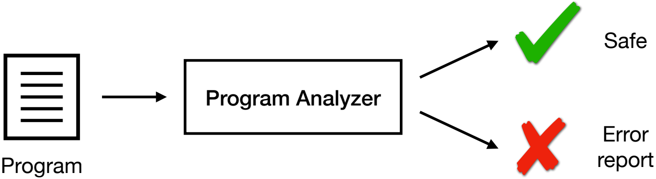
EC4219: Software Engineering

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.

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.

You might have several questions at this point.

- Why different kinds of analyses? Do they complement each other?
- More fundamentally, is there an ideal analyzer that can always output the correct results for any program?

Lecture's Contents

- Achieving an *ideal* analyzer is impossible (undecidable).
- The practical design choices of analyzers to address the undecidability.

An Ideal Analyzer: Sound and Complete

We say that an automatic analyzer A is ideal iff it is *sound* and *complete*.

- **Soundness**:

  for every program p, $A(p) = \texttt{true}^{1} \implies p$ satisfies ϕ

  All programs accepted by A satisfy ϕ.

[1] can be understood as “safe”

An Ideal Analyzer: Sound and Complete

We say that an automatic analyzer A is ideal iff it is *sound* and *complete*.

- **Soundness**:

  $$\text{for every program } p,\ A(p) = \texttt{true}^{1} \implies p \text{ satisfies } \phi$$

  All programs accepted by A satisfy ϕ.

  (=) All programs that violate ϕ must be rejected (contrapositive).

[1] can be understood as "safe"

An Ideal Analyzer: Sound and Complete

We say that an automatic analyzer A is ideal iff it is *sound* and *complete*.

- **Soundness**:

  $$\text{for every program } p,\ A(p) = \texttt{true}^{1} \implies p \text{ satisfies } \phi$$

  All programs accepted by A satisfy ϕ.
  $(=)$ All programs that violate ϕ must be rejected (contrapositive).
  $(=)$ A does not miss any SW errors.

[1] can be understood as “safe”

An Ideal Analyzer: Sound and Complete

We say that an automatic analyzer A is ideal iff it is *sound* and *complete*.

- **Soundness**:

  $$\text{for every program } p,\ A(p) = \texttt{true}^{1} \implies p \text{ satisfies } \phi$$

  All programs accepted by A satisfy ϕ.

  (=) All programs that violate ϕ must be rejected (contrapositive).

  (=) A does not miss any SW errors.

  (=) **No false negatives.**

[1] can be understood as "safe"

An Ideal Analyzer: Sound and Complete

We say that an automatic analyzer $\boldsymbol{A}$ is ideal iff it is *sound* and *complete*.

- **Soundness**:

  $$\text{for every program } \boldsymbol{p},\ \boldsymbol{A}(\boldsymbol{p}) = \texttt{true}^{1} \implies \boldsymbol{p} \text{ satisfies } \boldsymbol{\phi}$$

  All programs accepted by $\boldsymbol{A}$ satisfy $\boldsymbol{\phi}$.

  $(=)$ All programs that violate $\boldsymbol{\phi}$ must be rejected (contrapositive).

  $(=)$ $\boldsymbol{A}$ does not miss any SW errors.

  $(=)$ **No false negatives.**

- **Completeness**:

  $$\text{for every program } \boldsymbol{p},\ \boldsymbol{A}(\boldsymbol{p}) = \texttt{true} \impliedby \boldsymbol{p} \text{ satisfies } \boldsymbol{\phi}$$

  $\boldsymbol{A}$ accepts all programs that satisfy $\boldsymbol{\phi}$.

[1] can be understood as "safe"

An Ideal Analyzer: Sound and Complete

We say that an automatic analyzer A is ideal iff it is *sound* and *complete*.

- **Soundness**:

  $$\text{for every program } p,\ A(p) = \texttt{true}^{1} \implies p \text{ satisfies } \phi$$

  All programs accepted by A satisfy ϕ.

  $(=)$ All programs that violate ϕ must be rejected (contrapositive).

  $(=)$ A does not miss any SW errors.

  $(=)$ **No false negatives.**

- **Completeness**:

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

  A accepts all programs that satisfy ϕ.

