

# P4 타입 체계의 기계화 명세를 이용한 짜깁 없는 조건 커버리지 기반 타입 오류 테스트 생성

**Failing with Purpose: Dangling Coverage-Guided  
Negative Test Generation from a Mechanized P4 Type System (FSE'26)**

이재현, 정석훈, 류석영 / ERC 2026 겨울정기워크샵 2026.02.02

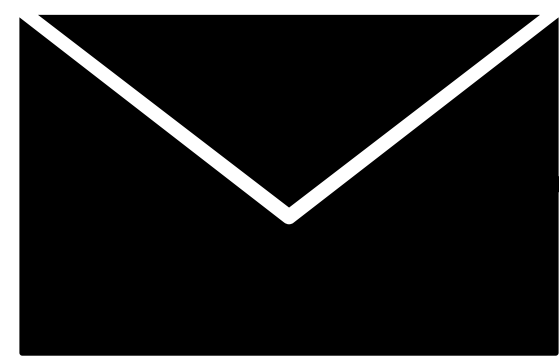


# P4 프로그래밍 언어란?

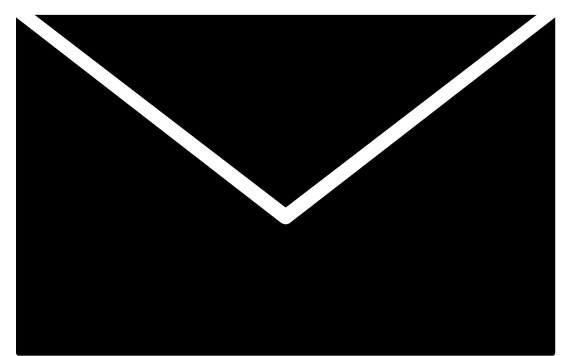
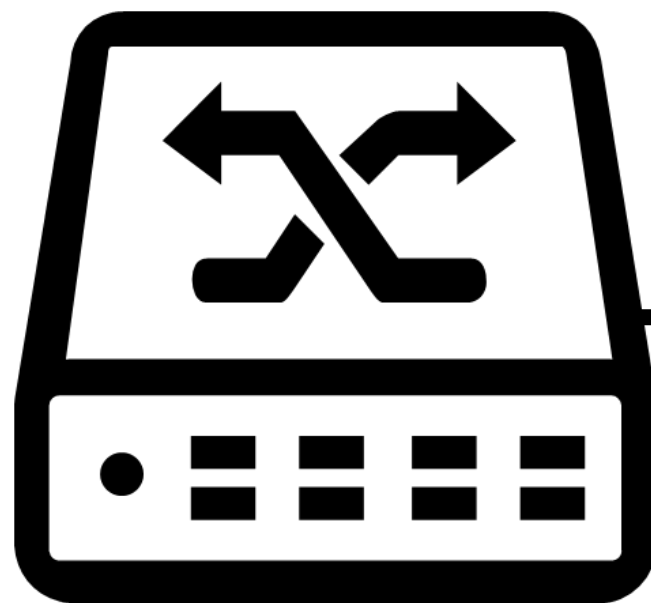
## Programming Protocol-independent Packet Processors (P4)

스위치 등 네트워크 장비의 행동을 기술하는 도메인 특화 언어

입력 패킷을 어떻게 파싱하고, 가공하여, 출력 패킷으로 내보내는지 표현



입력 패킷

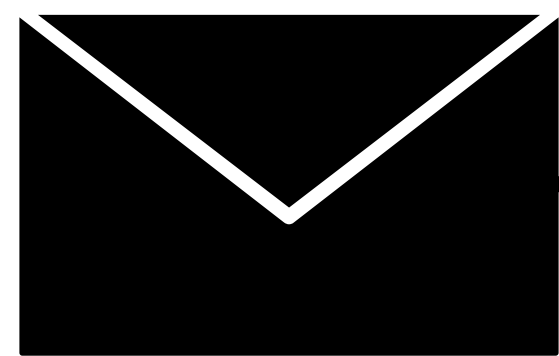


출력 패킷

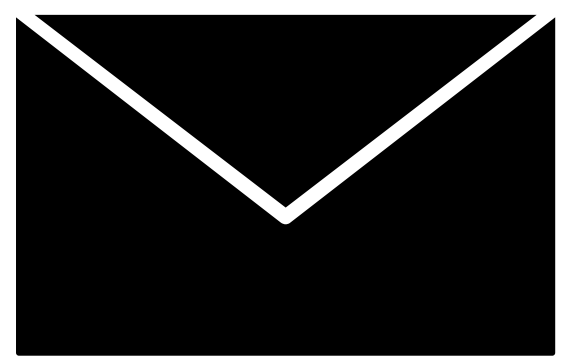
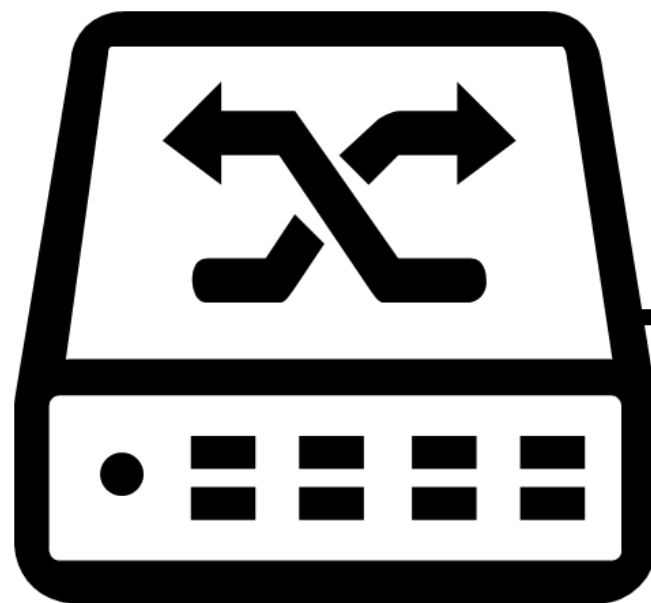
# P4 프로그래밍 언어란?

## Programming Protocol-independent Packet Processors (P4)

- ✓ C 언어와 비슷한 문법과 의미 구조
- ✓ 정적 타입 언어



입력 패킷



출력 패킷

# P4 타입 검사기를 제대로 구현했다면:

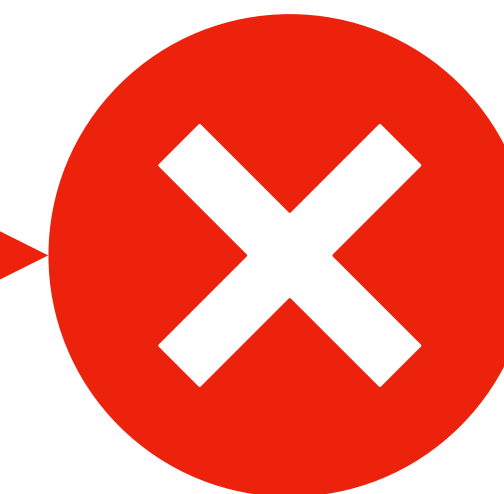
타입이 올바른 (well-typed)  
프로그램



타입 오류 (ill-typed)  
프로그램



P4 타입 검사기



# 테스트로 타입 검사기를 검사하기

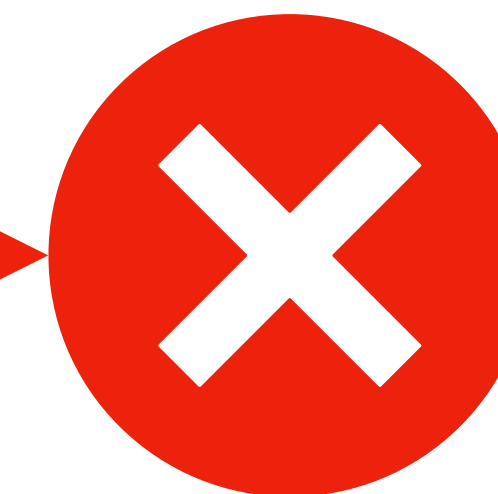
타입-올바름 테스트 (positive test)

타입이 올바른 (well-typed)  
프로그램



타입 오류 테스트 (negative test)

타입 오류 (ill-typed)  
프로그램



P4 타입 검사기

# P4 공식 명세가 정의하는 타입 체계 - 시프트 연산

## 8.10.2. A note about shifts

The left operand of shifts can be any one out of unsigned bit-strings, signed bit-strings, and arbitrary-precision integers, and the right operand of shifts must be either an expression of type `bit<S>` or a compile-time known value that is a non-negative integer. The result has the same type as the left operand.

