

# Program Reasoning

## 3. Concepts in Program Verification

Kihong Heo



# Impact of Poor Software Quality



The Patriot Missile (1991)  
Floating-point roundoff  
28 soldiers died



The Ariane-5 Rocket (1996)  
Integer Overflow  
\$100M



NASA's Mars Climate Orbiter (1999)  
Meters-Inches Miscalculation  
\$125M

**CNN** U.S. | World | Politics | Money | Opinion | Health | Entertainment | Tech | Style | Travel | Sports | Video | Live TV | U.S.

The 'Heartbleed' security flaw that affects most of the Internet

By Heather Kelly, CNN  
Updated 5:11 PM ET, Wed April 9, 2014

A large red heart outline with liquid dripping down from it, symbolizing the 'Heartbleed' bug.

This dangerous Android security bug could let anyone hack your phone camera

By Anthony Spadafora November 23, 2019

Camera app vulnerabilities allow attackers to remotely take photos, record video and spy on users

A smartphone lying on a keyboard, with a green digital skull icon on its screen, symbolizing a security breach.

**AIRLINE MARSHALL** TRANSPORTATION 08:30:2019 07:00 AM

**What Boeing's 737 MAX Has to Do With Cars: Software**

Investigators believe faulty software contributed to two fatal crashes. A newly discovered fault will likely keep the 737 MAX grounded until the fall.

A Boeing 737 MAX airplane captured from a low angle, flying through a cloudy sky.

Homeland Security warns that certain heart devices can be hacked

New in Life & Style

Homeroom fifth-graders bond through poetry, art and Steph Curry

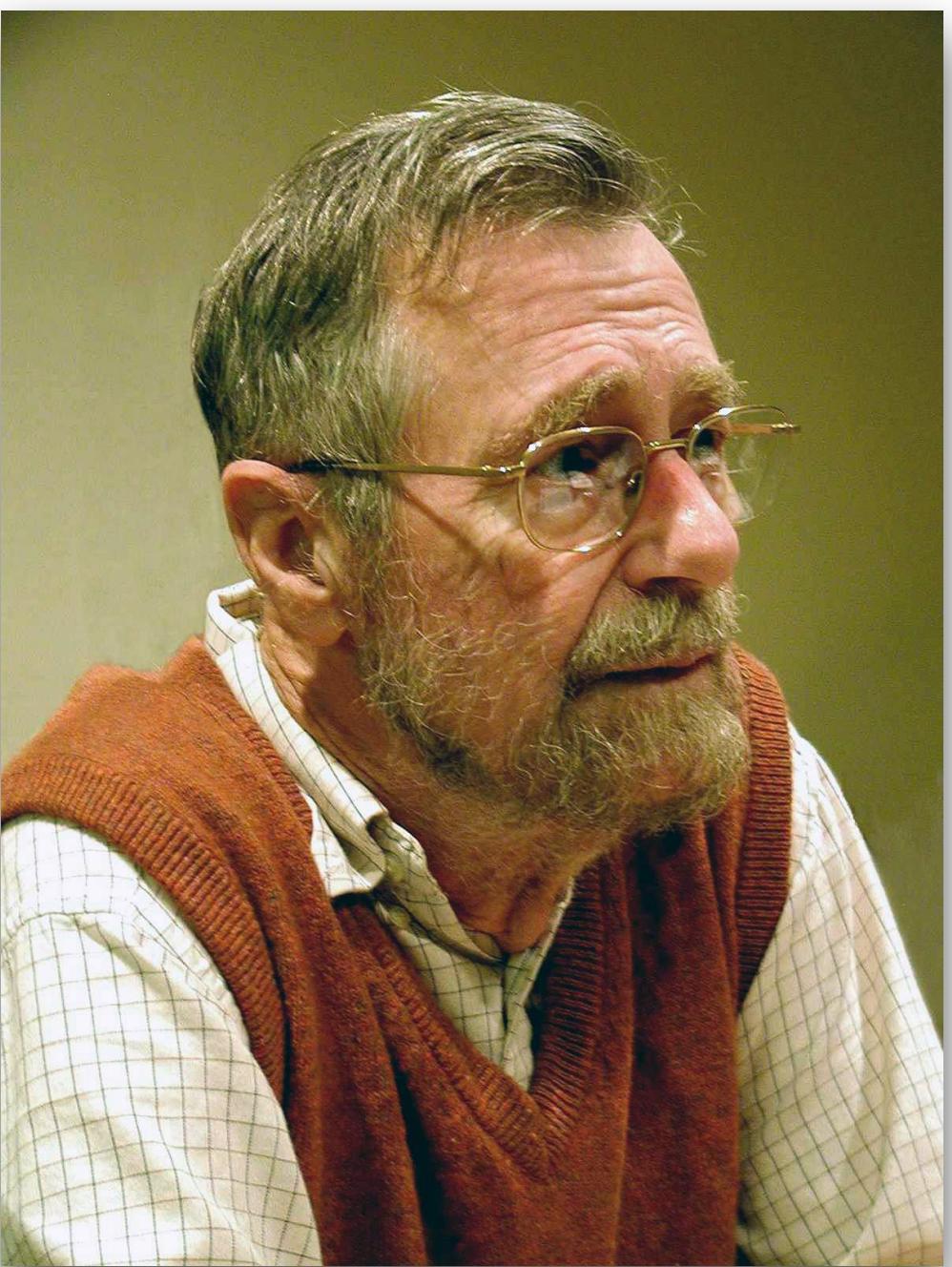
6 ways to celebrate Valentine's Day in Lake Geneva

Six ways to keep your kids healthy during winter

See More

Exterior view of the St. Jude Medical building, featuring a large sign that reads "ST. JUDE MEDICAL" and "MAIN ENTRANCE".