  $(=)$ All programs rejected by A must violate ϕ (contrapositive).

[1] can be understood as "safe"

An Ideal Analyzer: Sound and Complete

We say that an automatic analyzer A is ideal iff it is *sound* and *complete*.

- **Soundness**:

  $$\text{for every program } p,\ A(p) = \texttt{true}^{1} \implies p \text{ satisfies } \phi$$

  All programs accepted by A satisfy ϕ.

  $(=)$ All programs that violate ϕ must be rejected (contrapositive).

  $(=)$ A does not miss any SW errors.

  $(=)$ **No false negatives.**

- **Completeness**:

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

  A accepts all programs that satisfy ϕ.

  $(=)$ All programs rejected by A must violate ϕ (contrapositive).

  $(=)$ A does not result in any false alarms.

[1] can be understood as "safe"

An Ideal Analyzer: Sound and Complete

We say that an automatic analyzer $\boldsymbol{A}$ is ideal iff it is *sound* and *complete*.

- **Soundness**:

  $$\text{for every program } p,\ A(p) = \texttt{true}^{1} \implies p \text{ satisfies } \phi$$

  All programs accepted by $\boldsymbol{A}$ satisfy ϕ.

  (=) All programs that violate ϕ must be rejected (contrapositive).

  (=) $\boldsymbol{A}$ does not miss any SW errors.

  (=) **No false negatives.**

- **Completeness**:

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

  $\boldsymbol{A}$ accepts all programs that satisfy ϕ.

  (=) All programs rejected by $\boldsymbol{A}$ must violate ϕ (contrapositive).

  (=) $\boldsymbol{A}$ does not result in any false alarms.

  (=) **No false positives.**

[1] can be understood as "safe"

An Ideal Analyzer: Sound and Complete

We say that an automatic analyzer A is ideal iff it is *sound* and *complete*.

- **Soundness**:

  $$\text{for every program } p,\ A(p) = \texttt{true}^{1} \implies p \text{ satisfies } \phi$$

  All programs accepted by A satisfy ϕ.
  (=) All programs that violate ϕ must be rejected (contrapositive).
  (=) A does not miss any SW errors.
  (=) **No false negatives.**

- **Completeness**:

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

  A accepts all programs that satisfy ϕ.
  (=) All programs rejected by A must violate ϕ (contrapositive).
  (=) A does not result in any false alarms.
  (=) **No false positives.**

An ideal analyzer: No false negatives & No false positives

[1] can be understood as "safe"

Hard Limit: Undecidability

In general, we cannot have an ideal analyzer that can exactly answer either `true` or `false` for any program. Formally,

Theorem (Rice Theorem)

Let $\mathbb{L}$ be a Turing-complete language, and let ϕ be a nontrivial semantic property of programs of $\mathbb{L}$. There exists no algorithm $\boldsymbol{A}$ such that

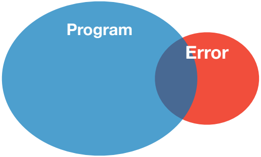
$$\text{for every program } \boldsymbol{p} \in \mathbb{L},\ \boldsymbol{A}(\boldsymbol{p}) = \texttt{true} \iff \boldsymbol{p} \text{ satisfies } \phi.$$

- cf1) We say that a semantic property ϕ is *nontrivial* when there are programs that satisfy ϕ and programs that do not satisfy ϕ.
- cf2) Since termination is an example of nontrivial semantic properties, the undecidability of the halting problem can be viewed as an instance of the Rice theorem.

Side-Stepping Undecidability via Approximation

Any automatic analyzer compromises soundness or completeness (knowingly or unknowingly).

	Static	**Dynamic**
Design Goal	(typically)sound but incomplete	complete but unsound
How?	over-approximation	under-approximation

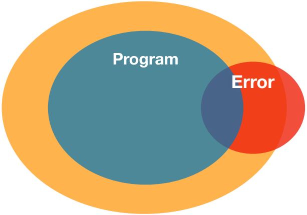


Static Analysis

Side-Stepping Undecidability via Approximation

Any automatic analyzer compromises soundness or completeness (knowingly or unknowingly).