시프트 연산자의 우항은:

- `bit<S>` 타입이거나,
- `int<S>` 타입이라면 컴파일 시점에 음수가 아니라는 것을 알 수 있어야 한다.

# 타입이 올바른 시프트 연산

```
bit<8> shl(in bit<8> x,  
           in bit<8> y) {  
    return x << y;  
}
```



```
bit<8> shl(in bit<8> x) {  
    const int<8> y = 42;  
    return x << y;  
}
```



시프트 연산자의 우항은:

- `bit<S>` 타입이거나,
- `int<S>` 타입이라면 컴파일 시점에 음수가 아니라는 것을 알 수 있어야 한다.

# 타입 오류를 일으키는 시프트 연산

```
bit<8> shl(in bit<8> x, in int<8> y) {  
    return x << y;  
}
```



시프트 연산자의 우항은:

- `bit<S>` 타입이거나,
- `int<S>` 타입이라면 컴파일 시점에 음수가 아니라는 것을 알 수 있어야 한다.

# 기존 타입 오류 테스트 집합이 충분하지 않음

공식 컴파일러 저장소의 타입 오류 테스트 254개 중,  
시프트 우항 조건에 대한 타입 오류 테스트 부재



시프트 연산자의 우항은:

- `bit<S>` 타입이거나,
- `int<S>` 타입이라면 컴파일 시점에 음수가 아니라는 것을 알 수 있어야 한다.

# 기존 타입 오류 테스트 집합이 충분하지 않음

공식 컴파일러 저장소의 타입 오류 테스트 254개 중,  
시프트 우항 조건에 대한 타입 오류 테스트 부재



Shift by a string literal is not rejected by type checker #5094

✓ Closed

Bug

#5446

컴파일러 결함

# 목표: 품질 좋은 타입 오류 테스트 집합 만들기

P4 타입 체계의 **모든 타입 오류 조건**을 적어도 한번씩 검사하기를 원함

## 8.10.2. A note about shifts

The left operand of shifts can be any one out of unsigned bit-strings, signed bit-strings, and arbitrary-precision integers, and the right operand of shifts must be either an expression of type `bit<S>` or a compile-time known value that is a non-negative integer. The result has the same type as the left operand.

```
bit<8> shl(in bit<8> x, in int<8> y) {  
    return x << y;  
}
```

# 문제: 타입 오류 테스트 집합의 품질을 정량화하기 어려움

P4 타입 체계의 **모든 타입 오류 조건**을 적어도 한번씩 검사하기를 원함

## 8.10.2. A note about shifts

The left operand of shifts can be any one of unsigned bit strings, signed bit strings, and a `bit<S>` or a compile-time known value that is a non-negative integer. The result has the same type as the left operand.

```
bit<8> shl(in bit<8> x, in int<8> y) {
```

주어진 테스트가 어떤 타입 오류 조건을 검사하는지 알 수 없음

```
}
```

기계화 명세를 바탕으로 타입 오류 테스트 집합의 품질을 정량화하는  
짜깁 없는 조건 커버리지 (dangling coverage)를 정의하고,  
커버리지 기반 마구생성 (fuzzing)을 통해 P4 언어의 타입 오류 조건을  
최대한 많이 검사하는 타입 오류 테스트 집합 자동 생성

# 기계화 명세 기반 타입 오류 테스트 생성

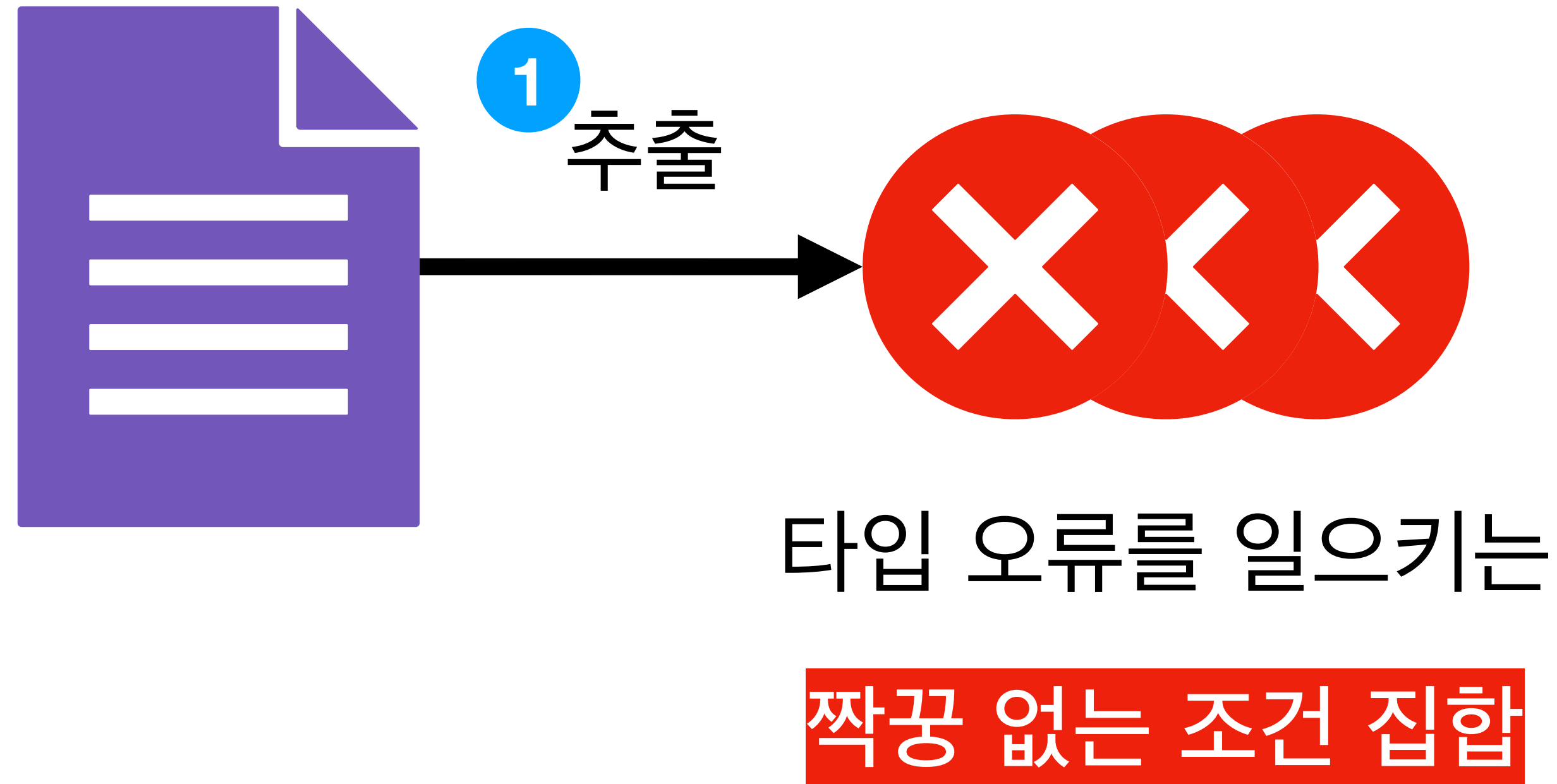
## P4 기계화 명세



1. SpecTec\* 명세 언어로 작성한 명세
2. P4의 문법과 타입 체계를 엄밀하게 표현함
3. 명세를 파싱하여 명세를 데이터로 다룰 수 있음
4. 명세를 실행 가능함
  - P4 타입 체계 명세를 실행 = P4 타입 검사기

