L4: Soundness and Completeness

Software Analysis

IIT Guwahati · August 4-5, 2026

Example

Finding variables that hold a constant at a program point

Can we prove the assertion — for *every* run, and every iteration?

```
void main() {
    x = user_input();
    z = 3;
    while (true) {
        if (x == 1) { y = 7; }
        else        { y = z + 4; }
        assert(y == 7);
    }
}
```

Example

Finding variables that hold a constant at a program point

Can we prove the assertion — for *every* run, and every iteration?

```
void main() {
    x = user_input();
    z = 3;
    while (true) {
        if (x == 1) { y = 7; }
        else        { y = z + 4; }
        assert(y == 7);
    }
}
```

Assertions of the form `variable == constant` are an instance of a classic static analysis problem:

*which variables hold a **constant value** at a given program point?*

Example

Finding variables that hold a constant at a program point

Can we prove the assertion — for *every* run, and every iteration?

```
void main() {
    x = user_input();
    z = 3;
    while (true) {
        if (x == 1) { y = 7; }
        else        { y = z + 4; }
        assert(y == 7);
    }
}
```

Assertions of the form `variable == constant` are an instance of a classic static analysis problem:

*which variables hold a **constant value** at a given program point?*

Note the loop: `while (true)` has an **unbounded** number of paths.

Terminology

1. **Control-flow graph (CFG)** — a graph representation of all possible flows of control in a program.

Terminology

1. **Control-flow graph (CFG)** — a graph representation of all possible flows of control in a program.
2. **Concrete** vs. **abstract** states — exact program executions versus finite abstractions used by the analysis.

Terminology

1. **Control-flow graph (CFG)** — a graph representation of all possible flows of control in a program.
2. **Concrete** vs. **abstract** states — exact program executions versus finite abstractions used by the analysis.
3. **Termination** — the analysis always completes, even if the analyzed program does not terminate.

Terminology

1. **Control-flow graph (CFG)** — a graph representation of all possible flows of control in a program.
2. **Concrete** vs. **abstract** states — exact program executions versus finite abstractions used by the analysis.
3. **Termination** — the analysis always completes, even if the analyzed program does not terminate.
4. **Completeness** — every true program property can be proved by the analysis.

Terminology

1. **Control-flow graph (CFG)** — a graph representation of all possible flows of control in a program.
2. **Concrete** vs. **abstract** states — exact program executions versus finite abstractions used by the analysis.
3. **Termination** — the analysis always completes, even if the analyzed program does not terminate.
4. **Completeness** — every true program property can be proved by the analysis.
5. **Soundness** — every property proved by the analysis is true for all program executions.

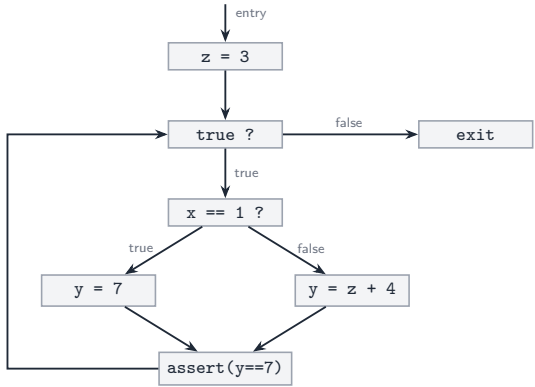
Terminology

1. **Control-flow graph (CFG)** — a graph representation of all possible flows of control in a program.
2. **Concrete** vs. **abstract** states — exact program executions versus finite abstractions used by the analysis.
3. **Termination** — the analysis always completes, even if the analyzed program does not terminate.
4. **Completeness** — every true program property can be proved by the analysis.
5. **Soundness** — every property proved by the analysis is true for all program executions.

Practical static analyses are typically **sound** and **terminating**, but not **complete**. Consequently, they may report **false positives** but should not miss real errors.

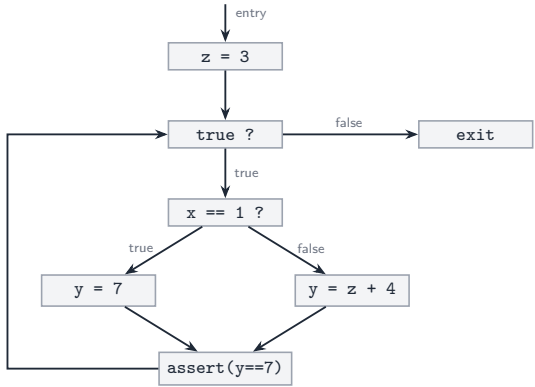
The Control-Flow Graph

A **directed graph** representing the possible flow of control in a program. Each node corresponds to a statement (or basic block), and each directed edge represents a possible transfer of control during execution.



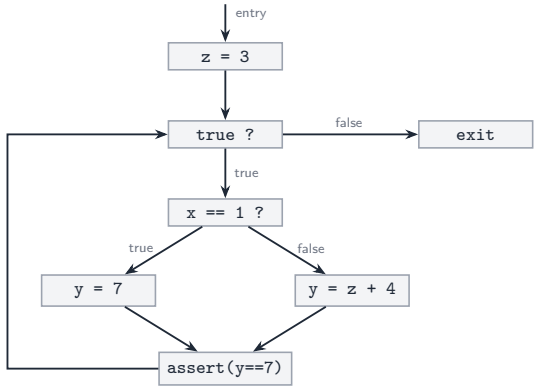
The Control-Flow Graph

A **directed graph** representing the possible flow of control in a program. Each node corresponds to a statement (or basic block), and each directed edge represents a possible transfer of control during execution.



The Control-Flow Graph

A **directed graph** representing the possible flow of control in a program. Each node corresponds to a statement (or basic block), and each directed edge represents a possible transfer of control during execution.



Concrete and Abstract States

Concrete State

- Exact program state during one execution.
- Example:

$$x = 5,\ y = 7,\ z = 3$$

- Defined by the program's **concrete semantics**.

Concrete and Abstract States

Concrete State

- Exact program state during one execution.
- Example:

$$x = 5,\ y = 7,\ z = 3$$

- Defined by the program's **concrete semantics**.

Abstract State

- Finite summary of a set of concrete states.
- Example:

$$x = ?,\ y = 7,\ z = 3$$

- Defined by the **abstract semantics** chosen for the analysis.

Concrete and Abstract States

Concrete State

- Exact program state during one execution.
- Example:

$$x = 5,\ y = 7,\ z = 3$$

- Defined by the program's **concrete semantics**.

Abstract State

- Finite summary of a set of concrete states.
- Example:

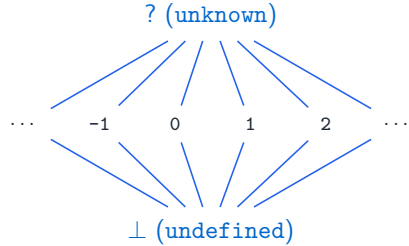
$$x = ?,\ y = 7,\ z = 3$$

- Defined by the **abstract semantics** chosen for the analysis.

A static analysis never observes concrete states directly. Instead, it computes over **abstract states**, which summarize all possible executions.