# Towards Error-free SW



***“Program testing can be used to show the presence of bugs,  
but never to show their absence!”***

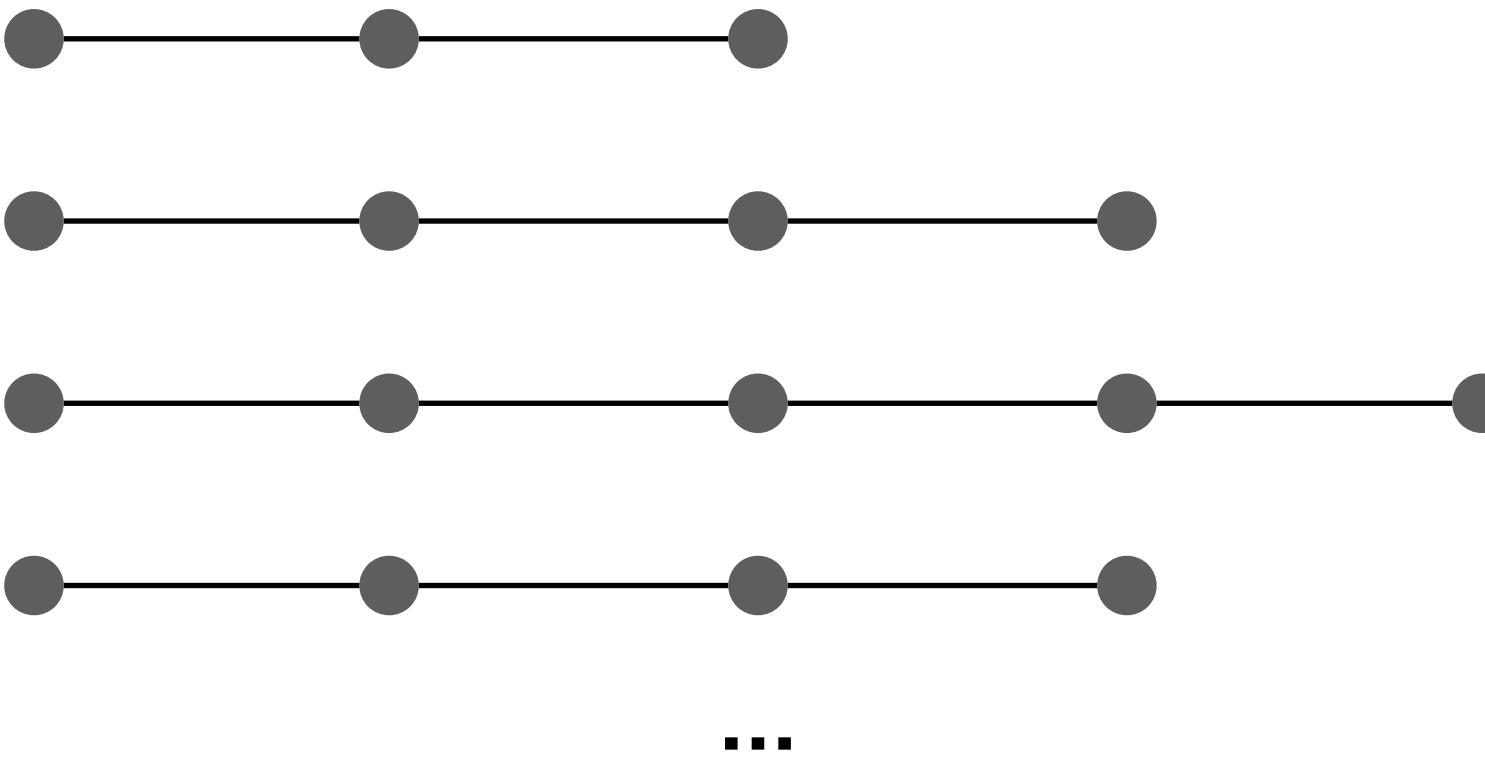
- Edsger W. Dijkstra, 1970

# Properties 성질

- Points of interest in programs
  - for verification, bug detection, optimization, understanding, etc
  - E.g., “ $p == \text{NULL?}$ ”, “ $\text{idx} < \text{size?}$ ”, “ $\text{fp}$  can be only f, g, or h?”, “value of x”, etc
- Two categories:
  - Trace properties = properties of individual execution traces
    - safety properties + liveness properties
  - Information-flow properties = properties of multiple execution traces

# Trace 실행경로

- Trace = a list of states  $(2 \times 2 \times 2) \times (2 + 1)$
- Recall small-step operational semantics  $\rightarrow (4 \times 2) \times (2 + 1)$
- A program can have an (infinite) set of traces  $\rightarrow 8 \times (2 + 1)$
- $\llbracket P \rrbracket$  : a set of all possible execution traces  $\rightarrow 8 \times 3$
- $\llbracket P \rrbracket$  : a set of all possible execution traces  $\rightarrow 24$