# 기계화 명세 기반 타입 오류 테스트 생성

P4 기계화 명세



# 기계화 명세 기반 타입 오류 테스트 생성

P4 기계화 명세

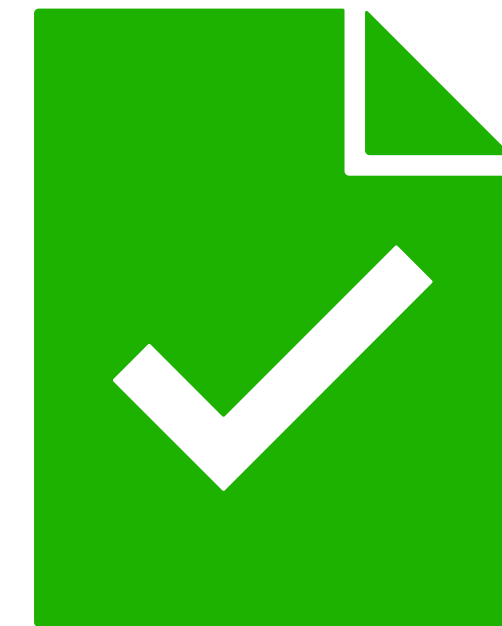


① 추출



타입 오류를 일으키는

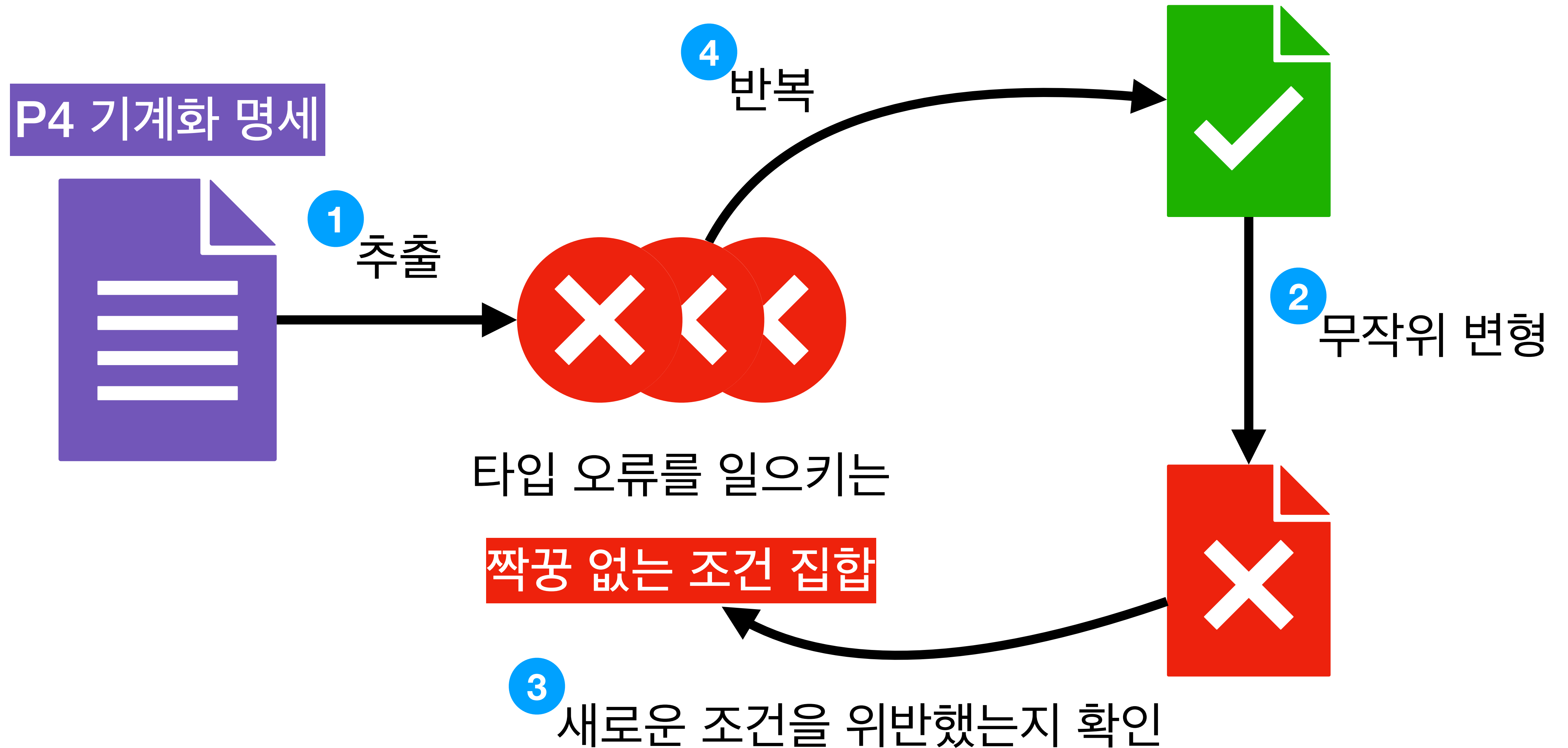
**짜깁 없는 조건 집합**



② 무작위 변형



# 기계화 명세 기반 타입 오류 테스트 생성



# 기계화 명세로 타입 규칙을 엄밀하게 표현하기

시프트 연산자의 우향은:

- `bit`<S> 타입이거나,
- `int`<S> 타입이라면 컴파일 시점에 음수가 아니라는 것을 알 수 있어야 한다.

## P4 기계화 명세



```
syntax binop =  
  | PLUS | MINUS  
  | SHL  | SHR
```

```
syntax type = IntT | BitT
```

```
syntax expr =  
  | VarE text  
  | IntE int  
  | BitE nat  
  | BinE binop expr expr
```

# 추론 규칙 (inference rule) 기반 타입 체계 명세

검사 대상( $C, \text{BinE } \dots$ )에 대해, 조건이 모두 성립한다면, 타입 검사의 결과는  $\text{type\_l}$  이다.

rule Expr\_ok/binary-shift:

$C \vdash \text{BinE binop expr\_l expr\_r} : \text{type\_l}$

----

-- if  $\text{binop} \in \{ \text{SHL}, \text{SHR} \}$

-- Expr\_ok:  $C \vdash \text{expr\_l} : \text{type\_l}$

-- Expr\_ok:  $C \vdash \text{expr\_r} : \text{type\_r}$

-- if  $\text{\$is\_int}(\text{type\_r}) \Rightarrow \text{\$is\_non\_negative}(C, \text{expr\_r})$

# 실행 가능한 추론 규칙

rule Expr\_ok/binary-shift:

$C \vdash \text{BinE } \text{binop } \text{expr}_l \text{ expr}_r : \text{type}_l$

----

-- if  $\text{binop} \in \{ \text{SHL}, \text{SHR} \}$

-- Expr\_ok:  $C \vdash \text{expr}_l : \text{type}_l$

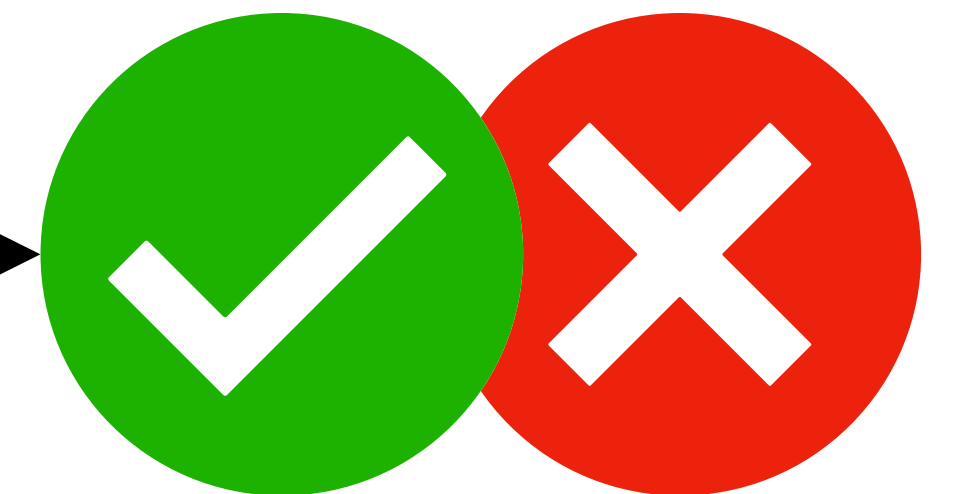
-- Expr\_ok:  $C \vdash \text{expr}_r : \text{type}_r$

-- if  $\text{\$is\_int}(\text{type}_r) \Rightarrow \text{\$is\_non\_negative}(C, \text{expr}_r)$

P4 프로그램



명세 실행기



# 타입이 올바른 프로그램을 주면

```
bit<8> shl(in bit<8> x,  
           in bit<8> y) {  
    return x << y;  
}
```

rule Expr\_ok/binary-shift:

C ⊢ BinE binop expr\_l expr\_r : type\_l

----

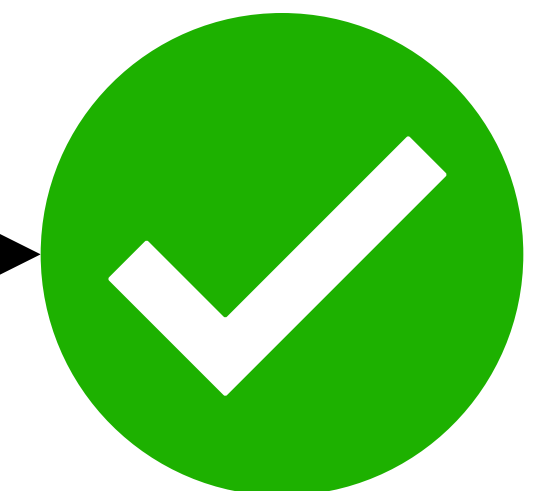
-- if binop ∈ { SHL, SHR }

-- Expr\_ok: C ⊢ expr\_l : type\_l

-- Expr\_ok: C ⊢ expr\_r : type\_r

-- if \$is\_int(type\_r) => \$is\_non\_negative(C, expr\_r)