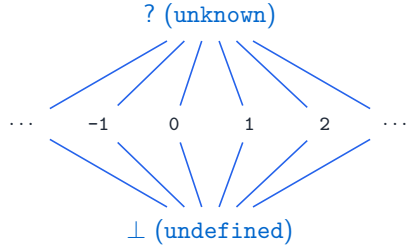
Example Abstract Domain

The **abstract domain** specifies the set of values used by the analysis. For our example, an integer variable may take any constant, together with two distinguished values: ? (unknown) and ⊥ (undefined).



Example Abstract Domain

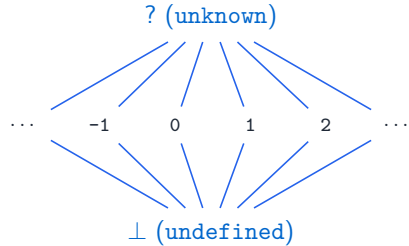
The **abstract domain** specifies the set of values used by the analysis. For our example, an integer variable may take any constant, together with two distinguished values: ? (unknown) and $\perp$ (undefined).



The ordering forms a **lattice**. During the analysis, abstract values move only upward: $\perp \rightarrow c \rightarrow ?$.

Example Abstract Domain

The **abstract domain** specifies the set of values used by the analysis. For our example, an integer variable may take any constant, together with two distinguished values: ? (unknown) and $\perp$ (undefined).



The ordering forms a **lattice**. During the analysis, abstract values move only upward: $\perp \rightarrow c \rightarrow ?$.

Every ascending chain has finite length. Therefore, each abstract value changes only finitely many times, guaranteeing termination of the analysis.

? and $\perp$: Unknown vs. Undefined

? — Unknown

- The analysis has reached this program point.
- No single constant can be determined.
- The variable may hold different values on different executions.

? and ⊥: Unknown vs. Undefined

? — Unknown

- The analysis has reached this program point.
- No single constant can be determined.
- The variable may hold different values on different executions.

⊥ — Undefined

- The variable has not yet been assigned a value.
- Typical at the beginning of the analysis.
- Represents the absence of information.

? and $\perp$: Unknown vs. Undefined

? — Unknown

- The analysis has reached this program point.
- No single constant can be determined.
- The variable may hold different values on different executions.

$\perp$ — Undefined

- The variable has not yet been assigned a value.
- Typical at the beginning of the analysis.
- Represents the absence of information.

During constant propagation, an abstract value moves only upward:

$$\perp \longrightarrow \text{constant} \longrightarrow ?$$

As information from different execution paths is merged, the analysis may lose precision but never gains information.

Iterative Approximation

We compute the analysis result using **iterative approximation**. Starting from an initial approximation, the abstract states are repeatedly updated until a **fixed point** is reached.

Iterative Approximation

We compute the analysis result using **iterative approximation**. Starting from an initial approximation, the abstract states are repeatedly updated until a **fixed point** is reached.

Initial state:

· Assign ? to x, y, and z at the **entry** of the program.

· Assign $\perp$ to **every other** program point.

Iterative Approximation

We compute the analysis result using **iterative approximation**. Starting from an initial approximation, the abstract states are repeatedly updated until a **fixed point** is reached.

Initial state:

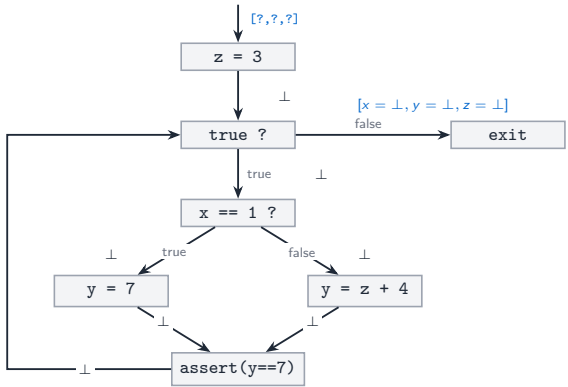
· Assign ? to x, y, and z at the **entry** of the program.

· Assign $\perp$ to **every other** program point.

The analysis propagates information through the control-flow graph. Abstract states are repeatedly updated until no further changes occur, yielding a fixed-point solution.

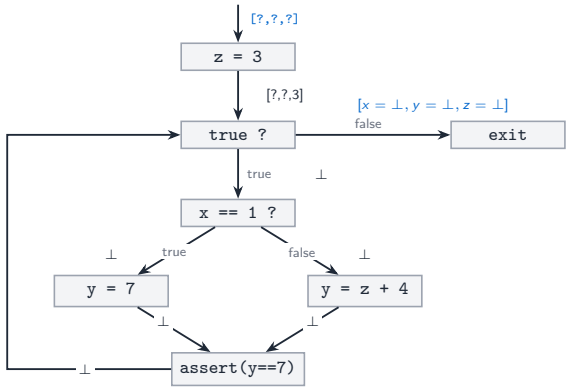
Iterative Approximation Example

Abstract states are written as `[x,y,z]`. Initially, only the entry program point is ?; every other program point is initialized to $\perp$.



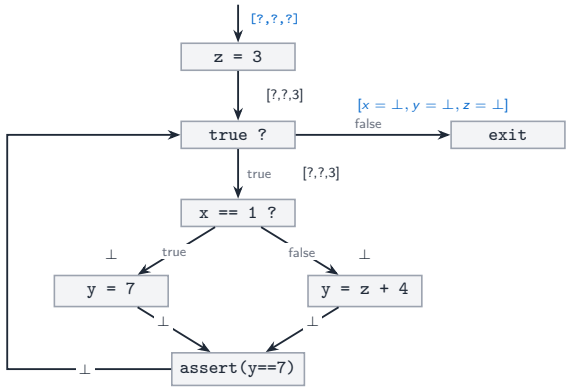
Iterative Approximation Example

Abstract states are written as `[x,y,z]`. Initially, only the entry program point is ?; every other program point is initialized to $\perp$.



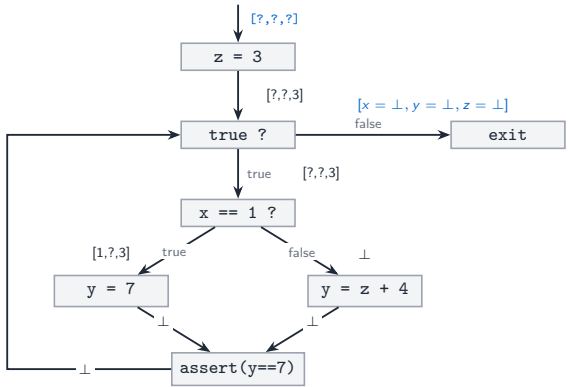
Iterative Approximation Example

Abstract states are written as `[x,y,z]`. Initially, only the entry program point is ?; every other program point is initialized to $\bot$.



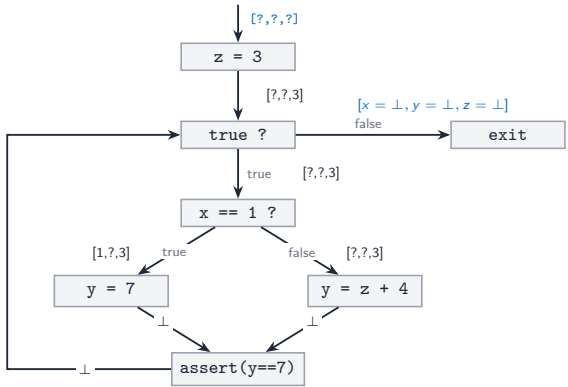
Iterative Approximation Example

Abstract states are written as `[x,y,z]`. Initially, only the entry program point is ?; every other program point is initialized to $\bot$.