	Static	**Dynamic**
Design Goal	(typically)sound but incomplete	complete but unsound
How?	over-approximation	under-approximation

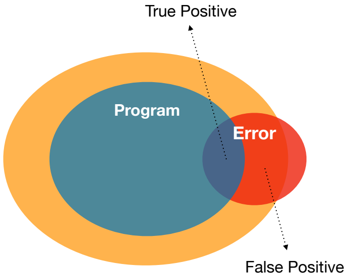


Static Analysis

Side-Stepping Undecidability via Approximation

Any automatic analyzer compromises soundness or completeness (knowingly or unknowingly).

	Static	**Dynamic**
Design Goal	(typically)sound but incomplete	complete but unsound
How?	over-approximation	under-approximation

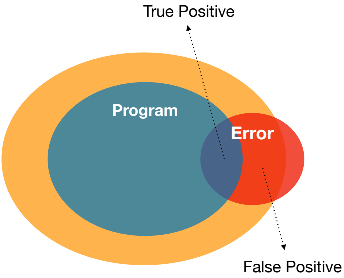


Static Analysis

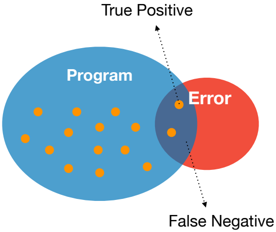
Side-Stepping Undecidability via Approximation

Any automatic analyzer compromises soundness or completeness (knowingly or unknowingly).

	Static	**Dynamic**
Design Goal	(typically)sound but incomplete	complete but unsound
How?	over-approximation	under-approximation



Static Analysis

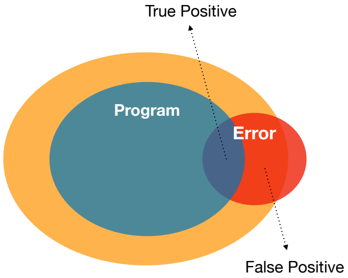


Dynamic Analysis

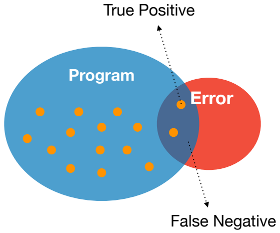
Side-Stepping Undecidability via Approximation

Any automatic analyzer compromises soundness or completeness (knowingly or unknowingly).

	Static	**Dynamic**
Design Goal	(typically)sound but incomplete	complete but unsound
How?	over-approximation	under-approximation



Static Analysis



Dynamic Analysis

Let's understand static and dynamic analyses better using examples.

Example Code with 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?

Example Code with 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.

Example Code with Program Invariant

```
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: $p(-7) - 7 = (-7) * (-7) - 7 = 49 - 7 = 42$
- We can prove the safety with invariants.

How do dynamic and static analyses work?

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

```
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 */
```

- 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 (line 7 might be safe).

Dynamic Analysis

```
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 */
```

- 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 (line 7 might be safe).
- Q. Are they useless?

Dynamic Analysis

```
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 */
```

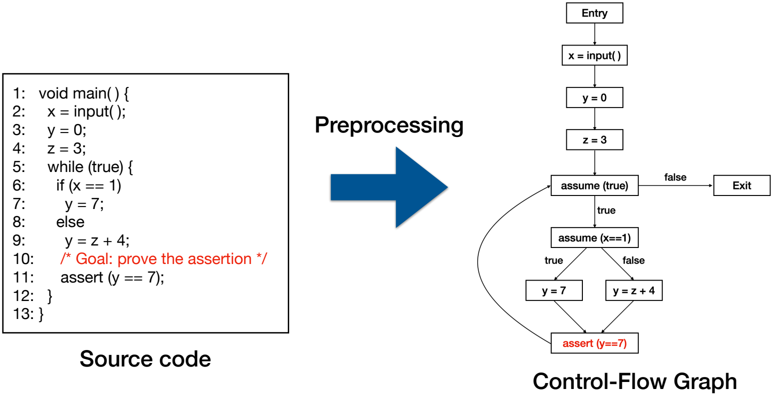
- In general, programs can have an unbounded number of paths due to recursion and loops.
- Since dynamic analyzers discover information by running the program a finite number of times, they cannot prove some properties.
- They can at best *conjecture* that $z = \mathbf{42}$ is an invariant (line 7 might be safe).
- Q. Are they useless?
- No. If line 6 is `z == 42`, they can *disprove* the safety.

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);
  }
}
```