명세 실행기



# 타입 오류 프로그램을 주면

rule Expr\_ok/binary-shift:

$C \vdash \text{BinE binop expr\_l expr\_r} : \text{type\_l}$

----

-- if  $\text{binop} \in \{ \text{SHL}, \text{SHR} \}$

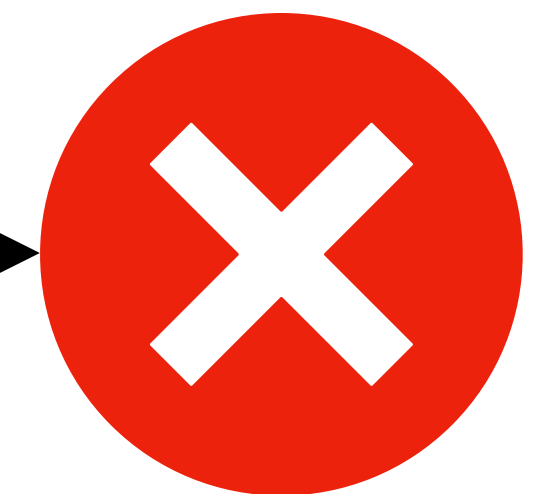
-- Expr\_ok:  $C \vdash \text{expr\_l} : \text{type\_l}$

-- Expr\_ok:  $C \vdash \text{expr\_r} : \text{type\_r}$

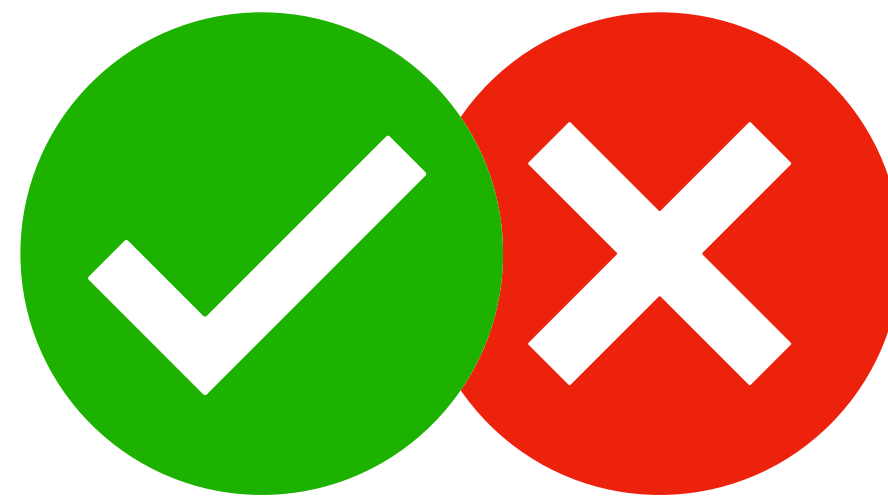
-- if  $\text{\$is\_int}(\text{type\_r}) \Rightarrow \text{\$is\_non\_negative}(C, \text{expr\_r})$

```
bit<8> shl(in bit<8> x,  
           in int<8> y) {  
    return x << y;  
}
```

명세 실행기



P4 프로그램



타입 검사 결과



P4 기계화 명세가 P4 타입 검사를 실행

P4 기계화 명세



SpecTec  
명세 언어

명세 실행기가 P4 기계화 명세를 실행

명세 실행기



OCaml

추론 규칙은 타입이 올바른 조건만을 정의함

추론 규칙의 조건을 위반하면 타입 오류에 해당함

```
rule Expr_ok/binary-shift:
```

```
  C ⊢ BinE binop expr_l expr_r : type_l
```

```
----
```

```
-- if binop ∈ { SHL, SHR }
```

```
-- Expr_ok: C ⊢ expr_l : type_l
```

```
-- Expr_ok: C ⊢ expr_r : type_r
```

```
-- if $is_int(type_r) => $is_non_negative(C, expr_r)
```

# 하지만, 모든 조건이 타입 오류 조건은 아님

rule Expr\_ok/binary-shift:

C ⊢ BinE binop expr\_l expr\_r : type\_l

----

-- if binop ∈ { SHL, SHR }

rule Expr\_ok/binary-plus-minus:

C ⊢ BinE binop expr\_l expr\_r : type\_l

----

-- if binop ∈ { PLUS, MINUS }

위반하면 다른 추론 규칙을 검사함;  
경우를 따지는 (case-analysis) 조건

짜꿍 조건 - 짝을 지어서 경우를 따지는 조건

# 목표: 경우를 따지는 조건 발라내기

rule Expr\_ok/binary-shift:

$C \vdash \text{BinE binop expr}_l \text{ expr}_r : \text{type}_l$

----

-- if  $\text{binop} \in \{ \text{SHL}, \text{SHR} \}$

-- Expr\_ok:  $C \vdash \text{expr}_l : \text{type}_l$

-- Expr\_ok:  $C \vdash \text{expr}_r : \text{type}_r$

-- if  $\text{is\_int}(\text{type}_r) \Rightarrow \text{is\_non\_negative}(C, \text{expr}_r)$

다른 규칙에 짝꿍 조건 있음;  
위반하면 타입 오류 대신 다른 경우 따짐

다른 규칙에 짝꿍 조건 없음;

위반하면 타입 오류가 일어나는 타입 오류 조건

# 생김새를 활용한 "짜꿍 없는 조건" 찾기

생김새 기반 (syntax-based heuristic) 짜꿍 찾기

P4 기계화 명세



짜꿍 있는 조건

```
-- if binop ∈ { SHL, SHR }
```

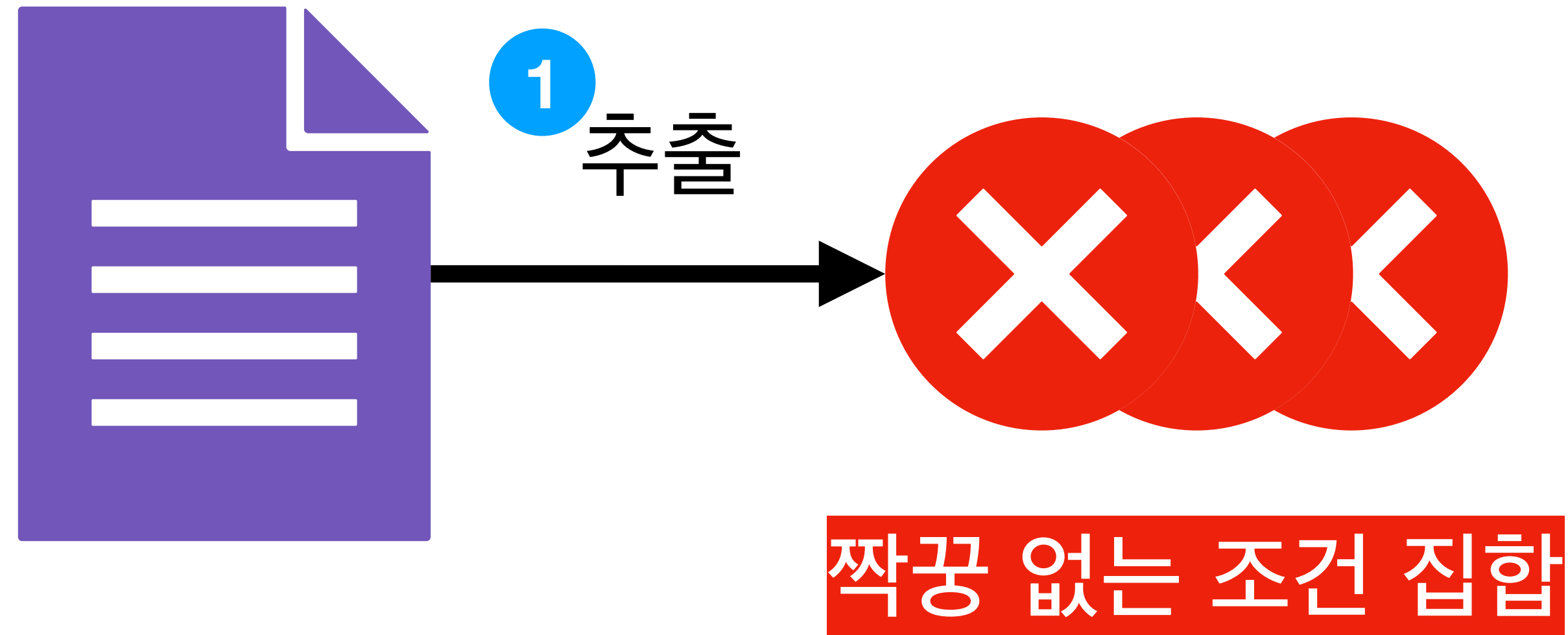
```
-- if binop ∈ { PLUS, MINUS }
```

짜꿍 없는 조건 (Dangling Premise)