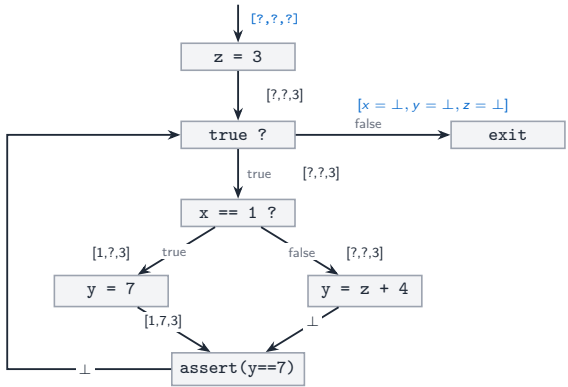
Iterative Approximation Example

Abstract states are written as `[x,y,z]`. Initially, only the entry program point is ?; every other program point is initialized to $\perp$.



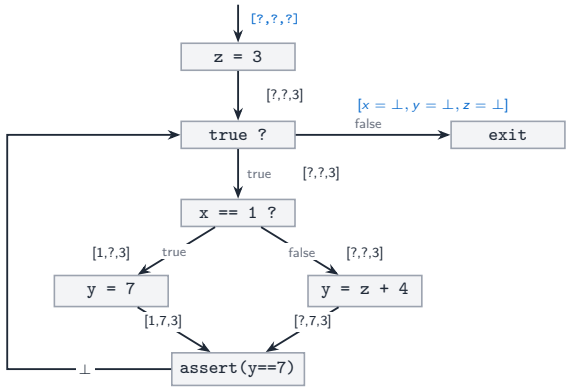
Iterative Approximation Example

Abstract states are written as `[x,y,z]`. Initially, only the entry program point is ?; every other program point is initialized to $\perp$.



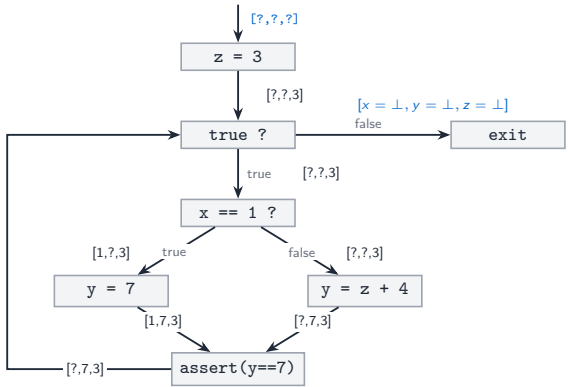
Iterative Approximation Example

Abstract states are written as `[x,y,z]`. Initially, only the entry program point is ?; every other program point is initialized to $\perp$.



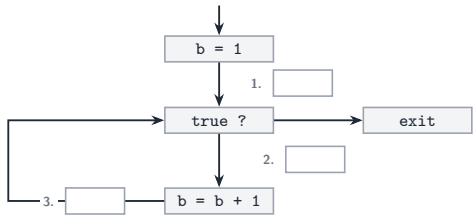
Iterative Approximation Example

Abstract states are written as `[x,y,z]`. Initially, only the entry program point is ?; every other program point is initialized to $\perp$.



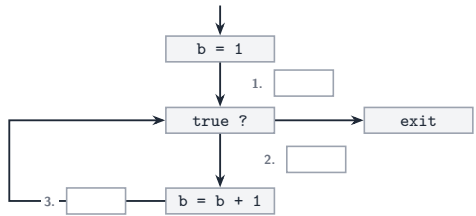
Quiz: Iterative Approximation

```
b = 1;
while (true) {
    b = b + 1;
}
```



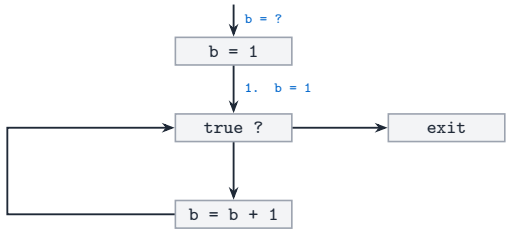
Quiz: Iterative Approximation

```
b = 1;
while (true) {
    b = b + 1;
}
```

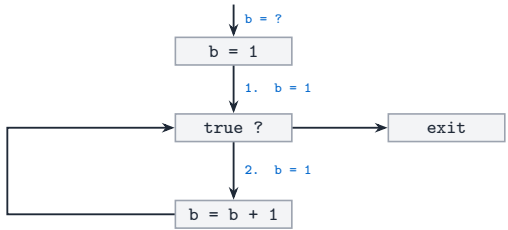


1. loop header **2.** entry of loop body **3.** exit of loop body

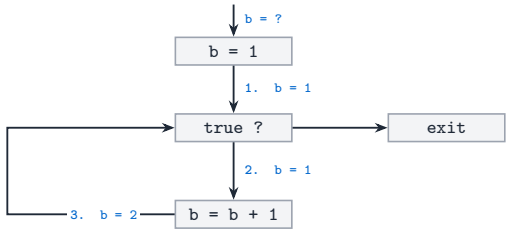
Solution: the First Pass



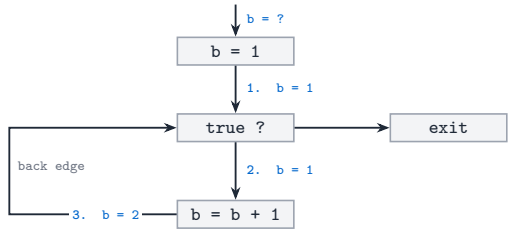
Solution: the First Pass



Solution: the First Pass

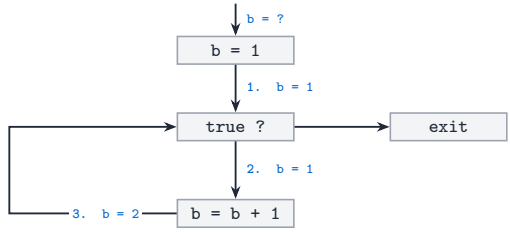


Solution: the First Pass

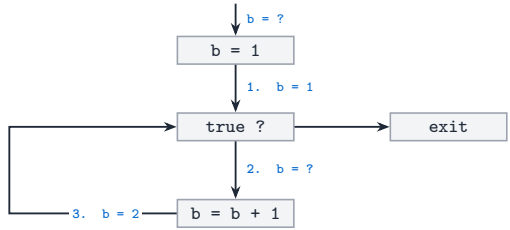


The back edge carries `b = 2` into the loop header.
The analysis must go round **again**.