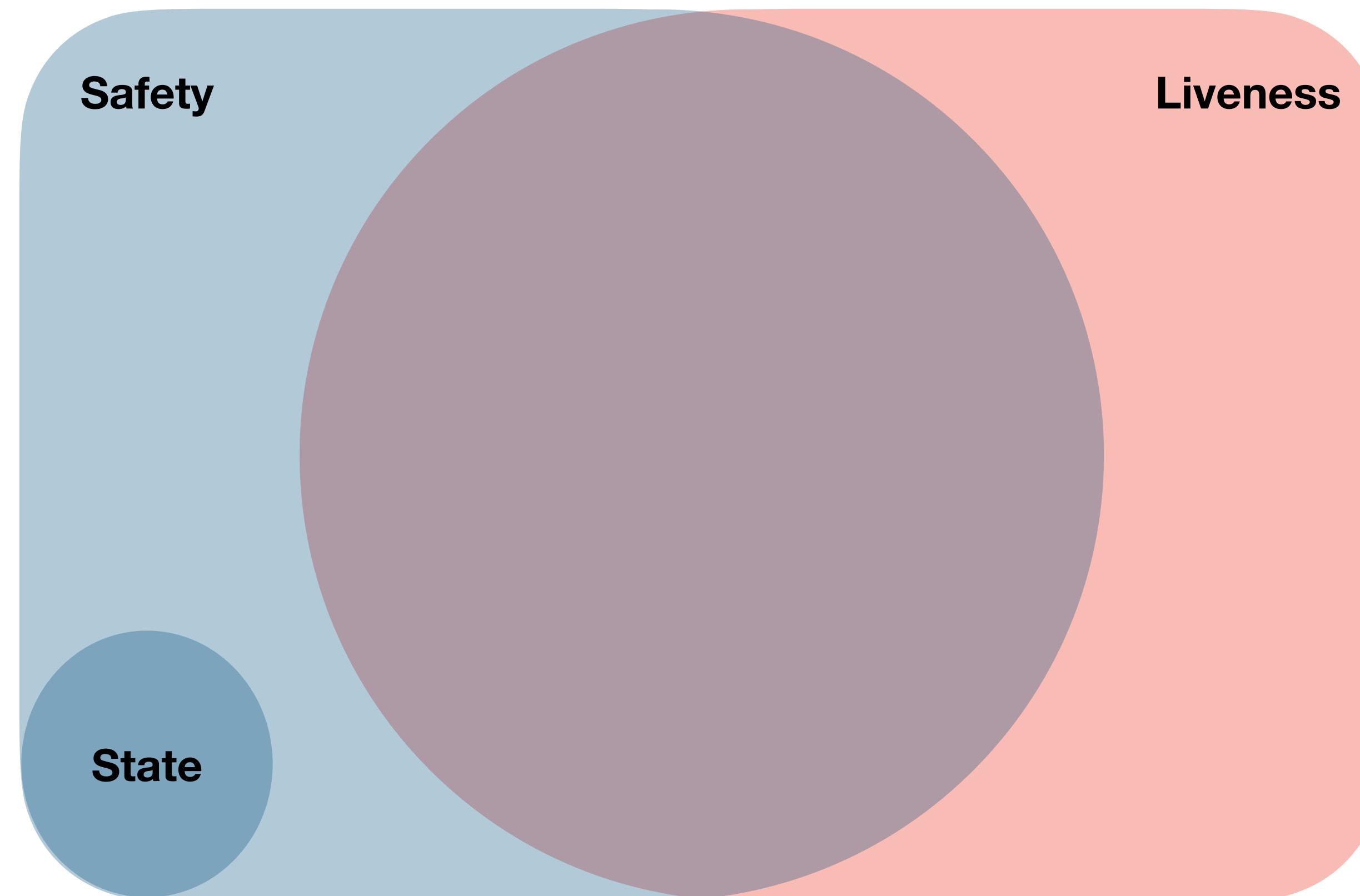


# Trace Properties

개별경로 성질

- A semantic property  $\mathcal{P}$  that can be defined by a **set of execution traces** that satisfies  $\mathcal{P}$ 
  - Ex1: “all traces that satisfies  $x \neq 0$ ”
  - Ex2: “all traces where the value of  $y$  at termination is the same as the one in the entry point”
- Program  $P$  satisfies property  $\mathcal{P}$  iff  $\llbracket P \rrbracket \subseteq T_{\mathcal{P}}$
- State properties: defined by a set of states (so, obviously trace properties)
  - E.g., division-by-zero, integer overflow
- Any trace property: the conjunction of a safety and a liveness property