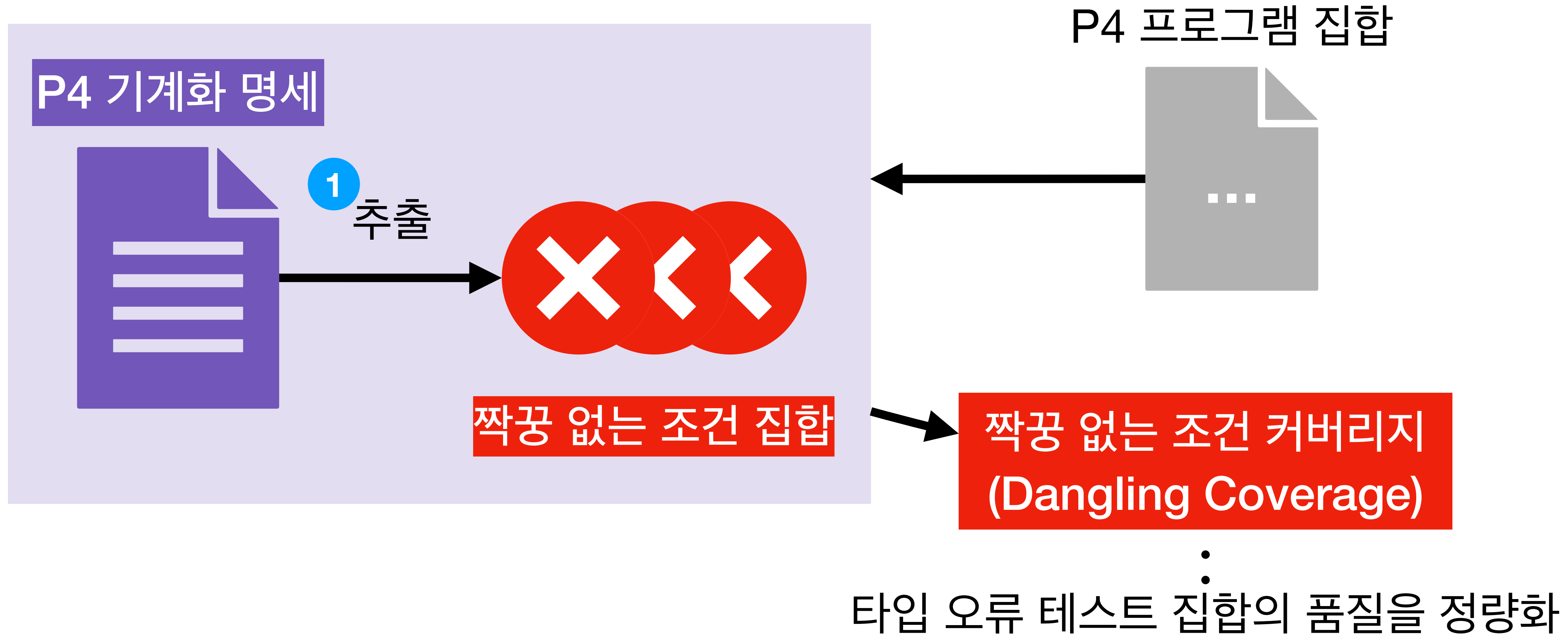
```
-- if $is_int(type_r) => $is_non_negative(C, expr_r)
```

# 기계화 명세로부터 짝꿍 없는 조건 추출

P4 기계화 명세



# 기계화 명세를 실행해 짝궁 없는 조건 커버리지 측정



# 타입 오류는 올바른 프로그램과 한끗 차이

```
bit<8> shl(in bit<8> x,  
           in bit<8> y) {  
    return x << y;  
}
```



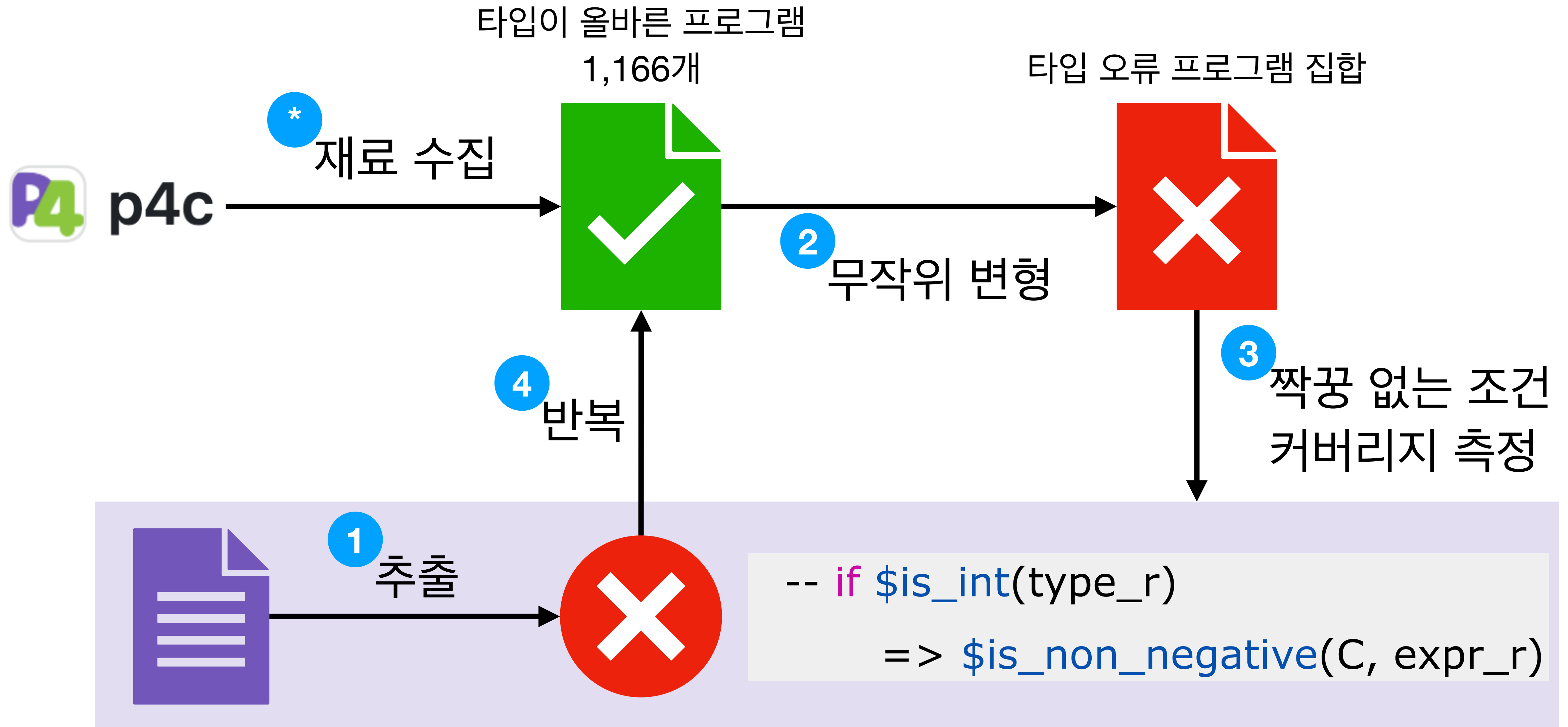
```
bit<8> shl(in bit<8> x,  
           in int<8> y) {  
    return x << y;  
}
```