Solution: the Second Pass and the Fixed Point

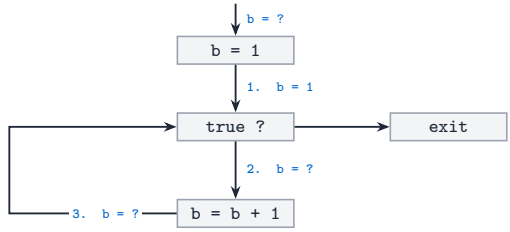


Solution: the Second Pass and the Fixed Point



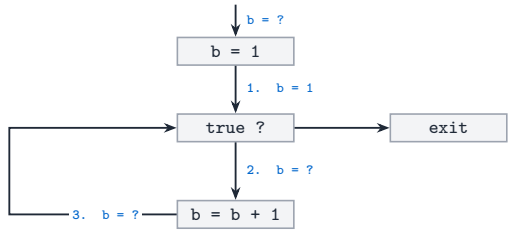
- At box 2 the analysis now sees 1 (from the header) *and* 2 (from the back edge) $\rightarrow$?.

Solution: the Second Pass and the Fixed Point



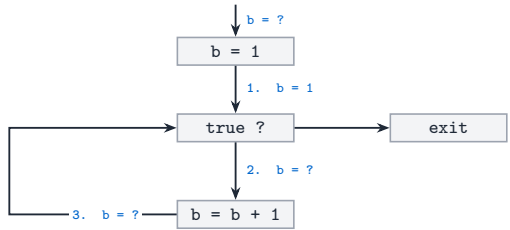
- At box 2 the analysis now sees 1 (from the header) *and* 2 (from the back edge) $\rightarrow$?.
- Box 3 follows: $? + 1 = ?$.

Solution: the Second Pass and the Fixed Point



- At box 2 the analysis now sees 1 (from the header) *and* 2 (from the back edge) $\rightarrow$?.
- Box 3 follows: $? + 1 = ?$.
- Third pass: **nothing changes**. The values have **saturated**. **Stop.**

Solution: the Second Pass and the Fixed Point



- At box 2 the analysis now sees 1 (from the header) *and* 2 (from the back edge) $\rightarrow$?.
- Box 3 follows: $? + 1 = ?$.
- Third pass: **nothing changes**. The values have **saturated**. **Stop.**

Answer: 1, ?, ?. This is a **fixed point** — the state where applying every transfer function changes nothing.

Merging: the Combination Operator

Where paths meet, abstract states must be **combined**. How is another design decision.

Conjunction ("must")

`b` = 1 only if `b` = 1 on **ALL** incoming paths.

Merge of 1 and 2 $\rightarrow$?.

Proves things. Used for *verification*.

Merging: the Combination Operator

Where paths meet, abstract states must be **combined**. How is another design decision.

Conjunction ("must")

`b = 1` only if `b = 1` on **ALL** incoming paths.

Merge of `1` and `2` $\rightarrow$?.

Proves things. Used for *verification*.

Disjunction ("may")

`b = 1` if `b = 1` on **AT LEAST ONE** path.

Answers "*can* `b` be `1` here?"

Finds things. Used for *bug hunting*, taint, aliasing.

Merging: the Combination Operator

Where paths meet, abstract states must be **combined**. How is another design decision.

Conjunction ("must")

`b = 1` only if `b = 1` on **ALL** incoming paths.

Merge of 1 and 2 $\rightarrow$?.

Proves things. Used for *verification*.

Disjunction ("may")

`b = 1` if `b = 1` on **AT LEAST ONE** path.

Answers "*can* `b` be 1 here?"

Finds things. Used for *bug hunting*, taint, aliasing.

Our example used **conjunction** — which is why box 2 became ?.
A "may" analysis would have answered *yes, `b` can be 1 here*.

Can an Abstract State Hold Two Values at Once?

At box 2 we knew $\mathtt{b}$ is **1 or 2**. Why write ? instead of $\{1, 2\}$?

Cousot & Cousot, POPL'77, §4 (widening) · SPA, §4.1, §6.1

Can an Abstract State Hold Two Values at Once?

At box 2 we knew `b` is **1 or 2**. Why write ? instead of $\{1, 2\}$?

- Because $\{1, 2\}$ **is not in our domain.** One variable maps to *exactly one* element of the lattice

Cousot & Cousot, POPL'77, §4 (widening) · SPA, §4.1, §6.1

Can an Abstract State Hold Two Values at Once?

At box 2 we knew `b` is **1 or 2**. Why write ? instead of $\{1, 2\}$?

- Because $\{1, 2\}$ **is not in our domain.** One variable maps to *exactly one* element of the lattice
- You *may* choose a richer domain — sets of constants, intervals `[1,2]` — and then `{1,2}` **is** expressible.

Cousot & Cousot, POPL'77, §4 (widening) · SPA, §4.1, §6.1

Can an Abstract State Hold Two Values at Once?

At box 2 we knew `b` is **1 or 2**. Why write ? instead of $\{1, 2\}$?

· Because $\{1, 2\}$ **is not in our domain.** One variable maps to *exactly one* element of the lattice

· You *may* choose a richer domain — sets of constants, intervals `[1,2]` — and then `{1,2}` **is** expressible.

· **But:** in this program `b` takes `1, 2, 3, ...` without bound. A set domain would grow forever and **never terminate**. Intervals would give $[1, \infty)$ — only with a **widening** operator to force convergence.

Cousot & Cousot, POPL'77, §4 (widening) · SPA, §4.1, §6.1

Can an Abstract State Hold Two Values at Once?

At box 2 we knew `b` is **1 or 2**. Why write ? instead of $\{1, 2\}$?

· Because $\{1, 2\}$ **is not in our domain.** One variable maps to *exactly one* element of the lattice

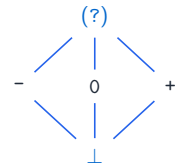
· You *may* choose a richer domain — sets of constants, intervals `[1,2]` — and then `{1,2}` **is** expressible.

· **But:** in this program `b` takes `1, 2, 3, ...` without bound. A set domain would grow forever and **never terminate**. Intervals would give $[1, \infty)$ — only with a **widening** operator to force convergence.

Cousot & Cousot, POPL'77, §4 (widening) · SPA, §4.1, §6.1

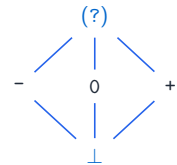
Divide-by-Zero: the Sign Domain Example

```
void main() {
    while (x > 0) {
S1:     ... 1/x ...
        if (x > 0) {
            x--;
        }
        else {
            x++;
        }
S2:     ... 1/x ...
    }
}
```



Divide-by-Zero: the Sign Domain Example

```
void main() {
    while (x > 0) {
S1:     ... 1/x ...
        if (x > 0) {
            x--;
        }
        else {
            x++;
        }
S2:     ... 1/x ...
    }
}
```

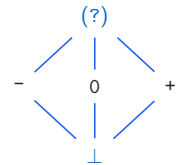


- 0 — the value zero
- − — any negative value
- + — any positive value
- ? — unknown

- ⊥ — undefined

Divide-by-Zero: the Sign Domain Example

```
void main() {
    while (x > 0) {
S1:     ... 1/x ...
        if (x > 0) {
            x--;
        }
        else {
            x++;
        }
S2:     ... 1/x ...
    }
}
```

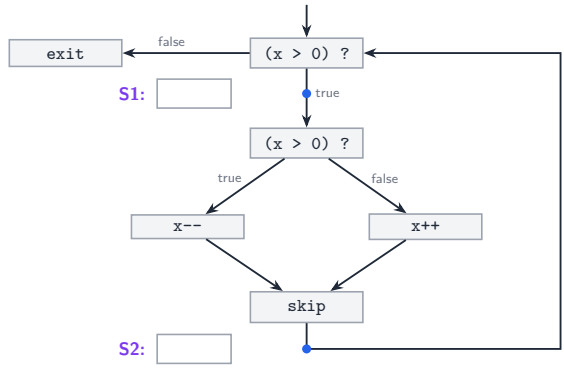


- 0 — the value zero
- − — any negative value
- + — any positive value
- ? — unknown