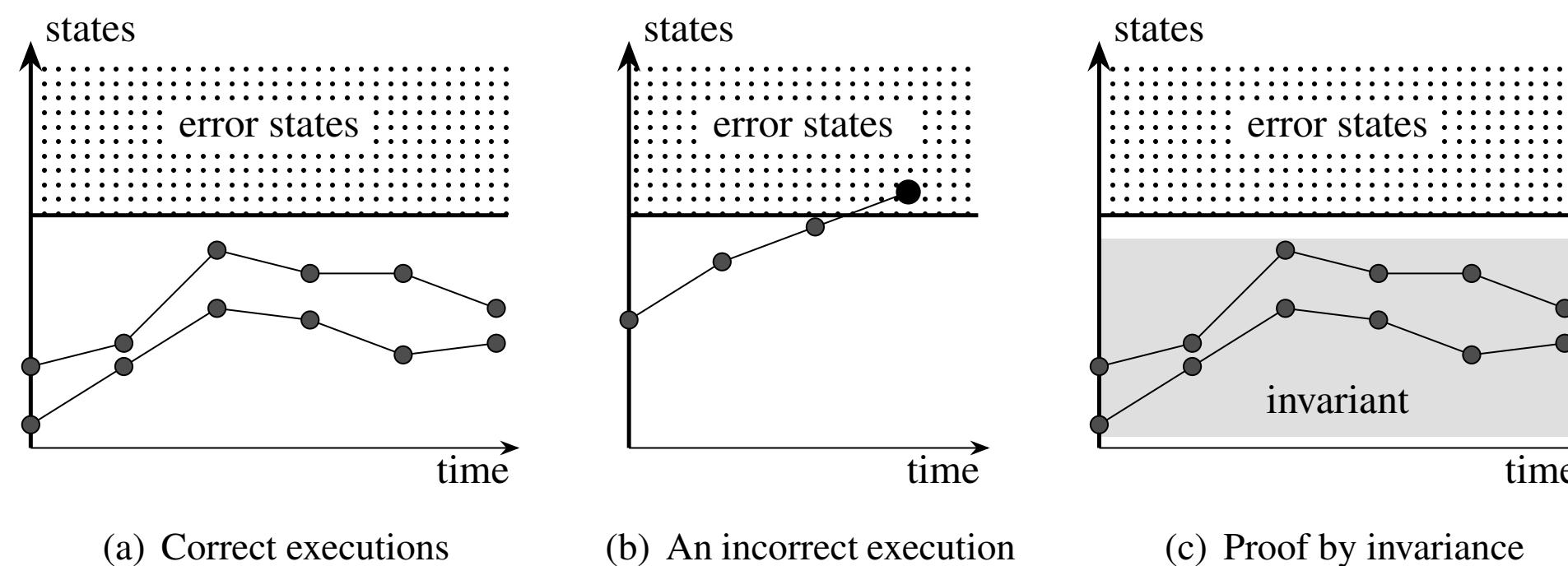
# Trace Properties



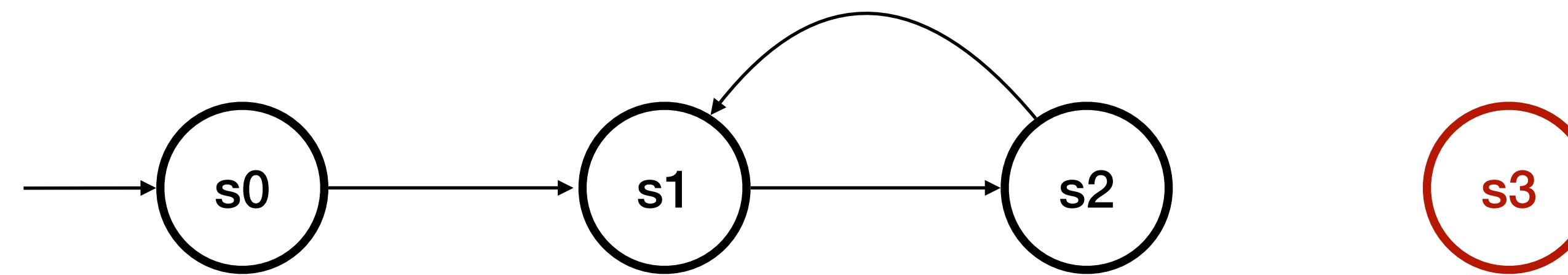
# Safety Property

항상성질

- A program **never** exhibit a behavior observable within **finite time**
  - “Bad things will never occur”
  - Bad things: integer overflow, buffer overrun, deadlock, etc
- If false, then there exists a **finite counterexample**
- To prove: all executions never reach error states



# Example



**Reachable states <= 0 step : {s0}**  
**Reachable states <= 1 step : {s0, s1}**  
**Reachable states <= 2 steps : {s0, s1, s2}**  
**Reachable states <= 3 steps : {s0, s1, s2}**  
**Reachable states <= 4 steps : {s0, s1, s2}**  
...  
**Reachable states <= 100 steps : {s0, s1, s2}**  
...  
**Reachable states <  $\infty$  steps : {s0, s1, s2}**

# Invariant

불변식

- Assertions supposed to be **always true** and **remain unchanged** after any operations
  - Starting from a state in the invariant, any computation step also leads to another state in the invariant (i.e., fixed point!)
  - E.g., “x has an int value during the execution”, “y is larger than 1 at line 5”
- Loop invariant: assertion to be true at the beginning of every loop iteration

```
x = 0;  
while (x < 10) {  
    x = x + 1;  
}  
assert(x > 0);  
assert(x == 10);
```

Loop invariant 1: “x is an integer”

Loop invariant 2: “ $x \geq 0$ ”

Loop invariant 3: “ $0 \leq x \leq 10$ ”

# Invariant in Art



by hyerinshelly, 2022



<https://github.com/prosyslab-classroom/cs424-program-reasoning/discussions/249>

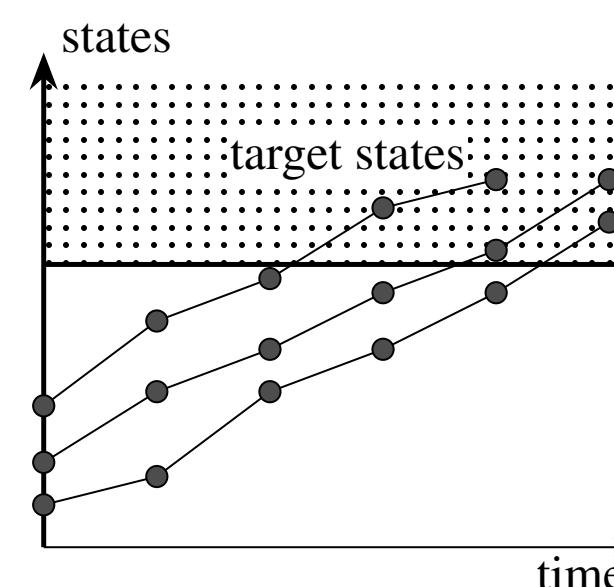
# Example: Division-by-Zero

```
1: int main(){
2:     int x = input();
3:     x = 2 * x - 1;
4:     while (x > 0) {
5:         x = x - 2;
6:     }
7:     assert(x != 0);
8:     return 10 / x;
9: }
```

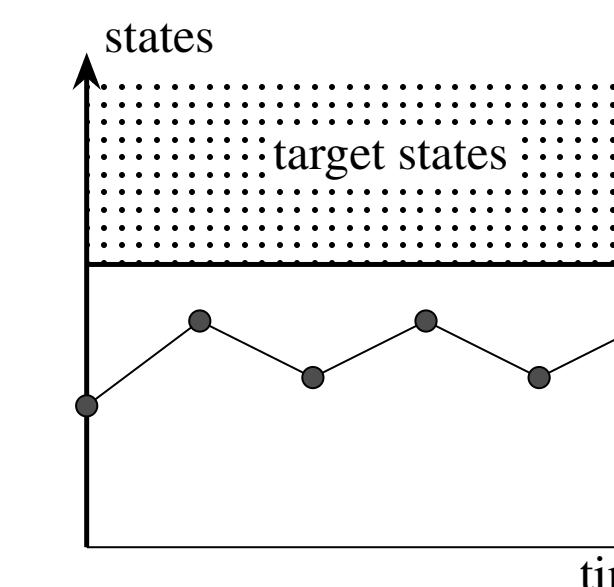
```
1: int main(){
2:     int x = input();
3:     x = 2 * x;
4:     while (x > 0) {
5:         x = x - 2;
6:     }
7:     assert(x != 0);
8:     return 10 / x;
9: }
```