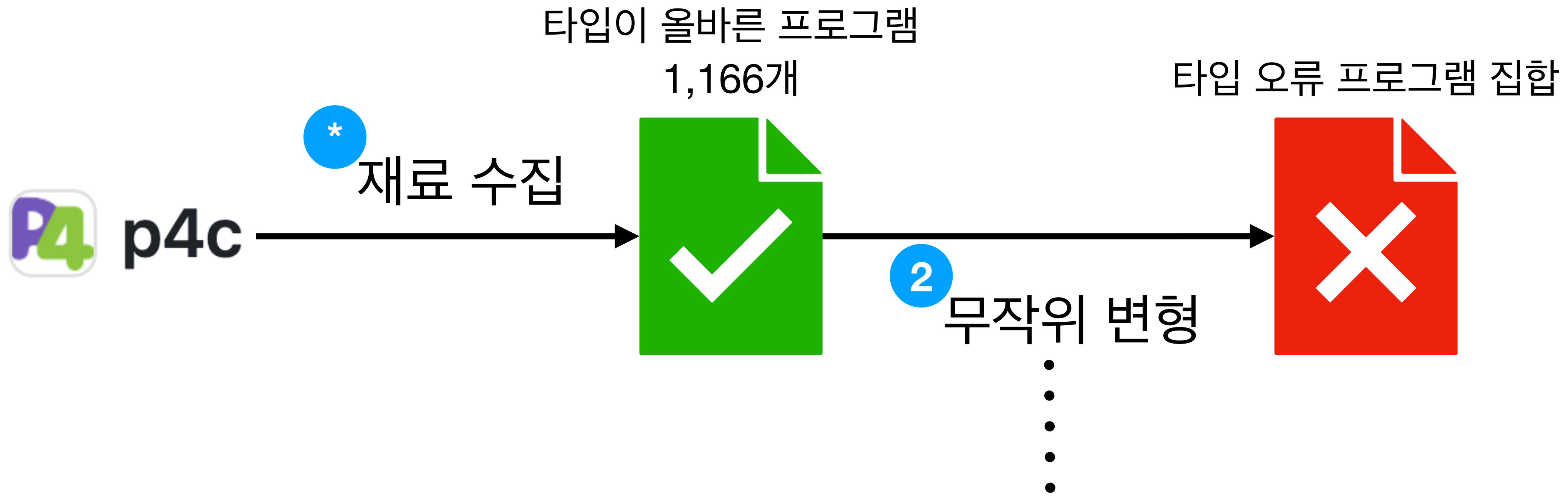
짜깁 없는 조건 위반

```
-- if $is_int(type_r)  
    => $is_non_negative(C, expr_r)
```

# 커버리지 기반 타입 오류 테스트 마구생성 (Fuzzing)



# 무엇을 변형할까?



1. 어떤 타입-올바른 프로그램을 변형할 것인가?
2. 타입-올바른 프로그램의 어떤 부분을 변형할 것인가?

# 1. 어떤 타입-올바른 프로그램을 변형할 것인가?

```
bit<8> shl(in bit<8> x,  
           in bit<8> y) {  
    return x << y;  
}
```

명세 실행기



```
-- if $is_int(type_r)  
    => $is_non_negative(C, expr_r)
```

위반하고자 하는 짝꿍 없는 조건에 도달했지만, 아직 위반하지는 못함

무작위 변형하면 아마도 조건을 위반할 수 있을 것

위반하고자 하는 조건마다, "한 곳 차이" 프로그램을 수집

## 2. 한 곳 차이 프로그램의 어떤 부분을 변형할 것인가?

```
bit<8> shl(in bit<8> x,  
           in bit<8> y) {  
    return x << y;  
}
```

명세 실행기



⋮

```
-- if $is_int(type_r)  
=> $is_non_negative(C, expr_r)
```

왜 이 짝꿍 없는 조건을 만족했을까?

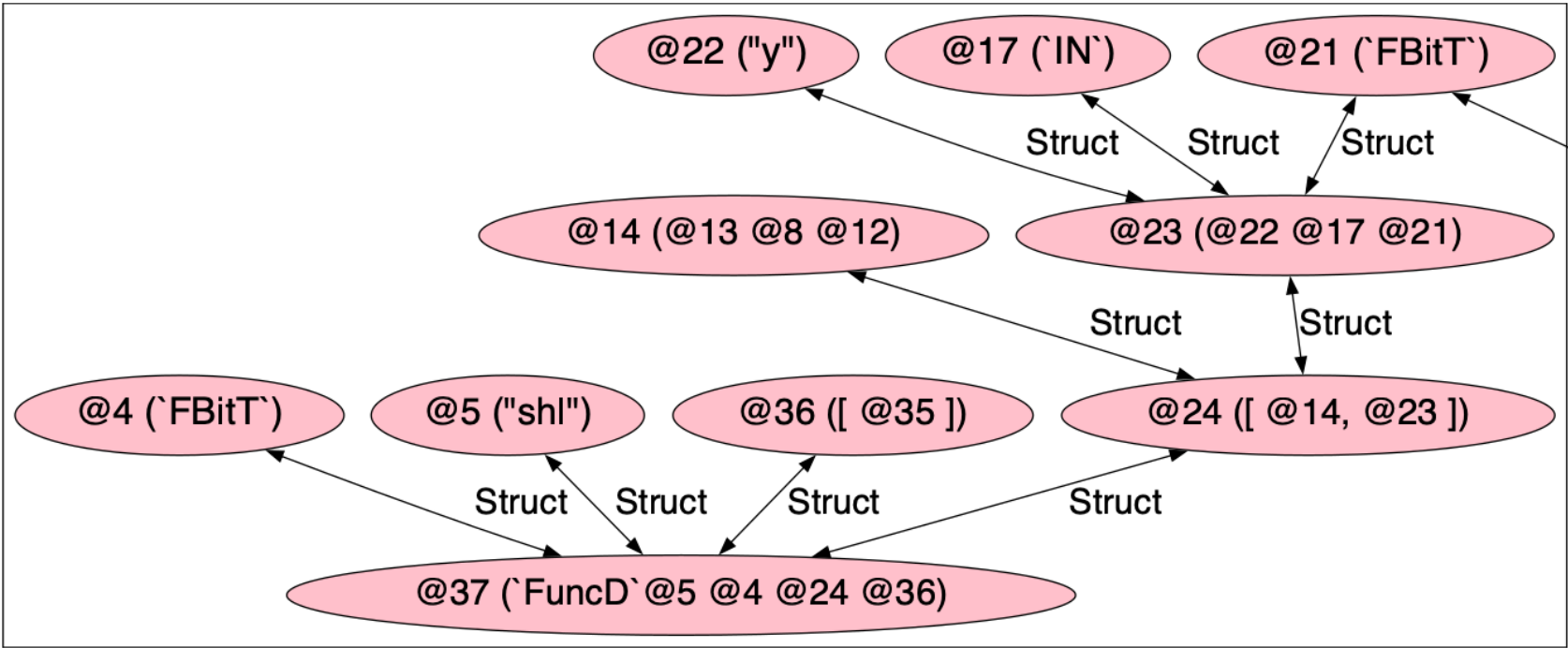
명세 실행기로부터 그 이유를 알아낼 수 있을까?

# 명세 실행기 관점에서 보면: 값에서 값으로

AST 값의 집합

P4 프로그램 AST

명세 실행기



```
-- if $is_int(type_r)
=> $is_non_negative(C, expr_r)
```

TRUE 값

⋮

TRUE 값과 가장 관련 있는 AST 값이 무엇일까?

# 명세 실행기 관점에서 보면: 값에서 값으로

AST 값의 집합

P4 프로그램 AST

명세 실행기

TRUE 값

```
-- if $is_int(type_r)  
=> $is_non_negative(C, expr_r)
```

1 왜 TRUE 가 나왔을까?

FALSE => FALSE 가 TRUE 이기 때문

# 값의 의존 관계를 그래프로 표현하면

AST 값의 집합

P4 프로그램 AST

명세 실행기

TRUE 값

```
-- if $is_int(type_r)  
=> $is_non_negative(C, expr_r)
```

`$is_int(type_r)`

@280 (false)

@285 (false)

`$is_non_negative(C, expr_r)`

@286 (true)