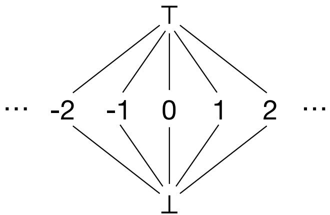
Source code

Control-Flow Graph

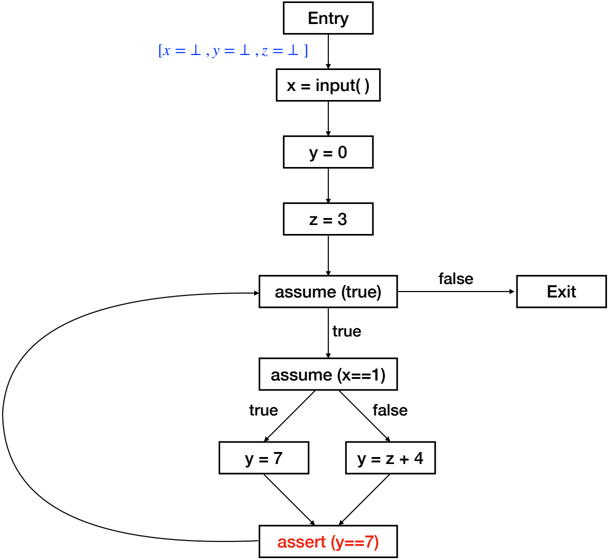
- Static analysis typically operates on a suitable intermediate representation of the program, called control-flow graph (CFG).
- CFG is a directed graph that summarizes the flow of control in all possible runs of the program.
  - Node: a unique atomic statement, Edge: a possible flow between nodes.

Preliminary: Abstract Domain

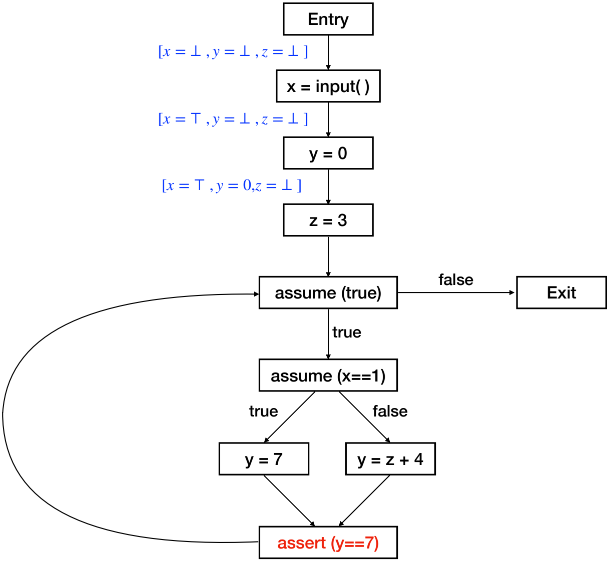
- The abstract domain shows the possible abstract values of variables.
- In our example, let's assume that there are three kinds of abstract values.
  - $\top$ (Top): values unknown to the analysis
  - $\cdots, \mathbf{-2}, \mathbf{-1}, \mathbf{0}, \mathbf{1}, \mathbf{2}, \cdots$: integer constants
  - $\bot$ (bottom): the value undefined by the analysis (e.g., uninitialized variables)
- The order between abstract values is defined as follows.
  - The value with a higher order means that it contains more information (e.g., $\mathbf{1} \sqsubseteq \top$, $\mathbf{1} \not\sqsubseteq \mathbf{2}$).