# Liveness Property 결국성질

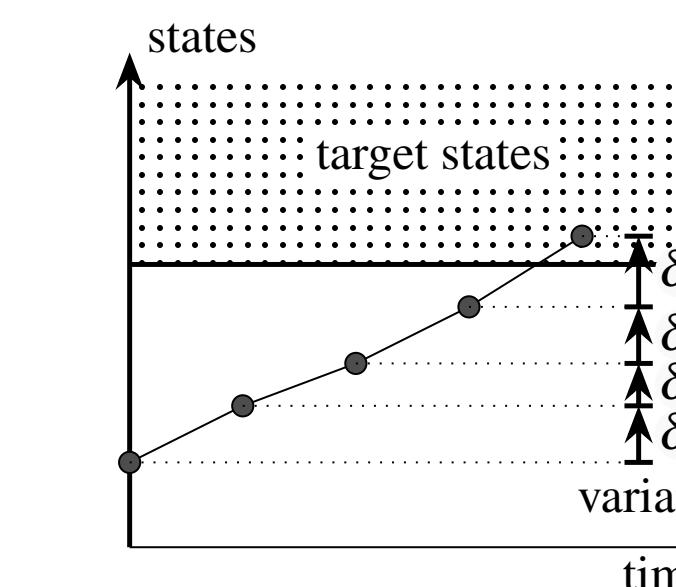
- A program will **never** exhibit a behavior observable only after **infinite time**  
(A program will **eventually** exhibit a behavior observable within **finite time**)
  - “Good things will eventually occur”
  - Good things: termination, fairness, etc
- If false then there exists an **infinite counterexample**
- To prove: all executions eventually reach target states



(a) Correct executions

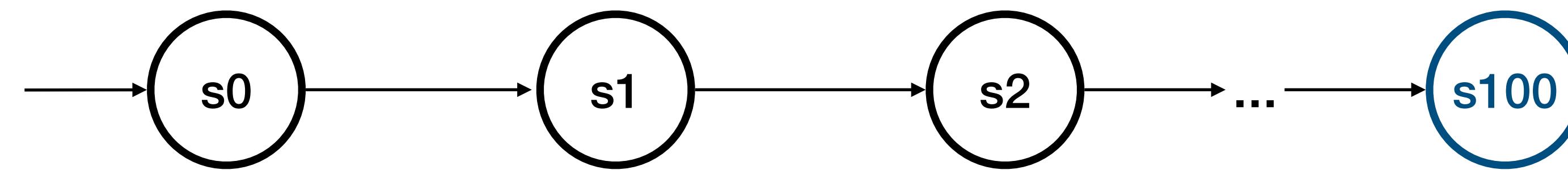


(b) An incorrect execution



(c) Proof by variance

# Example



**Shortest distance after 0 step : 100**

**Shortest distance after 1 step : 99**

**Shortest distance after 2 steps : 98**

**Shortest distance after 3 steps : 97**

...

(if we are sure that the distance will keep decreasing)

...

**Reachable states after 100 steps : 0**

# Variant

변동식

- A quantity that **evolves towards** the set of target states (so guarantee any execution eventually reaches the set)
- Usually, a value that is strictly decreasing for some well-founded order relation
  - Well-founded order: there is no infinite decreasing chain
  - E.g., an integer value that is always positive and strictly decreasing

```
x = pos_int();  
while (x > 0) {  
    x = x - 1;  
}
```

**x is always a positive integer**  $\wedge$  **x is strictly decreasing**  $\Rightarrow$  **The program terminates**

# Example: Termination

```
// A factorial program
fun fact(n) =
    r = 1;
    i = 0;
    while (i <= n) {
        r = r * i;
        i = i + 1;
    }
}
```

i  $\leq$  n + 1

0  $\leq$  n - i + 1  $\wedge$  n - i + 1 is strictly decreasing  $\Rightarrow$  termination

# Example

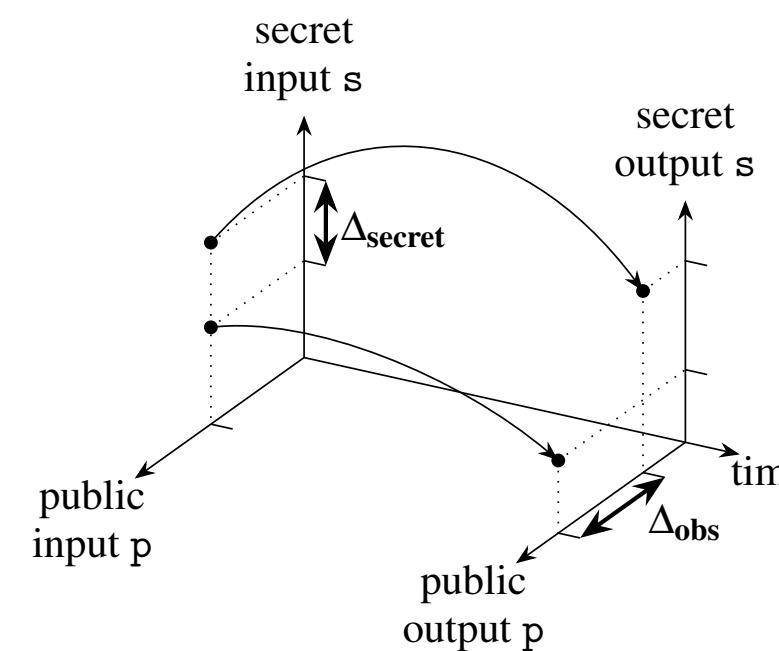
- Correctness of a sorting algorithm as trace property

Property	Safety or Liveness?	State?
Should not fail with a run-time error		
Should terminate		
Should return a sorted array (if terminated)		
Should return an array with the same elements and multiplicity (if terminated)		

