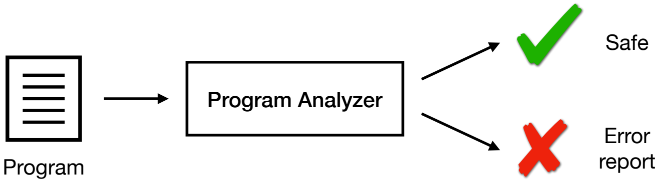
EC4219: Software Engineering

Lecture 1 — Introduction to Program Analysis

Sunbeom So
2024 Spring

Review: Program Analysis



Program analysis is the process of automatically discovering useful facts about programs. Examples include:

- Verification: is this program correct with respect to specifications?
- Bug-finding: does this program have integer-overflow bugs?
- Equivalence: are two programs semantically equivalent?
- Compiler optimization: Does the optimized program preserve the semantics of the original one?
- Many others

Review: Types of Program Analysis

Program analysis techniques can be broadly classified into three kinds.

- Dynamic analysis: the class of run-time analyses. These analyses discover information by running the program and observing its behavior.
- Static analysis: the class of compile-time analyses. These analyses discover information by inspecting the source code or binary code.
- Hybrid analysis: combines aspects of both dynamic and static analyses, by incorporating runtime and compile-time information in certain ways.

To understand the difference between the dynamic and static analysis, let's take a look at an example.

Program Invariant

```
1  int p(int x) { return x*x; }
2  void main () {
3    int z;
4    if (getc() == ‘a’)  z = p(6) + 6;
5    else  z = p(-7) - 7;
6    if (z != 42)
7      /* some error */
```

Let `getc()` be the function that reads a character from a user input.

- Q. To trigger the error at line 7, which branch should we take?

Program Invariant

```
1  int p(int x) { return x*x; }
2  void main () {
3    int z;
4    if (getc() == 'a')  z = p(6) + 6;
5    else  z = p(-7) - 7;
6    if (z != 42)
7      /* some error */
```

Let getc() be the function that reads a character from a user input.

- Q. To trigger the error at line 7, which branch should we take?
- A. The error will never happen! $z = \mathbf{42}$ is an *invariant* after line 4 and 5.

Program Invariant

```c
int p(int x) { return x*x; }
void main () {
  int z;
  if (getc() == ‘a’)  z = p(6) + 6;
  else  z = p(-7) - 7;
  if (z != 42)
    /* some error */
```

- Invariant: a fact that is true in every run of the program.
  - true branch: $p(6) + 6 = 6 * 6 + 6 = 36 + 6 = 42$
  - false branch: $p(-7) - 7 = (-7) * (-7) - 7 = 49 - 7 = 42$
- Inferring hidden invariants helps to prove the safety.

How do dynamic and static analyses work to discover invariants?

Dynamic Analysis

```
int p(int x) { return x*x; }
void main () {
  int z;
  if (getc() == ‘a’)  z = p(6) + 6;
  else  z = p(-7) - 7;
  if (z != 42)
    /* some error */
```

- In general, programs can have an unbounded number of paths due to recursion and loops.

Dynamic Analysis

```
int p(int x) { return x*x; }
void main () {
  int z;
  if (getc() == ‘a’)  z = p(6) + 6;
  else  z = p(-7) - 7;
  if (z != 42)
    /* some error */
```

- In general, programs can have an unbounded number of paths due to recursion and loops.
- Since dynamic analyzers discover information by running the program a finite number of times, they cannot prove some properties.
- They can at best *conjecture* that $z = 42$ is an invariant.

Dynamic Analysis

```
int p(int x) { return x*x; }
void main () {
  int z;
  if (getc() == ‘a’)  z = p(6) + 6;
  else  z = p(-7) - 7;
  if (z != 42)
    /* some error */
```

- In general, programs can have an unbounded number of paths due to recursion and loops.
- Since dynamic analyzers discover information by running the program a finite number of times, they cannot prove some properties.
- They can at best *conjecture* that $z = 42$ is an invariant.
- Q. Are they useless?

Dynamic Analysis

```
int p(int x) { return x*x; }
void main () {
  int z;
  if (getc() == ‘a’)  z = p(6) + 6;
  else  z = p(-7) - 7;
  if (z != 42)
    /* some error */
```

- In general, programs can have an unbounded number of paths due to recursion and loops.
- Since dynamic analyzers discover information by running the program a finite number of times, they cannot prove some properties.
- They can at best *conjecture* that $z = \mathbf{42}$ is an invariant.
- Q. Are they useless?
- No. They can *disprove* some properties (e.g., $z = \mathbf{30}$ is not an invariant).