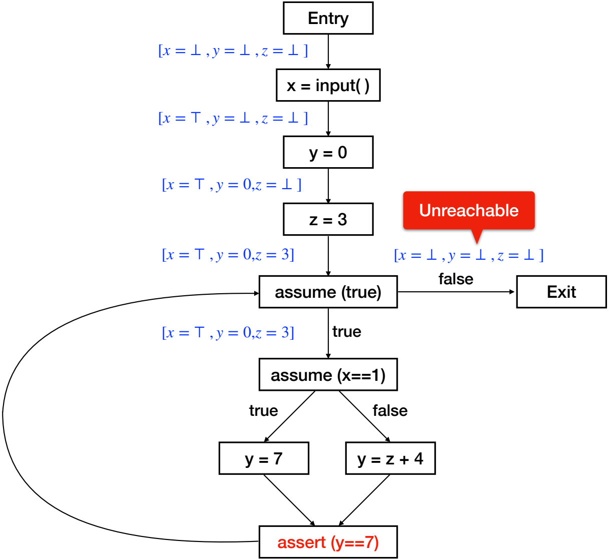
How Static Analysis Works (1)



How Static Analysis Works (2)

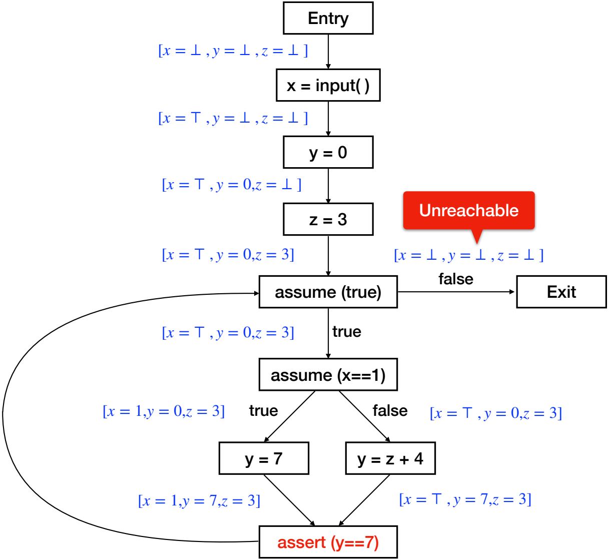


How Static Analysis Works (3)

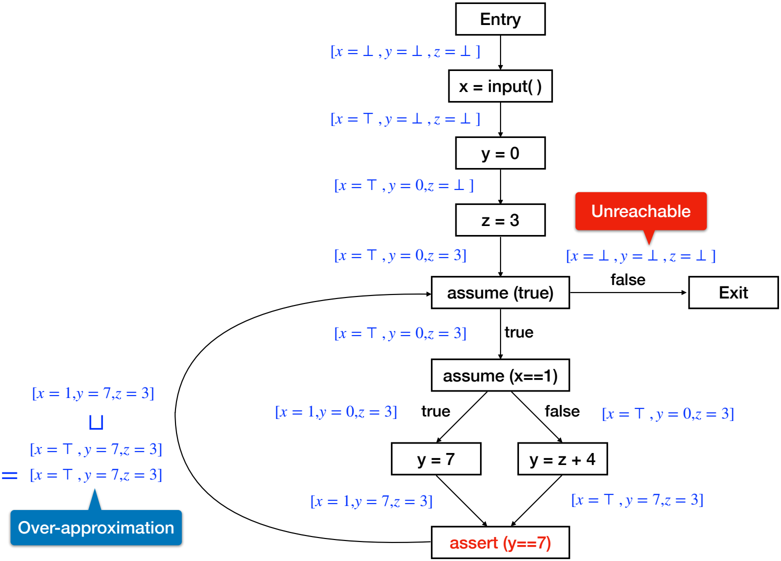


Entry
$[x = \bot, y = \bot, z = \bot]$
x = input( )
$[x = \top, y = \bot, z = \bot]$
y = 0
$[x = \top, y = 0, z = \bot]$
z = 3
$[x = \top, y = 0, z = 3]$
Unreachable
$[x = \bot, y = \bot, z = \bot]$
false
assume (true)
Exit
$[x = \top, y = 0, z = 3]$
true
assume (x==1)
true
false
y = 7
y = z + 4
assert (y==7)

How Static Analysis Works (4)


Entry
$[x = \bot, y = \bot, z = \bot]$
x = input( )
$[x = \top, y = \bot, z = \bot]$
y = 0
$[x = \top, y = 0, z = \bot]$
z = 3
$[x = \top, y = 0, z = 3]$
Unreachable
$[x = \bot, y = \bot, z = \bot]$
false
assume (true)
Exit
$[x = \top, y = 0, z = 3]$
true
assume (x==1)
$[x = 1, y = 0, z = 3]$
true
false
$[x = \top, y = 0, z = 3]$
y = 7
y = z + 4
$[x = 1, y = 7, z = 3]$
$[x = \top, y = 7, z = 3]$
assert (y==7)