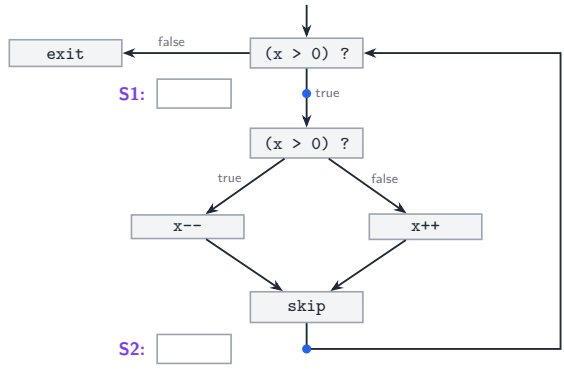
- ⊥ — undefined

A **finite** domain — only five elements, whatever the program. Assume the analysis **interprets conditionals** such as `x > 0`.

Quiz: What Is the Sign of `x` at S1 and S2?

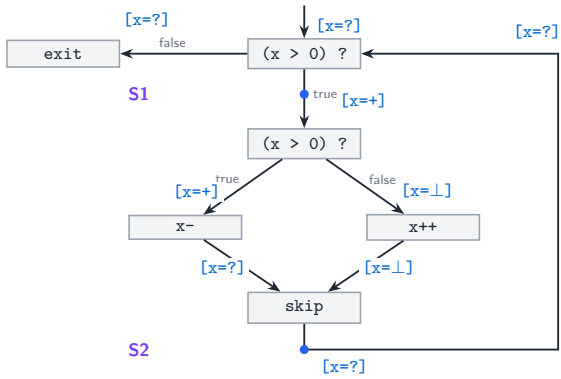


Quiz: What Is the Sign of `x` at S1 and S2?

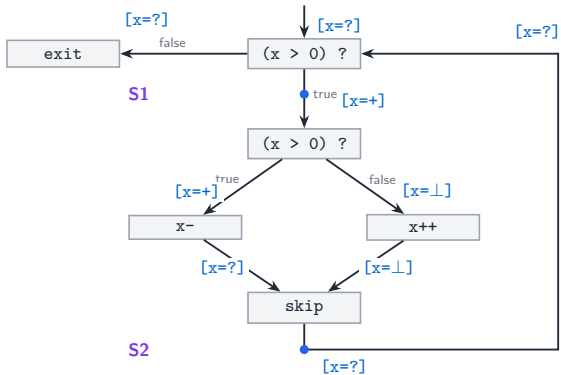


Enter one of −, 0, +, ?, ⊥ in each box. Assume conditionals *are* interpreted.

Solution: What Is the Sign of x at S1 and S2?



Solution: What Is the Sign of x at S1 and S2?



- At **S1**, the loop body is entered only when the condition `x > 0` is true. Therefore, the abstract value of `x` is `+`.
- At **S2**, information from different iterations is merged. The sign domain cannot precisely represent all possible signs after the updates, so the merged result is `?`.

Typical Components of a Data-Flow Static Analysis

1. **Program representation** — how the program is represented.
   Control-flow graph (CFG), abstract syntax tree (AST), intermediate representation (IR), or bytecode.

Typical Components of a Data-Flow Static Analysis

1. **Program representation** — how the program is represented.
   Control-flow graph (CFG), abstract syntax tree (AST), intermediate representation (IR), or bytecode.

2. **Abstract domain** — what properties can be represented.
   In this lecture, we used the value lattice consisting of constants, ? (unknown), and $\perp$ (undefined).

Typical Components of a Data-Flow Static Analysis

1. **Program representation** — how the program is represented.
   Control-flow graph (CFG), abstract syntax tree (AST), intermediate representation (IR), or bytecode.

2. **Abstract domain** — what properties can be represented.
   In this lecture, we used the value lattice consisting of constants, ? (unknown), and $\perp$ (undefined).

3. **Transfer functions** — how statements transform abstract states.
   Assignments update abstract values, expressions are evaluated abstractly, and merge (join) operators combine information at control-flow joins.

Typical Components of a Data-Flow Static Analysis

1. **Program representation** — how the program is represented.
   Control-flow graph (CFG), abstract syntax tree (AST), intermediate representation (IR), or bytecode.

2. **Abstract domain** — what properties can be represented.
   In this lecture, we used the value lattice consisting of constants, ? (unknown), and $\perp$ (undefined).

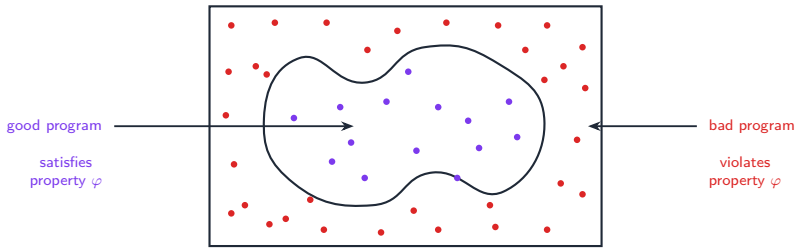
3. **Transfer functions** — how statements transform abstract states.
   Assignments update abstract values, expressions are evaluated abstractly, and merge (join) operators combine information at control-flow joins.

4. **Fixed-point algorithm** — how the analysis reaches a stable solution.
   Repeatedly applies transfer functions until no abstract state changes. Practical implementations typically use a worklist algorithm.

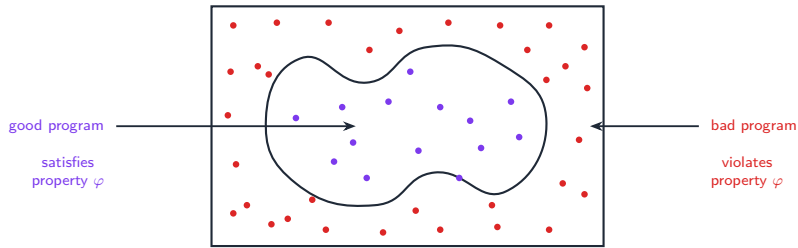
Characterising Analyses

Soundness, completeness, and the two ways to be wrong

The Space of All Programs



The Space of All Programs



Example property φ: **no divide-by-zero errors**.

This picture is the **ground truth** — what an analysis is trying to discover. Nobody can actually draw it.

Two Programs, One Property

```
void main() {
    while (x > 0) {
        assert(x != 0);
        ... 1/x ...
        if (x > 0)
            x--;
        else
            x++;
    }
}
```

Program A

```
void main() {
    while (x > 0) {
        if (x > 0)
            x--;
        else
            x++;
        assert(x != 0);
        ... 1/x ...
    }
}
```

Program B

Two Programs, One Property

```
void main() {
    while (x > 0) {
        assert(x != 0);
        ... 1/x ...
        if (x > 0)
            x--;
        else
            x++;
    }
}
```

Program A

```
void main() {
    while (x > 0) {
        if (x > 0)
            x--;
        else
            x++;
        assert(x != 0);
        ... 1/x ...
    }
}
```

Program B

Question: Which program satisfies the property

$\varphi =$ “No divide-by-zero errors”?