Static Analysis

```
int p(int x) { return x*x; }
void main () {
  int z;
  if (getc() == ‘a’)  z = p(6) + 6;
  else  z = p(-7) - 7;
  if (z != 42)
    /* some error */
```

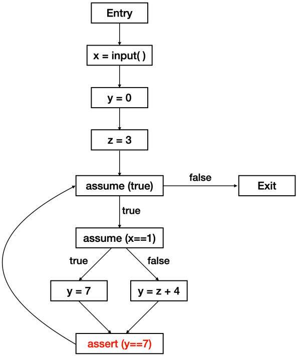
- By contrast, static analyses can discover invariants at compile-time.
- They can *prove* that $z = \mathbf{42}$ is an invariant, and can conclude that the program is safe.
- They are essential for safety-critical software.
  - E.g., Astrée – a static analyzer for aircraft software. Used by Airbus since 2003.
- They work by executing with *abstract* values.

Preliminary: Control-Flow Graph

```
void main() {
  x = input();
  y = 0;
  z = 3;
  while (true) {
    if (x == 1)
      y = 7;
    else
      y = z + 4;
    /* Goal: prove the assertion */
    assert (y == 7);
  }
}
```

Source code

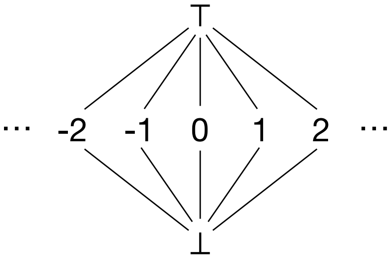
Preprocessing



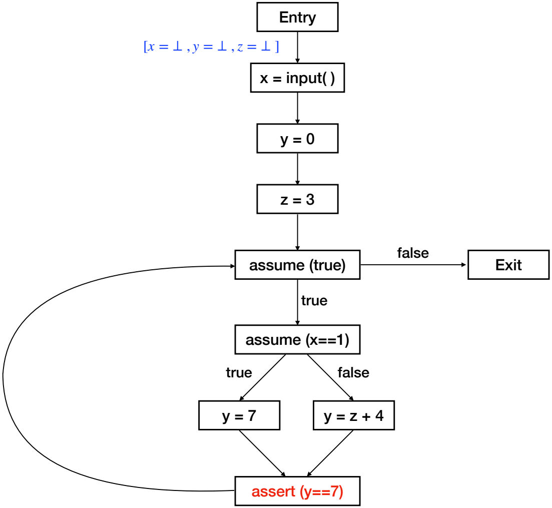
Control-Flow Graph

- Static analysis typically operates on a suitable intermediate representation of the program, called control-flow graph (CFG).
- CFG is a directed graph that summarizes the flow of control in all possible runs of the program.
  - Node: a unique atomic statement, Edge: a possible flow between nodes.

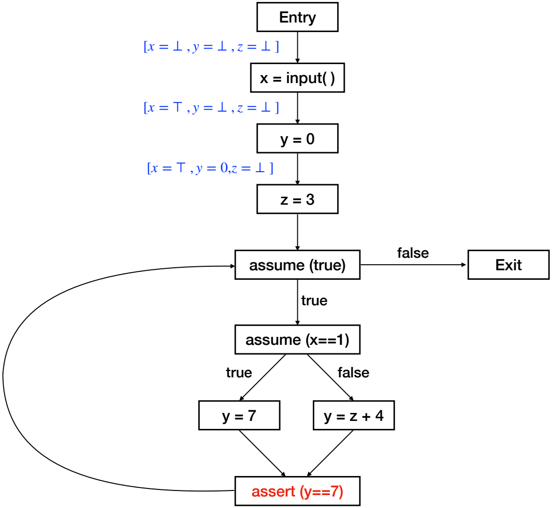
- The abstract domain shows the possible abstract values of variables.
- In our example, let's assume that there are three kinds of abstract values.
  - $\top$ (Top): values unknown to the analysis
  - $\cdots, \mathbf{-2, -1, 0, 1, 2,} \cdots$: integer constants
  - $\bot$ (bottom): the value undefined by the analysis (e.g., uninitialized variables)
- The order between abstract values is defined as follows.



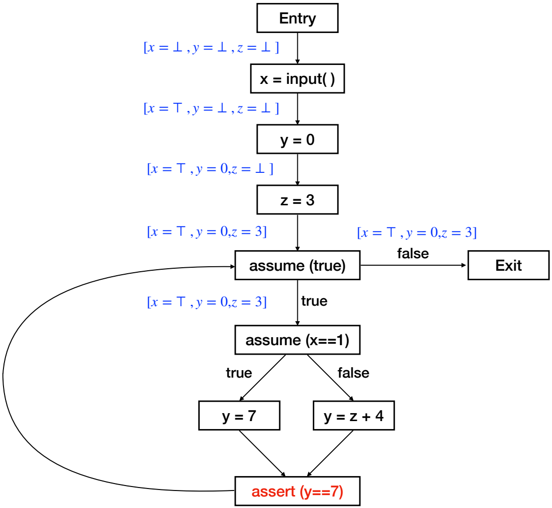
How Static Analysis Works (1)