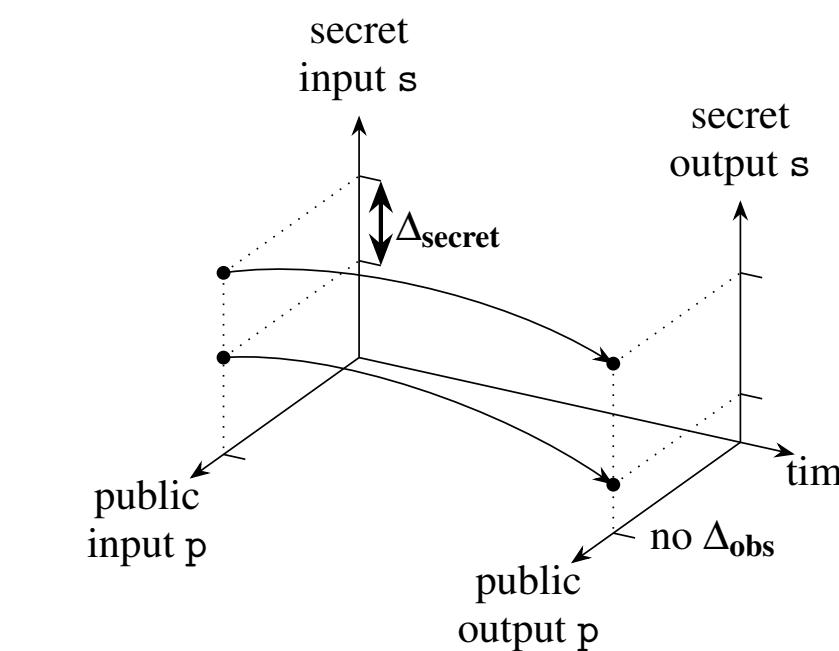
# Information Flow Properties

실행 간섭 성질

- Properties stating the absence of dependence between **pairs of executions**
  - Beyond trace properties: so called **hyper-properties**
- Mostly for security: multiple executions with public data should not derive private data
- E.g., a door lock beeps louder if a right digit is pressed at the right position



A pair of executions with insecure information flow



A pair of executions without insecure information flow

# Example

- Assume that variables s (secret) and p (public) take only 0 and 1

```
// Program 0  
p_out := p_in * [0, 1]
```

```
// Program 1  
p_out := p_in * s * [0, 1]
```

```
// Program 2  
p_out := p_in + [0, 1] - s
```

Input		Output
p	s	p
0	0	{0, 1}
0	1	{0, 1}
1	0	{0, 1}
1	1	{0, 1}

Input		Output
p	s	p
0	0	{0}
0	1	{0}
1	0	{0}
1	1	{0, 1}

Input		Output
p	s	p
0	0	{0, 1}
0	1	{0, 1}
1	0	{0, 1}
1	1	{0, 1}

# A Hard Limit: Undecidability

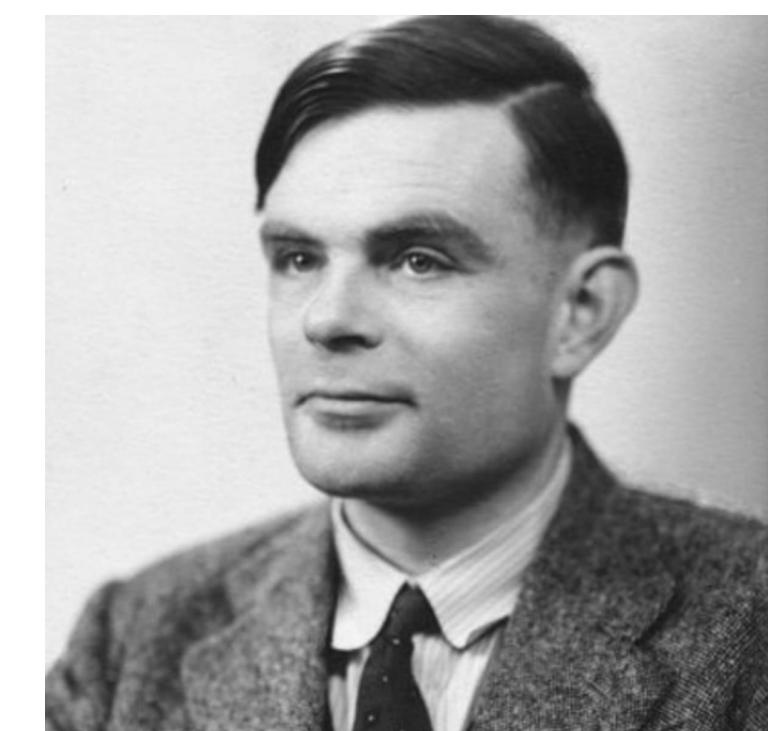
계산불가능성

**Theorem (Rice's theorem).** Any **non-trivial** semantic properties are **undecidable**.

- Non-trivial property: worth the effort of designing a program analyzer for
  - trivial: true or false for all programs
- Undecidable? If decidable, it can solves the Halting problem!

HP: Given a Turing machine  $T$  and an input  $i$ , does  $T$  eventually halt on  $i$ ?

Undecidable: There is no Turing machine that can solve HP!



# Informal Proof of Undecidability of HP

HP: Given a Turing machine  $T$  and an input  $i$ , does  $T$  eventually halt on  $i$ ?

- Assume  $H(T, i)$  returns true or false
- Let  $F(x) = \text{if } H(x, x) \text{ then loop() else halt()}$
- Does  $F(F)$  terminate?

# If Decidable?

- Many mathematical problems become trivial!

```
// Fermat's last theorem
```

```
for (a, b, c, n) in N4 do
  if n > 2 && an + bn = cn then
    exit()
```

```
// Goldbach's conjecture
```

```
for k in Even do
  for (p, q) in Prime2 do
    if k != p + q then
      exit()
```



# Informal Proof of Rice's Theorem

- Assumption: HP is undecidable
- An analyzer **A** for a property: “*This program always prints 1 and finishes*”
- Given a program **P**, generate **P'** = “**P**; print 1;”
- Analyze **P'** using **A**: **A(P')**
  - **A(P')** says “Yes”: **P** halts,
  - **A(P')** says “No”: **P** does not halt
- HP is decidable if we use **A** : contradiction!

# Toward Computability

## Undecidable

⇒ Automatic, terminating, and exact reasoning is impossible  
⇒ If we give up one of them, it is computable!