1

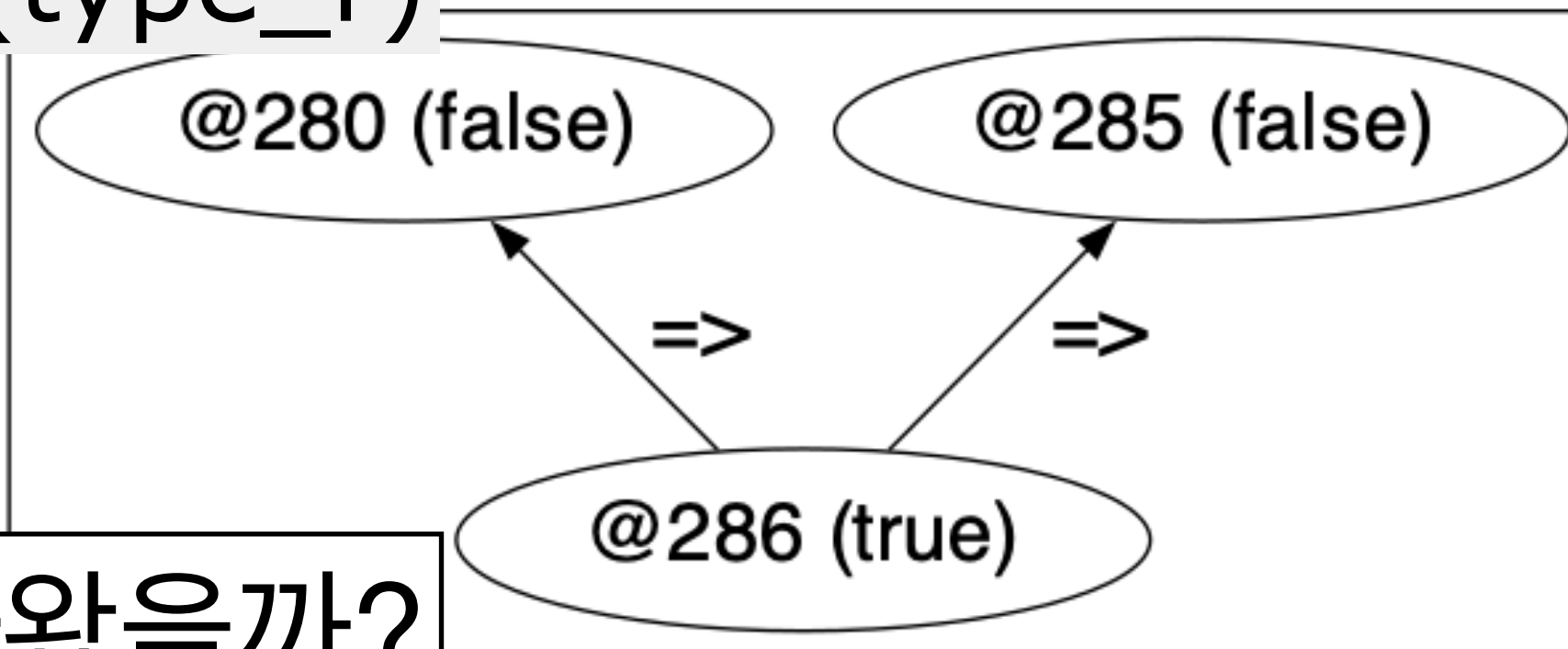
왜 TRUE 가 나왔을까?

# 값의 의존 관계를 거슬러 올라가기

2

왜 **FALSE** 가 나왔을까?

`$is_int(type_r)`



1

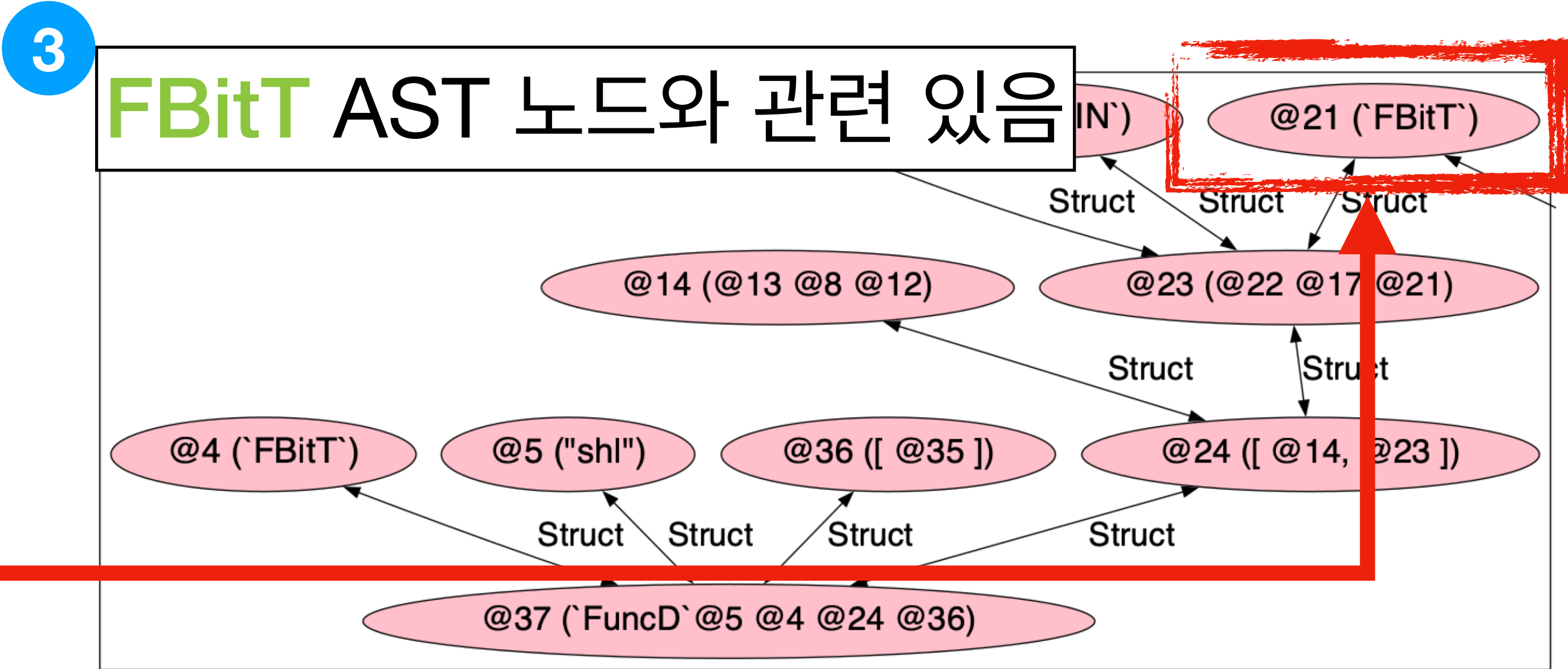
왜 **TRUE** 가 나왔을까?

# 값의 의존 관계를 거슬러 올라가기

2 왜 FALSE 가 나왔을까?

`$is_int(type_r)`

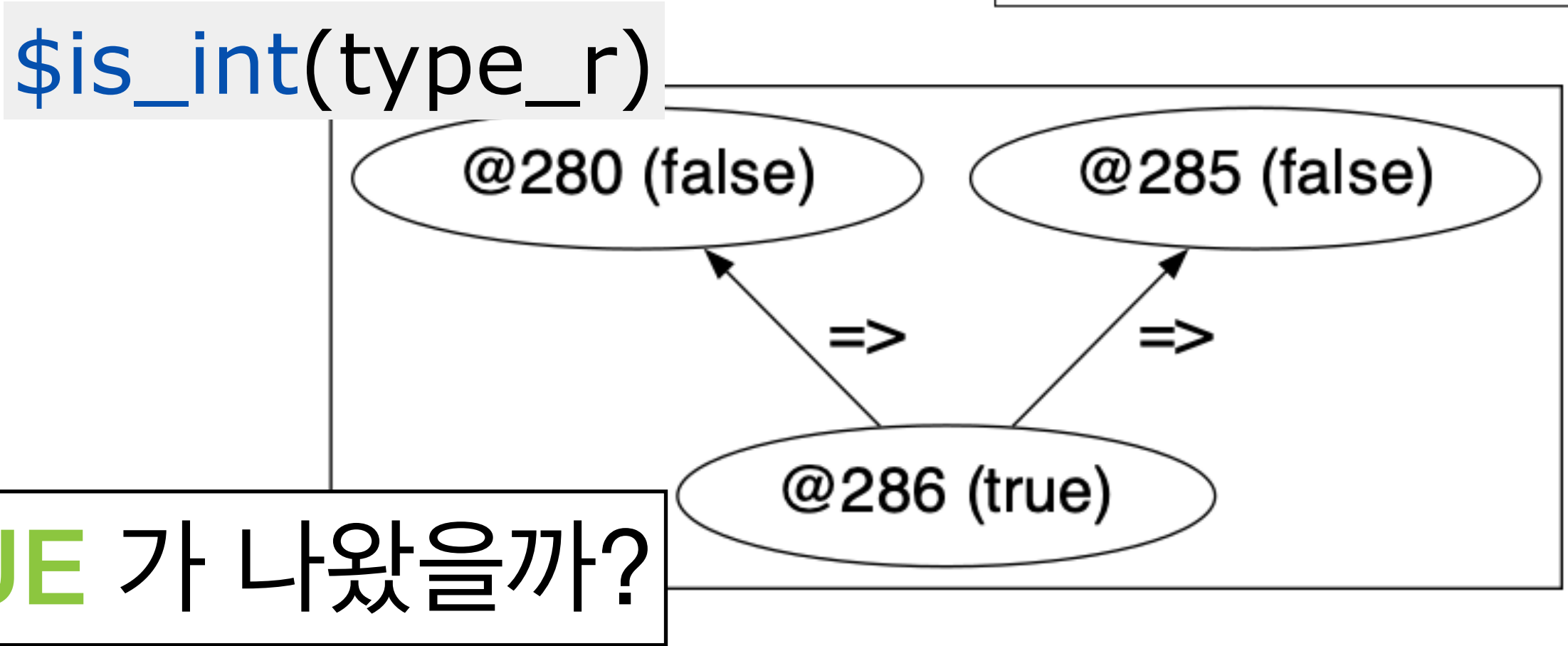
1 왜 TRUE 가 나왔을까?



