{safe} \implies p \text{ satisfies } \phi$$

  If p is proven to be safe by A, p is indeed safe.
  ($=$) If p is unsafe, A must return `unsafe` (contrapositive).
  ($=$) A misses no errors, i.e., A produces **no false negatives**.

An Ideal Analyzer: Sound and Complete

An automatic analyzer A is ideal iff it is **sound** and **complete**.

- **Soundness**:

  for every program p, $A(p, \phi) = \texttt{safe} \implies p$ satisfies ϕ

  If p is proven to be safe by A, p is indeed safe.
  ($=$) If p is unsafe, A must return `unsafe` (contrapositive).
  ($=$) A misses no errors, i.e., A produces **no false negatives**.

- **Completeness**:

  for every program p, $A(p, \phi) = \texttt{safe} \impliedby p$ satisfies ϕ

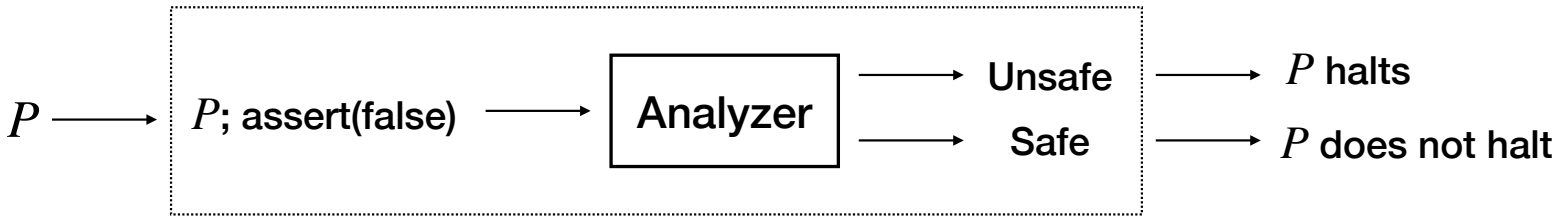
  If p is safe, A must prove its safety.
  ($=$) If A returns `unsafe`, p is unsafe (contrapositive).
  ($=$) A produces **no false positives**.

An ideal analyzer: No false negatives & No false positives

Hard Limit: Undecidability

We cannot have an ideal analyzer that can always produce a correct answer (`safe` or `unsafe`) for any program.

(Proof by Contradiction) Suppose exact analysis is possible. Then, we can solve the Halting problem!



Theorem (Rice Theorem)

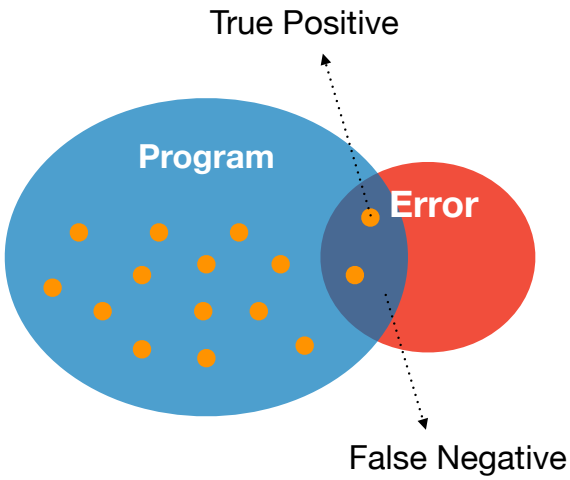
Let $\mathbb{L}$ be a Turing-complete language. Let ϕ be a nontrivial semantic property, i.e., there are $\mathbb{L}$ programs that satisfy ϕ and $\mathbb{L}$ programs that do not satisfy ϕ. There exists no algorithm $\boldsymbol{A}$ such that

$$\textit{for every program } \boldsymbol{p} \in \mathbb{L},\ \boldsymbol{A}(\boldsymbol{p}, \phi) = \texttt{true} \iff \boldsymbol{p} \textit{ satisfies } \phi.$$

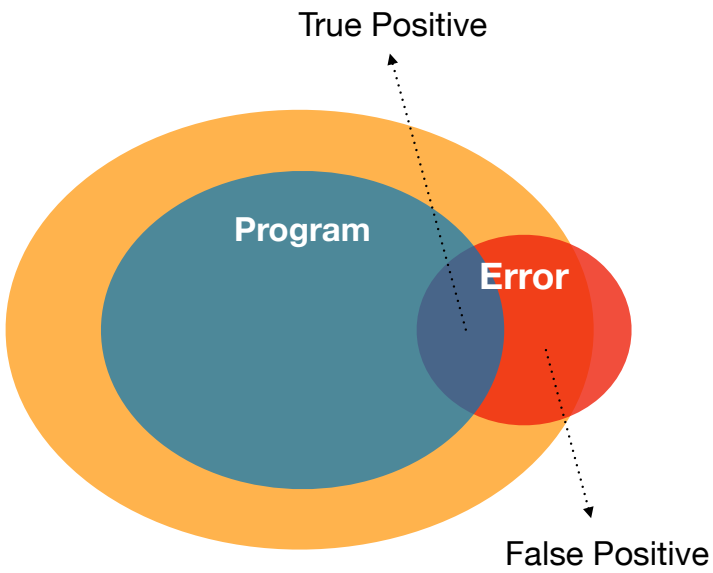
Side-Stepping Undecidability via Approximation

Any automatic analyzer compromises soundness or completeness.