Two Programs, One Property

```
void main() {
    while (x > 0) {
        assert(x != 0);
        ... 1/x ...
        if (x > 0)
            x--;
        else
            x++;
    }
}
```

Good Program

(safe, bug-free)

```
void main() {
    while (x > 0) {
        if (x > 0)
            x--;
        else
            x++;
        assert(x != 0);
        ... 1/x ...
    }
}
```

Program B

Question: Which program satisfies the property

$\varphi =$ "No divide-by-zero errors"?

- **Program A:** The division occurs immediately after checking `x > 0`, so `x` is always positive. **Safe.**

Two Programs, One Property

```
void main() {
    while (x > 0) {
        assert(x != 0);
        ... 1/x ...
        if (x > 0)
            x--;
        else
            x++;
    }
}
```

Good Program

(safe, bug-free)

```
void main() {
    while (x > 0) {
        if (x > 0)
            x--;
        else
            x++;
        assert(x != 0);
        ... 1/x ...
    }
}
```

Bad Program

(unsafe, buggy)

Question: Which program satisfies the property

$\varphi =$ "No divide-by-zero errors"?

- **Program A:** The division occurs immediately after checking `x > 0`, so `x` is always positive. **Safe.**
- **Program B:** If `x == 1`, then `x--` makes it 0, and the subsequent `1/x` causes a divide-by-zero. **Unsafe.**

Two Programs, One Property

```
void main() {
    while (x > 0) {
        assert(x != 0);
        ... 1/x ...
        if (x > 0)
            x--;
        else
            x++;
    }
}
```

Good Program

(safe, bug-free)

```
void main() {
    while (x > 0) {
        if (x > 0)
            x--;
        else
            x++;
        assert(x != 0);
        ... 1/x ...
    }
}
```

Bad Program

(unsafe, buggy)

Question: Which program satisfies the property

$\varphi =$ "No divide-by-zero errors"?

- **Program A:** The division occurs immediately after checking `x > 0`, so `x` is always positive. **Safe.**
- **Program B:** If `x == 1`, then `x--` makes it 0, and the subsequent `1/x` causes a divide-by-zero. **Unsafe.**

The programs contain the same statements—the only difference is their **execution order**, which changes whether

An Analysis Accepts or Rejects

An analysis is an **oval** drawn on that picture: programs **inside** are **accepted**, programs **outside** are **rejected**.

An Analysis Accepts or Rejects

An analysis is an **oval** drawn on that picture: programs **inside** are **accepted**, programs **outside** are **rejected**.

Sound

Never accepts a bad program.

"If I say safe, it **is** safe."

No **false negatives**.

Complete

Never rejects a good program.

"If I complain, it **is** broken."

No **false positives**.

An Analysis Accepts or Rejects

An analysis is an **oval** drawn on that picture: programs **inside** are **accepted**, programs **outside** are **rejected**.

Sound

Never accepts a bad program.

"If I say safe, it **is** safe."

No **false negatives**.

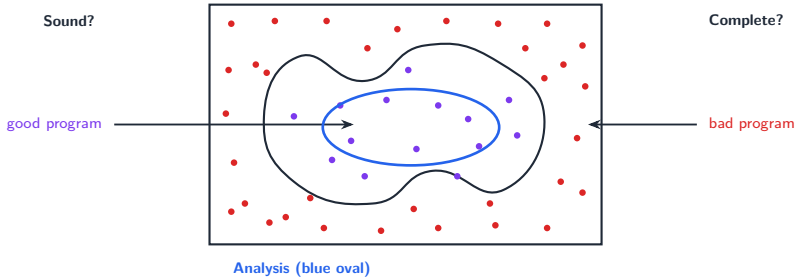
Complete

Never rejects a good program.

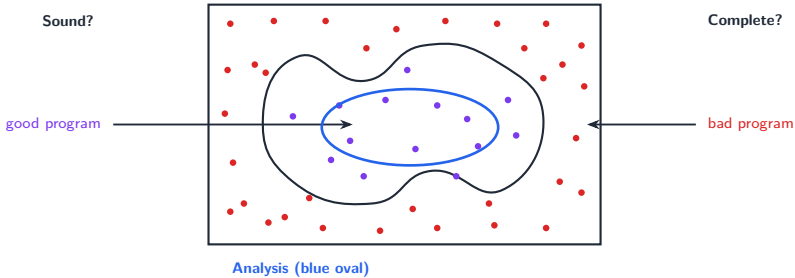
"If I complain, it **is** broken."

No **false positives**.

Kind 1

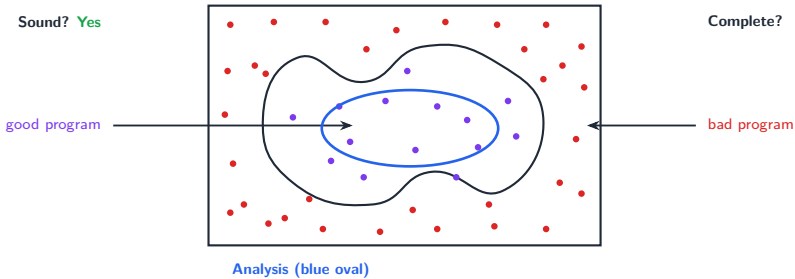

Sound?
Complete?
good program
bad program
Analysis (blue oval)

Kind 1



Question: Is this analysis **Sound**, **Complete**, **Both**, or **Neither**?

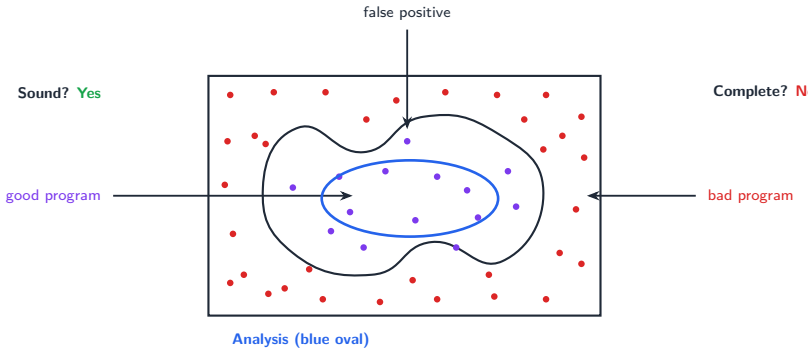
Kind 1



Question: Is this analysis **Sound**, **Complete**, **Both**, or **Neither**?

· **Sound:** Every accepted program is genuinely safe because the **analysis (blue oval)** lies completely inside the set of safe (purple) programs.

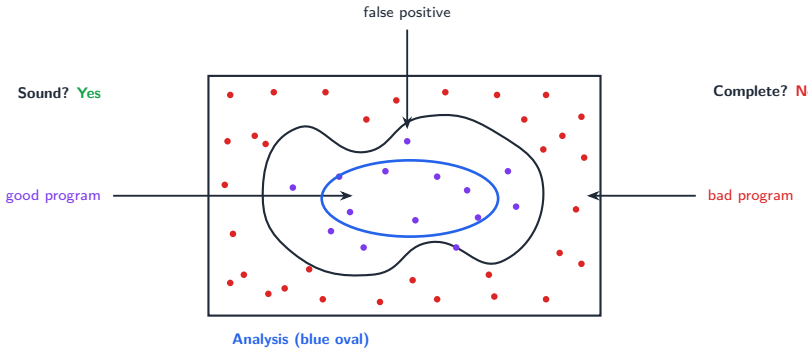
Kind 1



Question: Is this analysis **Sound**, **Complete**, **Both**, or **Neither**?