- Manual rather than automatic: assisted proving
  - require expertise and manual effort
- Possibly nonterminating rather than terminating: model checking, testing
  - require stopping mechanisms such as timeout
- Approximate rather than exact: static analysis
  - report spurious results

# Soundness<sub>안전성</sub> and Completeness<sub>완전성</sub>

- Given a semantic property  $\mathcal{P}$ , and an analysis tool  $A$
- If  $A$  were perfectly accurate,

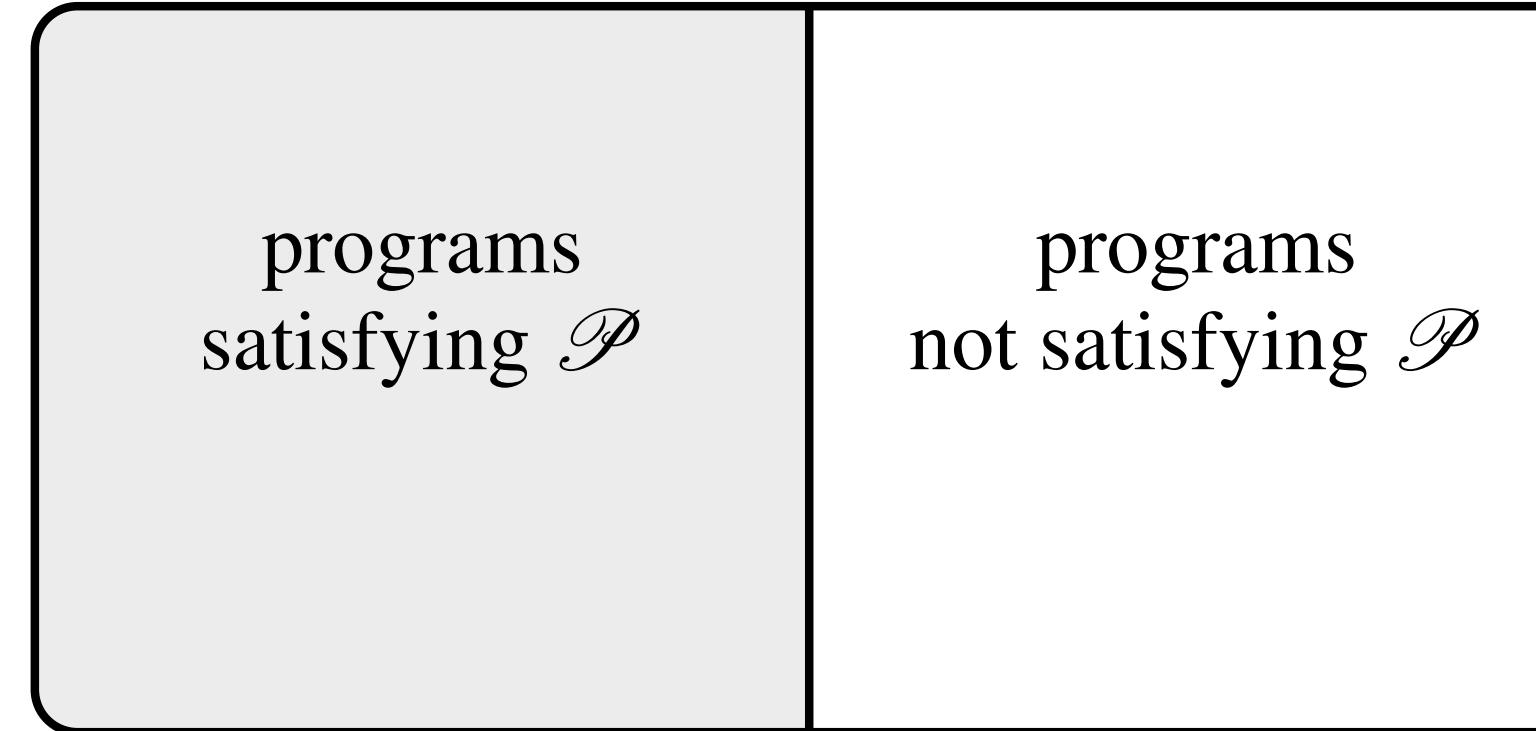
For all program  $p$ ,  $A(p) = \text{true} \iff p \text{ satisfies } \mathcal{P}$

which consists of

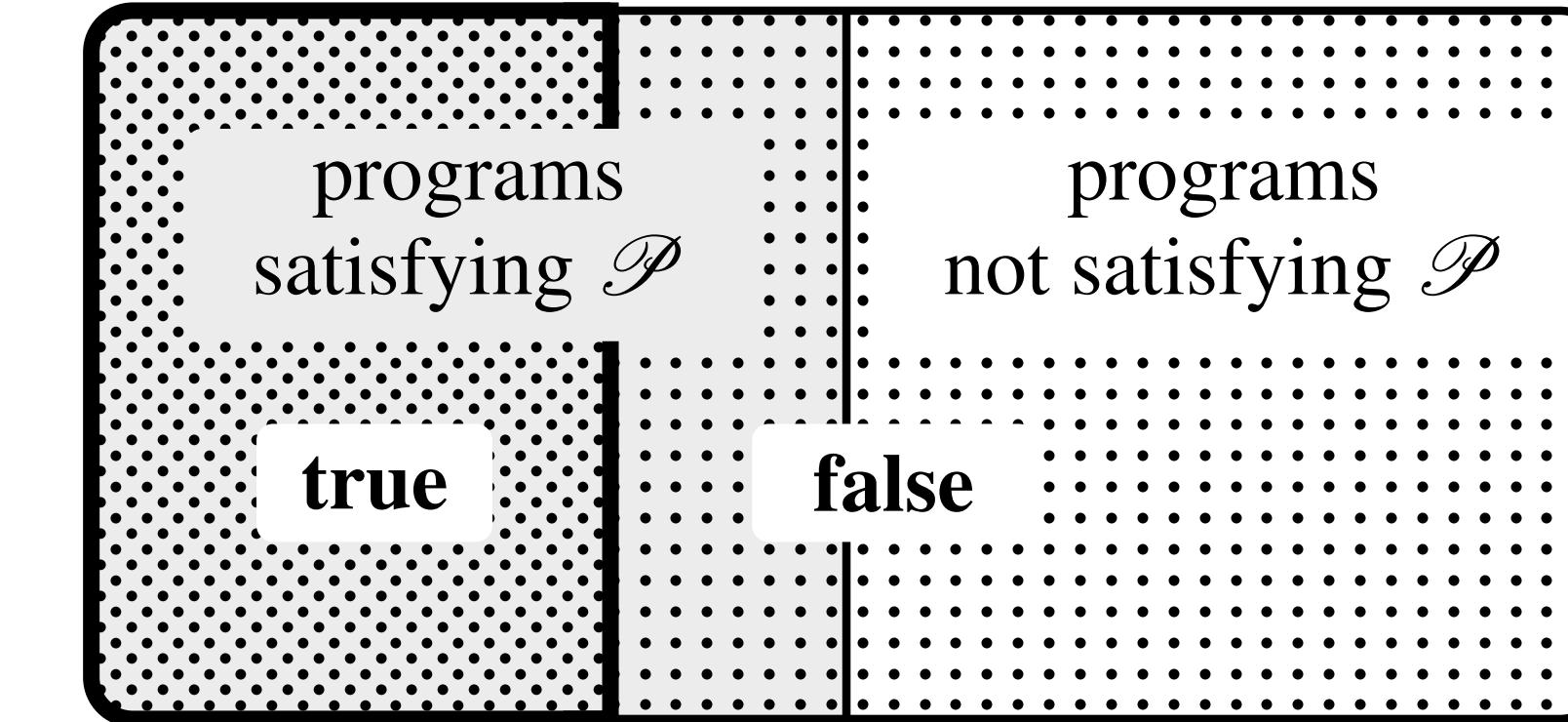
For all program  $p$ ,  $A(p) = \text{true} \Rightarrow p \text{ satisfies } \mathcal{P}$  **(soundness)**