How Static Analysis Works (5)



Entry

$[x = \bot, y = \bot, z = \bot]$

x = input( )

$[x = \top, y = \bot, z = \bot]$

y = 0

$[x = \top, y = 0, z = \bot]$

z = 3

$[x = \top, y = 0, z = 3]$

assume (true)

false

Exit

Unreachable

$[x = \bot, y = \bot, z = \bot]$

true

$[x = \top, y = 0, z = 3]$

assume (x==1)

true

$[x = 1, y = 0, z = 3]$

false

$[x = \top, y = 0, z = 3]$

y = 7

y = z + 4

$[x = 1, y = 7, z = 3]$

$[x = \top, y = 7, z = 3]$

assert (y==7)

$[x = 1, y = 7, z = 3]$

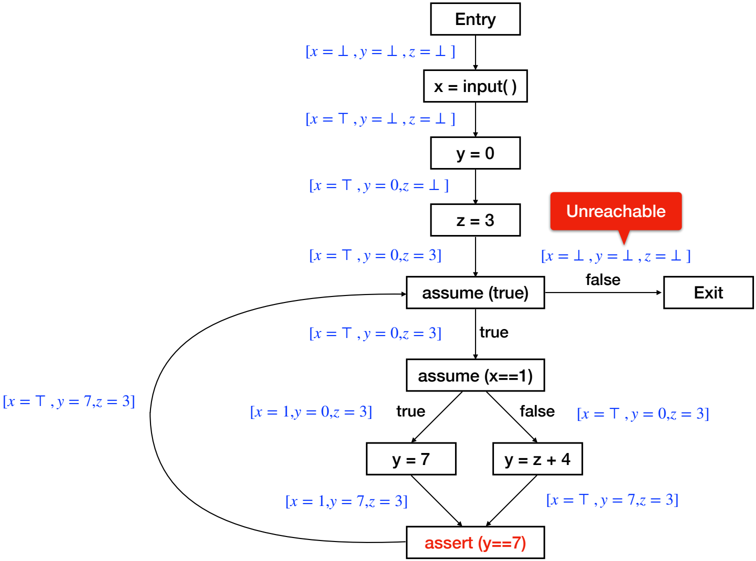
$\sqcup$

$[x = \top, y = 7, z = 3]$

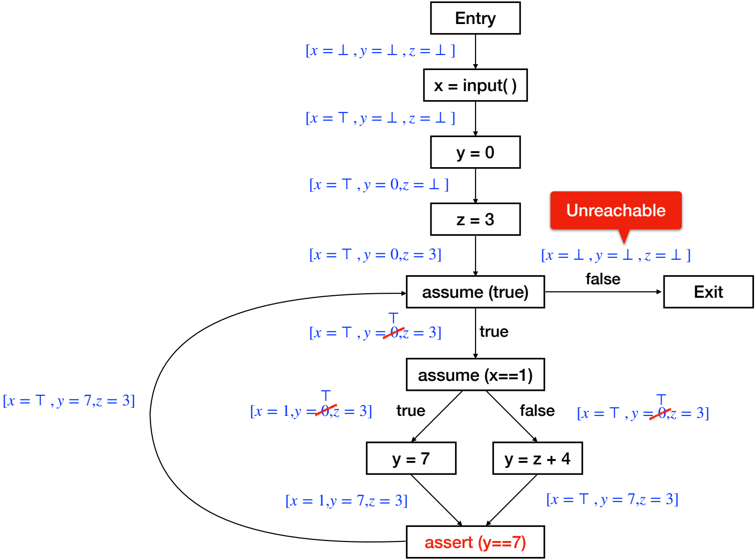
$= [x = \top, y = 7, z = 3]$

Over-approximation

How Static Analysis Works (6) – 2nd iteration


Entry
$[x = \bot, y = \bot, z = \bot]$
x = input( )
$[x = \top, y = \bot, z = \bot]$
y = 0
$[x = \top, y = 0, z = \bot]$
z = 3
$[x = \top, y = 0, z = 3]$
Unreachable
$[x = \bot, y = \bot, z = \bot]$
false
assume (true)
Exit
$[x = \top, y = 0, z = 3]$
true
assume (x==1)
$[x = \top, y = 7, z = 3]$
$[x = 1, y = 0, z = 3]$
true
false
$[x = \top, y = 0, z = 3]$
y = 7
y = z + 4
$[x = 1, y = 7, z = 3]$
$[x = \top, y = 7, z = 3]$
assert (y==7)