- **Sound:** Every accepted program is genuinely safe because the **analysis (blue oval)** lies completely inside the set of safe (purple) programs.
- **Not Complete:** Some safe programs lie outside the analysis and are rejected. These are **false positives**.

Kind 1



Question: Is this analysis **Sound**, **Complete**, **Both**, or **Neither**?

- **Sound:** Every accepted program is genuinely safe because the **analysis (blue oval)** lies completely inside the set of safe (purple) programs.
- **Not Complete:** Some safe programs lie outside the analysis and are rejected. These are **false positives**.
- **Kind 1: Sound but Incomplete** — this is where most practical static analyzers operate.

Where the False Positive Comes From

```
void main() {
    while (x > 0) {
        assert(x != 0);
        ... 1/x ...
        if (x > 0) x--;
        else       x++;
    }
}
```

good program

```
void main() {
    while (x > 0) {
        if (x > 0) x--;
        else       x++;
        assert(x != 0);
        ... 1/x ...
    }
}
```

bad program

Where the False Positive Comes From

```
void main() {
    while (x > 0) {
        assert(x != 0);
        ... 1/x ...
        if (x > 0) x--;
        else       x++;
    }
}
```

good program

```
void main() {
    while (x > 0) {
        if (x > 0) x--;
        else       x++;
        assert(x != 0);
        ... 1/x ...
    }
}
```

bad program

- Sign domain, **not** interpreting conditionals: `x` is ? at the assertion on the *left* → **rejected**. A **false positive**.

Where the False Positive Comes From

```
void main() {
    while (x > 0) {
        assert(x != 0);
        ... 1/x ...
        if (x > 0) x--;
        else       x++;
    }
}
```

good program

```
void main() {
    while (x > 0) {
        if (x > 0) x--;
        else       x++;
        assert(x != 0);
        ... 1/x ...
    }
}
```

bad program

- Sign domain, **not** interpreting conditionals: `x` is ? at the assertion on the *left* → **rejected**. A **false positive**.
- Sign domain, **interpreting** conditionals: `x` is + on the left → **accepted**. The false positive is **averted**.

Where the False Positive Comes From

```
void main() {
    while (x > 0) {
        assert(x != 0);
        ... 1/x ...
        if (x > 0) x--;
        else       x++;
    }
}
```

good program

```
void main() {
    while (x > 0) {
        if (x > 0) x--;
        else       x++;
        assert(x != 0);
        ... 1/x ...
    }
}
```

bad program

- Sign domain, **not** interpreting conditionals: `x` is ? at the assertion on the *left* → **rejected**. A **false positive**.
- Sign domain, **interpreting** conditionals: `x` is + on the left → **accepted**. The false positive is **averted**.
- On the *right*, `x` is ? at the assertion **either way** → **rejected**. Correctly — it really is buggy.

Where the False Positive Comes From

```
void main() {
    while (x > 0) {
        assert(x != 0);
        ... 1/x ...
        if (x > 0) x--;
        else       x++;
    }
}
```

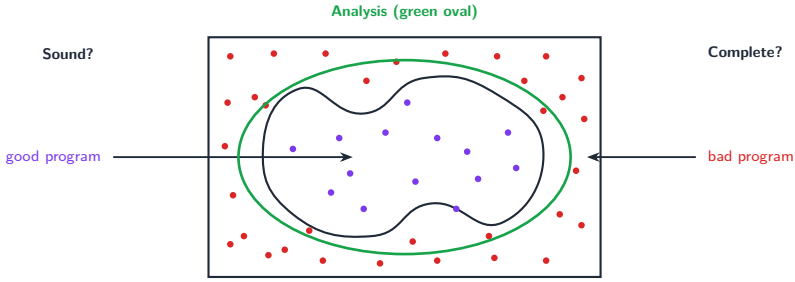
good program

```
void main() {
    while (x > 0) {
        if (x > 0) x--;
        else       x++;
        assert(x != 0);
        ... 1/x ...
    }
}
```

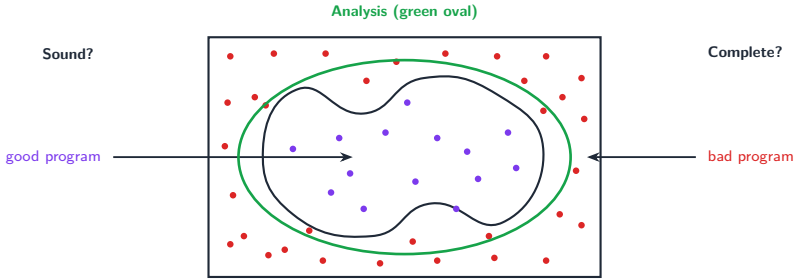
bad program

- Sign domain, **not** interpreting conditionals: `x` is ? at the assertion on the *left* → **rejected**. A **false positive**.
- Sign domain, **interpreting** conditionals: `x` is + on the left → **accepted**. The false positive is **averted**.
- On the *right*, `x` is ? at the assertion **either way** → **rejected**. Correctly — it really is buggy.

Kind 2

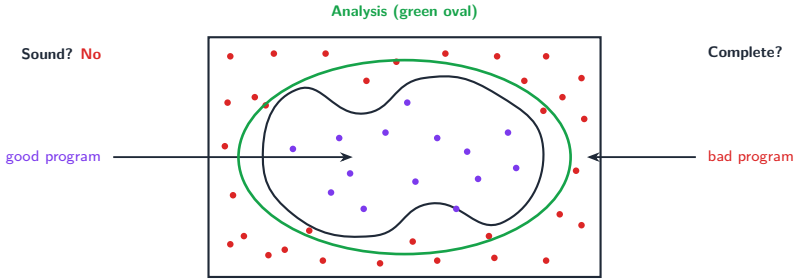

Analysis (green oval)
Sound?
Complete?
good program
bad program

Kind 2



Is this analysis **Sound**,**Complete**, **Both**, or **Neither**?

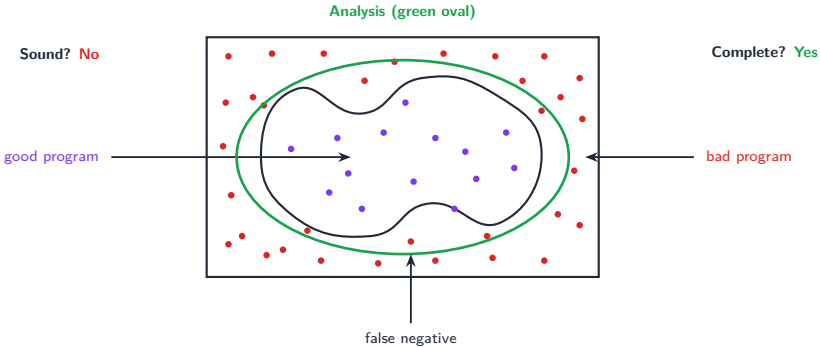
Kind 2



Is this analysis **Sound**,**Complete**, **Both**, or **Neither**?