For all program  $p$ ,  $A(p) = \text{true} \Leftarrow p \text{ satisfies } \mathcal{P}$  **(completeness)**

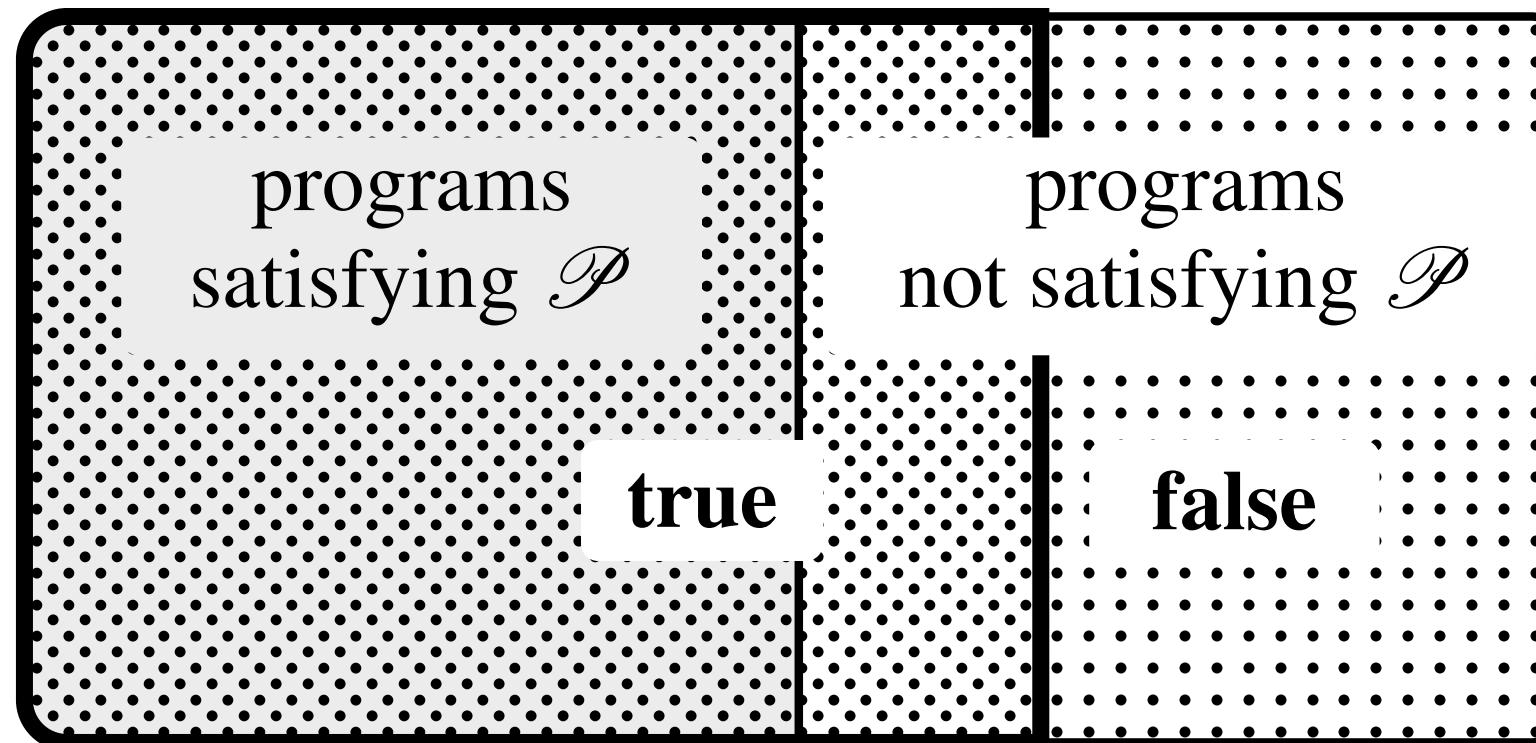
# Soundness<sub>안전성</sub> and Completeness<sub>완전성</sub>



(a) Programs



(b) Sound, incomplete analysis



(c) Unsound, complete analysis

- programs that satisfy  $\mathcal{P}$
- programs that do not satisfy  $\mathcal{P}$
- programs for which the analysis returns **true**
- programs for which the analysis returns **false**

(d) Legend

# Program Verification

- Prove a given program satisfies the target properties
  - Loop invariants provided by the user or another program analyzer
- **Sound and complete** if a “good” invariant is provided
- **Sound and incomplete** if an imprecise invariant is provided
- **Sound, complete, yet non-terminating** if the invariant generation does not terminate
- How to describe the target property (specification)?
- How to prove the target property (specification)?

## A: Program Logic

# Summary

- Property: point of interest in a program (safety, liveness, information flow, etc)
- Program verification: check whether a property is satisfied or not
- Hard limit of program analysis: generally undecidable problem
- Practical solutions:
  - **Manual** rather than **automatic**
  - **Possibly nonterminating** rather than **terminating**
  - **Approximate** rather than **exact**