How Static Analysis Works (7) – 2nd iteration



Entry

$[x = \bot, y = \bot, z = \bot]$

x = input( )

$[x = \top, y = \bot, z = \bot]$

y = 0

$[x = \top, y = 0, z = \bot]$

z = 3

$[x = \top, y = 0, z = 3]$

Unreachable

$[x = \bot, y = \bot, z = \bot]$

assume (true)

false

Exit

$[x = \top, y = \overset{\top}{\cancel{0}}, z = 3]$

true

assume (x==1)

$[x = \top, y = 7, z = 3]$

$[x = 1, y = \overset{\top}{\cancel{0}}, z = 3]$

true

false

$[x = \top, y = \overset{\top}{\cancel{0}}, z = 3]$

y = 7

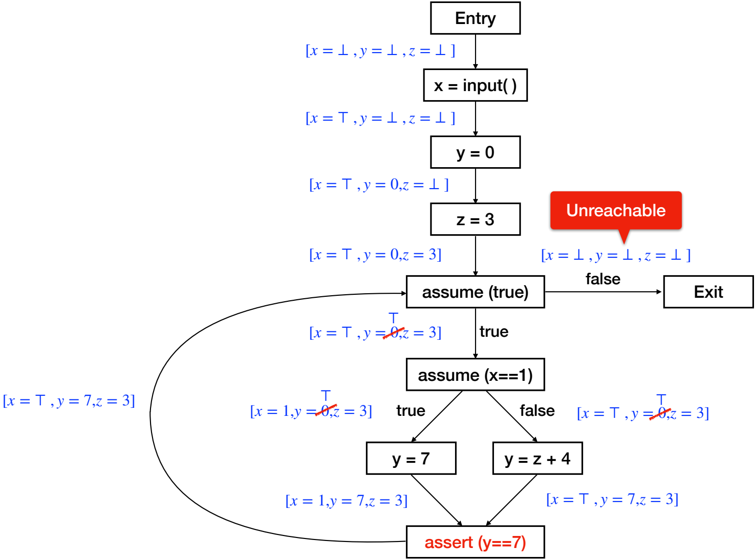
y = z + 4

$[x = 1, y = 7, z = 3]$

$[x = \top, y = 7, z = 3]$

assert (y==7)

How Static Analysis Works (8) – 3rd iteration



Entry

$[x = \bot, y = \bot, z = \bot]$

x = input( )

$[x = \top, y = \bot, z = \bot]$

y = 0

$[x = \top, y = 0, z = \bot]$

z = 3

$[x = \top, y = 0, z = 3]$

Unreachable

$[x = \bot, y = \bot, z = \bot]$

assume (true)

false

Exit

$[x = \top, y = \overset{\top}{\cancel{0}}, z = 3]$

true

assume (x==1)

$[x = \top, y = 7, z = 3]$

$[x = 1, y = \overset{\top}{\cancel{0}}, z = 3]$

true

false

$[x = \top, y = \overset{\top}{\cancel{0}}, z = 3]$

y = 7

y = z + 4

$[x = 1, y = 7, z = 3]$

$[x = \top, y = 7, z = 3]$

assert (y==7)

Summary: How Static Analysis Works

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

Summary: How Static Analysis Works

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

Q. In our example, is the termination guaranteed?

Summary: How Static Analysis Works

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

Q. In our example, is the termination guaranteed?

A. Yes. The abstract domain is finite, and (assuming) the semantic function is extensive ($x \sqsubseteq \hat{F}(x)$). ensure the termination of the fixed-point iteration?

Summary

- Achieving an *ideal* analyzer is impossible (undecidable).
- The practical design choices of analyzers to address the undecidability.
  - 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.