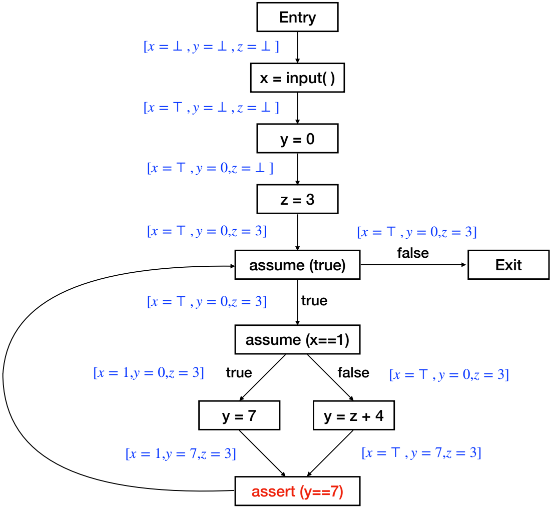
How Static Analysis Works (2)



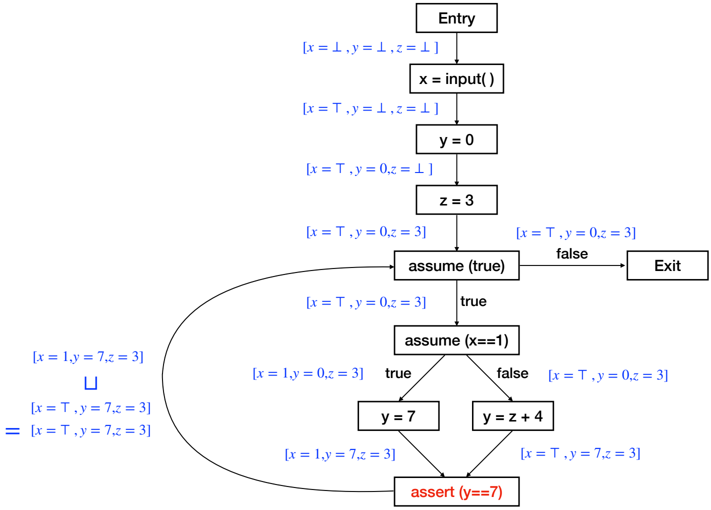
How Static Analysis Works (3)



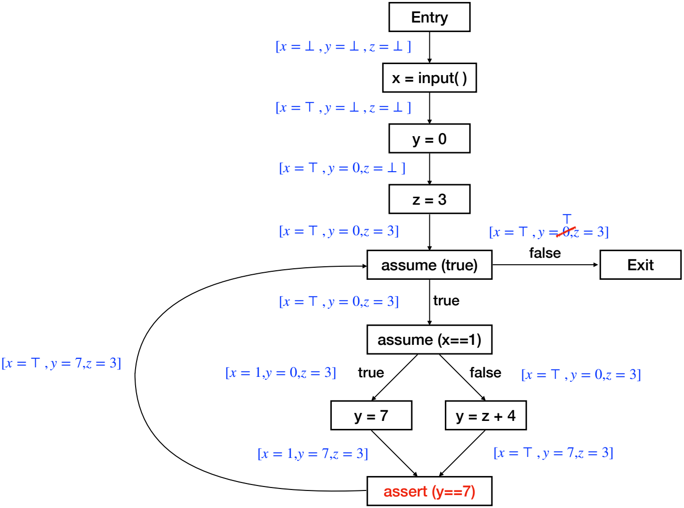
How Static Analysis Works (4)



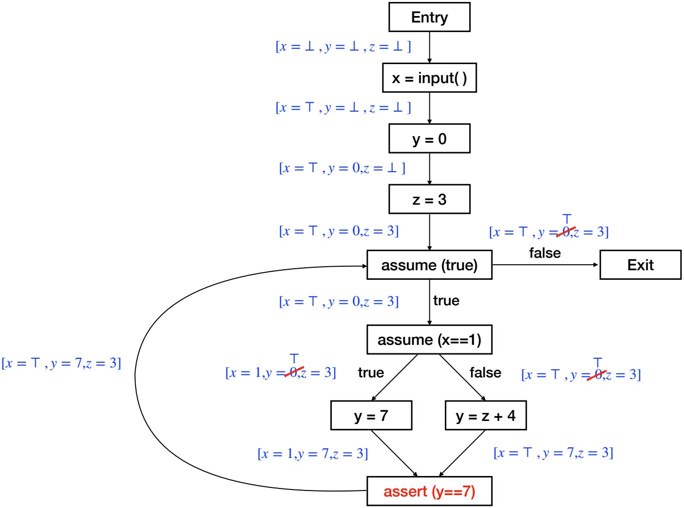
How Static Analysis Works (5)



How Static Analysis Works (6) – 2nd iteration



How Static Analysis Works (7) – 2nd iteration



Summary: How Static Analysis Works

- Program representation: e.g., control-flow graph
- Abstract domain: how to approximate program values
- Semantic functions: how to treat each assignment to produce resulting abstract states
- Fixed-point computation algorithm: terminates when the abstract states are no longer changing

Summary

We have briefly looked into how dynamic and static analysis works.

- Dynamic analysis: discover information by running the program a finite number of times, so cannot prove given properties.
- Static analysis: execute programs with abstract values, and can discover invariants at compile-time.
- Each of them has own its strengths.