	Under-approximation	**Over-approximiation**
Analysis Goal	complete but unsound	sound but incomplete
Examples	fuzzing, symbolic execution	formal verification, abstract interpretation



Under-approximation



Over-approximation

Fuzzing (Random Testing)

```
1  void testme (int x, int y) {
2    z = 2 * y;
3    if (z == x) {
4      if (x > y + 10) {
5        /* some error */
6      }
7    }
```

Suppose our goal is to find an input that triggers the error at line 5.

- Q. Test cases for reaching line 5?
  - ▶
- Q. Success probability of random testing? (assume $\mathbf{0 \leq x, y \leq 100}$)
  - ▶

Fuzzing (Random Testing)

```
1  void testme (int x, int y) {
2    z = 2 * y;
3    if (z == x) { /* x = 2 * y */
4      if (x > y + 10) { /* y > 10 */
5        /* some error */
6      }
7    }
```

Suppose our goal is to find an input that triggers the error at line 5.

- Q. Test cases for reaching line 5?
  - **(x,y)** : $(22, 11), (24, 12), \cdots, (100, 50)$
- Q. Success probability of random testing? (assume $0 \leq x, y \leq 100$)
  - 0.04%

Symbolic Execution

Testing methods that analyze program behavior based on logical formulas.

```
1  void testme (int x, int y) {
2    z = 2 * y;
3    if (z == x) {
4      if (x > y + 10) {
5        /* some error */
6      }
7    }
```

1. Generate the constraint (α, β: symbolic inputs for `x` and `y`).

$$(x = \alpha) \wedge (y = \beta) \wedge (z = 2 * y) \wedge (z = x) \wedge (x > y + 10)$$

2. Solve the constraint using an SMT solver.[1]

$$[x \mapsto 22, y \mapsto 11, z \mapsto 22, \underline{\alpha \mapsto 22, \beta \mapsto 11}]$$

[1] E.g., Z3 – https://github.com/Z3Prover/z3

Formal Verification

Symbolic execution with **invariant inference**.

```
1  @pre: n ≥ 0 /* precondition: assumed to be true at the entry */
2  @post: j = n /* postcondition: expected to be true at the exit */
3  void testme (int n) {
4    int i := 0;
5    int j := 0;
6    while (i ≠ n) {
7      i := i + 1;
8      j := j + 1;
9    }
10   return;
11 }
```

- Fuzzing and symbolic execution cannot prove the postcondition.
- To prove it, we must compute the **loop invariant** that summarizes the behavior of infinite iterations.

$$\underline{i = j} \wedge i = n \rightarrow j = n$$

Verification Challenge: Invariant Inference

```
1   @pre: ⊤
2   @post: sorted(rv, 0, |rv| − 1)
3   bool BubbleSort (int a[]) {
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)
                ∧ partitioned(a, 0, j − 1, j, j)
                ∧ sorted(a, i, |a| − 1) }
8       for (int j := 0; j < i; j := j + 1) {
9         if (a[j] > a[j + 1]) {
10          int t := a[j];
11          a[j] := a[j + 1];
12          a[j + 1] := t;
13        }
14      }
15    }
16    return a;
17  }
```

$$\textbf{sorted}(a, l, u) \iff \forall i, j. l \leq i \leq j \leq u \rightarrow a[i] \leq a[j]$$

$$\textbf{partitioned}(a, l_1, u_1, l_2, u_2) \iff \forall i, j. l_1 \leq i \leq u_1 < l_2 \leq j \leq u_2 \rightarrow a[i] \leq a[j]$$

Abstract Interpretation

Execute the program with abstract inputs.

```
1 void testme (int x) {
2   y = x * 12 + 9 * 11;
3   assert (y % 2 == 1); /* Goal: prove the safety */
4 }
```

- Underapproximation-based approaches cannot prove the safety.
- Verification approaches can prove the safety, but exact symbolic encoding on realistic programs is too expensive.
- We can prove the assertion using more lightweight reasoning.

$$\text{even number} + \text{odd number} = \text{odd number}$$

Abstract Interpretation

Of course, abstract interpretation is not a panacea.

```
1  void testme (int x) {
2    y = x;
3    y = y + x + 1;
4    assert (y % 2 == 1); /* false alarm */
5  }
```

$$\text{any number} + \text{any number} + \text{odd number} = \text{any number}$$

Summary

- Achieving an *ideal* analyzer is undecidable, i.e., impossible.
- Principles of program analysis: approximate behavior.

	Under-approximation	**Over-approximiation**
Analysis Goal	complete but unsound	sound but incomplete
Examples	fuzzing, symbolic execution	formal verification, abstract interpretation

- In this course, we will focus on **abstract interpretation**, as it is arguably the most common and cost-effective approach in compilers.
- If you are interested in other analysis approaches, consider joining our lab or taking my Software Engineering course.