- **Not Sound:** the analysis accepts some unsafe programs because the green oval contains red programs.

Kind 2

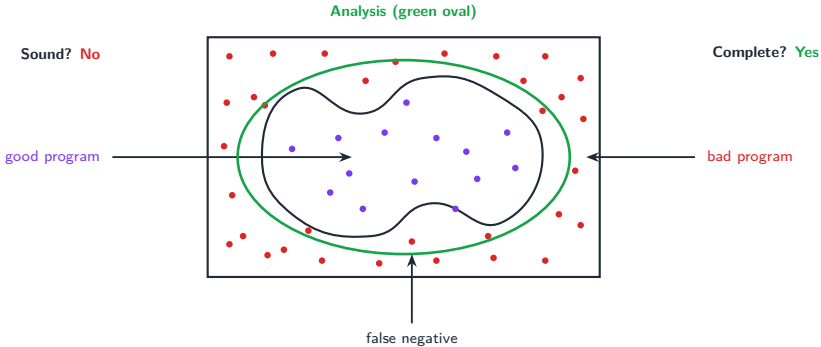


Is this analysis **Sound**,**Complete**, **Both**, or **Neither**?

· **Not Sound:** the analysis accepts some unsafe programs because the green oval contains red programs.

· **Complete:** every safe program is accepted because the entire purple region lies inside the analysis. The accepted buggy programs are **false negatives**.

Kind 2



Is this analysis **Sound**,**Complete**, **Both**, or **Neither**?

· **Not Sound:** the analysis accepts some unsafe programs because the green oval contains red programs.

· **Complete:** every safe program is accepted because the entire purple region lies inside the analysis. The accepted buggy programs are **false negatives**.

· This is the behaviour of many **dynamic analyses** such as testing, fuzzing, sanitizers and Valgrind.

Where the False Negative Comes From

```
void main() {
    while (x > 0) {
        assert(x != 0);
        ... 1/x ...
        if (x > 0) x--;
        else       x++;
    }
}
```

good program

run with `x = 3`

```
void main() {
    while (x > 0) {
        if (x > 0) x--;
        else       x++;
        assert(x != 0);
        ... 1/x ...
    }
}
```

bad program

run with `x = -3`

Where the False Negative Comes From

```
void main() {
    while (x > 0) {
        assert(x != 0);
        ... 1/x ...
        if (x > 0) x--;
        else       x++;
    }
}
```

good program

run with `x = 3`

```
void main() {
    while (x > 0) {
        if (x > 0) x--;
        else       x++;
        assert(x != 0);
        ... 1/x ...
    }
}
```

bad program

run with `x = -3`

· A dynamic analysis infers facts by **monitoring runs** on a suite of test inputs. Assume it performs no abstraction at all.

Where the False Negative Comes From

```
void main() {
    while (x > 0) {
        assert(x != 0);
        ... 1/x ...
        if (x > 0) x--;
        else       x++;
    }
}
```

good program

run with `x = 3`

```
void main() {
    while (x > 0) {
        if (x > 0) x--;
        else       x++;
        assert(x != 0);
        ... 1/x ...
    }
}
```

bad program

run with `x = -3`

· A dynamic analysis infers facts by **monitoring runs** on a suite of test inputs. Assume it performs no abstraction at all.

· Left, `x = 3`: no error observed. After enough such runs it **accepts**. **Correct** — the program is good.

Where the False Negative Comes From

```
void main() {
    while (x > 0) {
        assert(x != 0);
        ... 1/x ...
        if (x > 0) x--;
        else       x++;
    }
}
```

good program

run with `x = 3`

```
void main() {
    while (x > 0) {
        if (x > 0) x--;
        else       x++;
        assert(x != 0);
        ... 1/x ...
    }
}
```

bad program

run with `x = -3`

- A dynamic analysis infers facts by **monitoring runs** on a suite of test inputs. Assume it performs no abstraction at all.
- Left, `x = 3`: no error observed. After enough such runs it **accepts**. **Correct** — the program is good.
- Right, `x = -3`: the loop is **never entered**. No error observed → **accepted**. **False negative.**

Where the False Negative Comes From

```
void main() {
    while (x > 0) {
        assert(x != 0);
        ... 1/x ...
        if (x > 0) x--;
        else       x++;
    }
}
```

good program

run with `x = 3`

```
void main() {
    while (x > 0) {
        if (x > 0) x--;
        else       x++;
        assert(x != 0);
        ... 1/x ...
    }
}
```

bad program

run with `x = -3`

- A dynamic analysis infers facts by **monitoring runs** on a suite of test inputs. Assume it performs no abstraction at all.
- Left, `x = 3`: no error observed. After enough such runs it **accepts**. **Correct** — the program is good.
- Right, `x = -3`: the loop is **never entered**. No error observed → **accepted**. **False negative.**

- *Any* $x \leq 0$ hides this bug. The bug needs `x = 1`, or a run that reaches it.

Four Kinds of Analysis

Sound, incomplete

false positives only

static analysis

Four Kinds of Analysis

Sound, incomplete
false positives only
static analysis

Complete, unsound
false negatives only
dynamic analysis / testing

Four Kinds of Analysis

Sound, incomplete

false positives only

static analysis

Complete, unsound

false negatives only

dynamic analysis / testing

Neither

both kinds of error

most practical bug-finders: Coverity, Semgrep rulesets, linters

Four Kinds of Analysis

Sound, incomplete

false positives only

static analysis

Complete, unsound

false negatives only

dynamic analysis / testing

Neither

both kinds of error

most practical bug-finders: Coverity, Semgrep rulesets, linters

Both

no errors at all

impossible in general (Rice's theorem)

Four Kinds of Analysis

Sound, incomplete

false positives only

static analysis

Complete, unsound

false negatives only

dynamic analysis / testing

Neither

both kinds of error

most practical bug-finders: Coverity, Semgrep rulesets, linters

Both

no errors at all

impossible in general (Rice's theorem)

Quiz: Dynamic vs. Static

Match each box with its feature.

	Dynamic	Static
Cost		
Effectiveness		

Quiz: Dynamic vs. Static

Match each box with its feature.

	Dynamic	Static
Cost		
Effectiveness		

A. Unsound (may miss errors)

B. Proportional to the program's run time

C. Proportional to the program's size

D. Incomplete (may report spurious errors)

Quiz: Dynamic vs. Static

Match each box with its feature.

	Dynamic	Static
Cost	B	C
Effectiveness	A	D

A. Unsound (may miss errors)

B. Proportional to the program's run time

C. Proportional to the program's size

D. Incomplete (may report spurious errors)

Quiz: Dynamic vs. Static

Match each box with its feature.

	Dynamic	Static
Cost	B	C
Effectiveness	A	D

A. Unsound (may miss errors)

B. Proportional to the program's run time

C. Proportional to the program's size

D. Incomplete (may report spurious errors)

Quiz: Dynamic vs. Static

Match each box with its feature.

	Dynamic	Static
Cost	B	C
Effectiveness	A	D

A. Unsound (may miss errors)

B. Proportional to the program's run time

C. Proportional to the program's size

D. Incomplete (may report spurious errors)