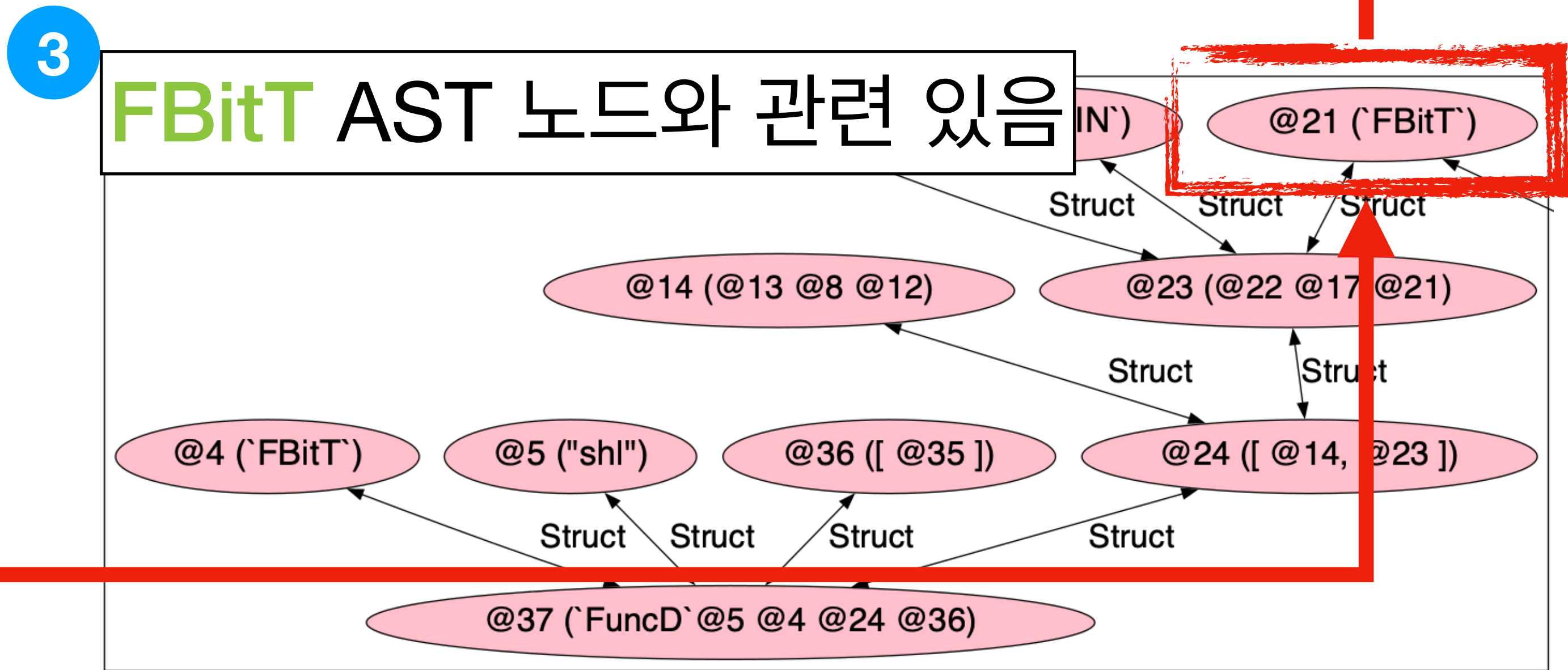
4 프로그램의 `bit<8>`과 관련 있음

```
bit<8> shl(in bit<8> x,  
           in bit<8> y) {  
    return x << y;  
}
```

2 왜 **FALSE** 가 나왔을까?



1 왜 **TRUE** 가 나왔을까?



4 프로그램의 `bit<8>`과 관련 있음

```
bit<8> shl(in bit<8> x,  
           in bit<8> y) {  
    return x << y;  
}
```

5 무작위 변형

```
bit<8> shl(in bit<8> x,  
           in int<8> y) {  
    return x << y;  
}
```

6 타입 오류 테스트

# 값-의존 그래프를 활용해 변형할 AST 노드 찾기

```
bit<8> shl(in bit<8> x,  
           in bit<8> y) {  
    return x << y;  
}
```

명세 실행기



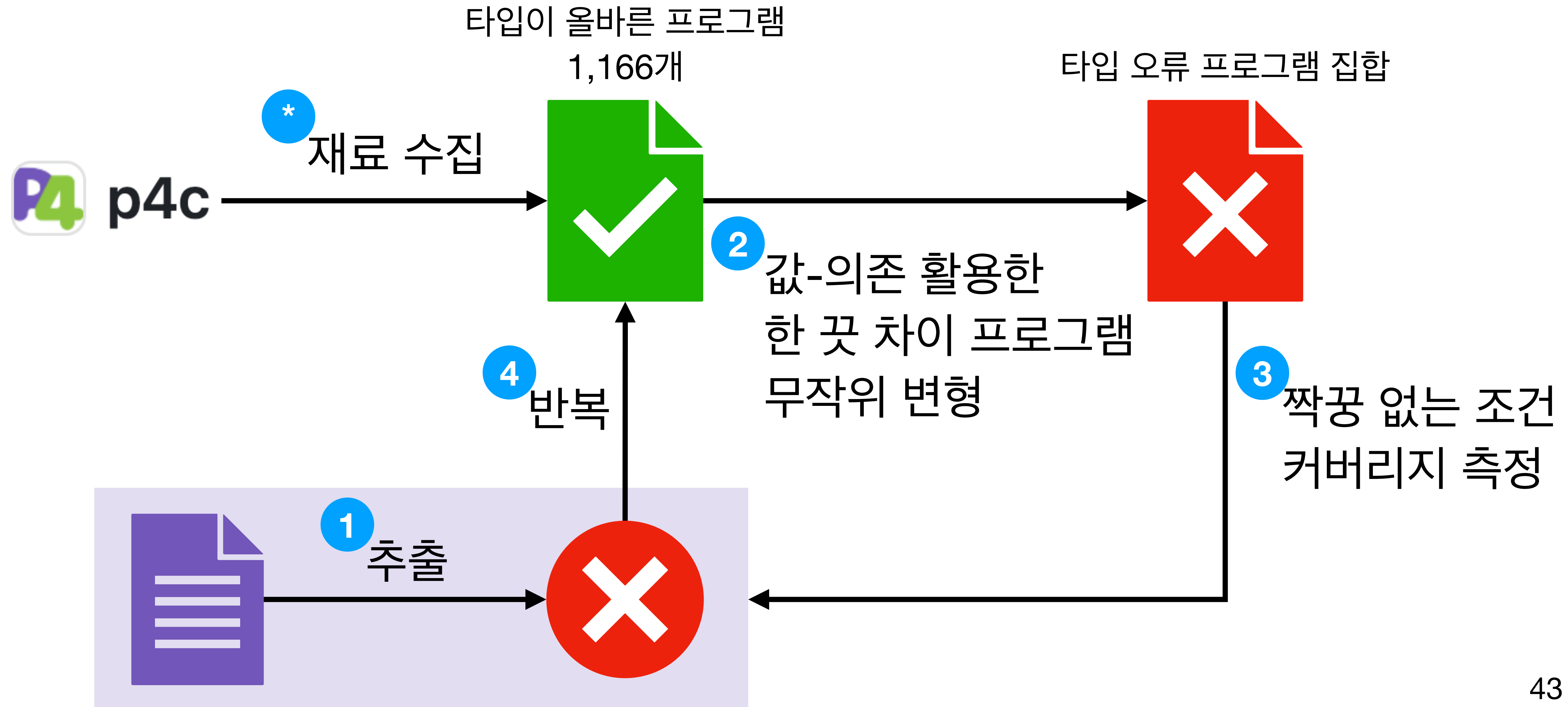
```
-- if $is_int(type_r)  
=> $is_non_negative(C, expr_r)
```

값-의존 그래프 (Value Dependency Graph)

명세를 실행하면서 값-의존 그래프도 같이 생성

조건을 실행 결과가 왜 **TRUE**가 되었는지 역추적

# 커버리지 기반 타입 오류 테스트 마구생성 (Fuzzing)



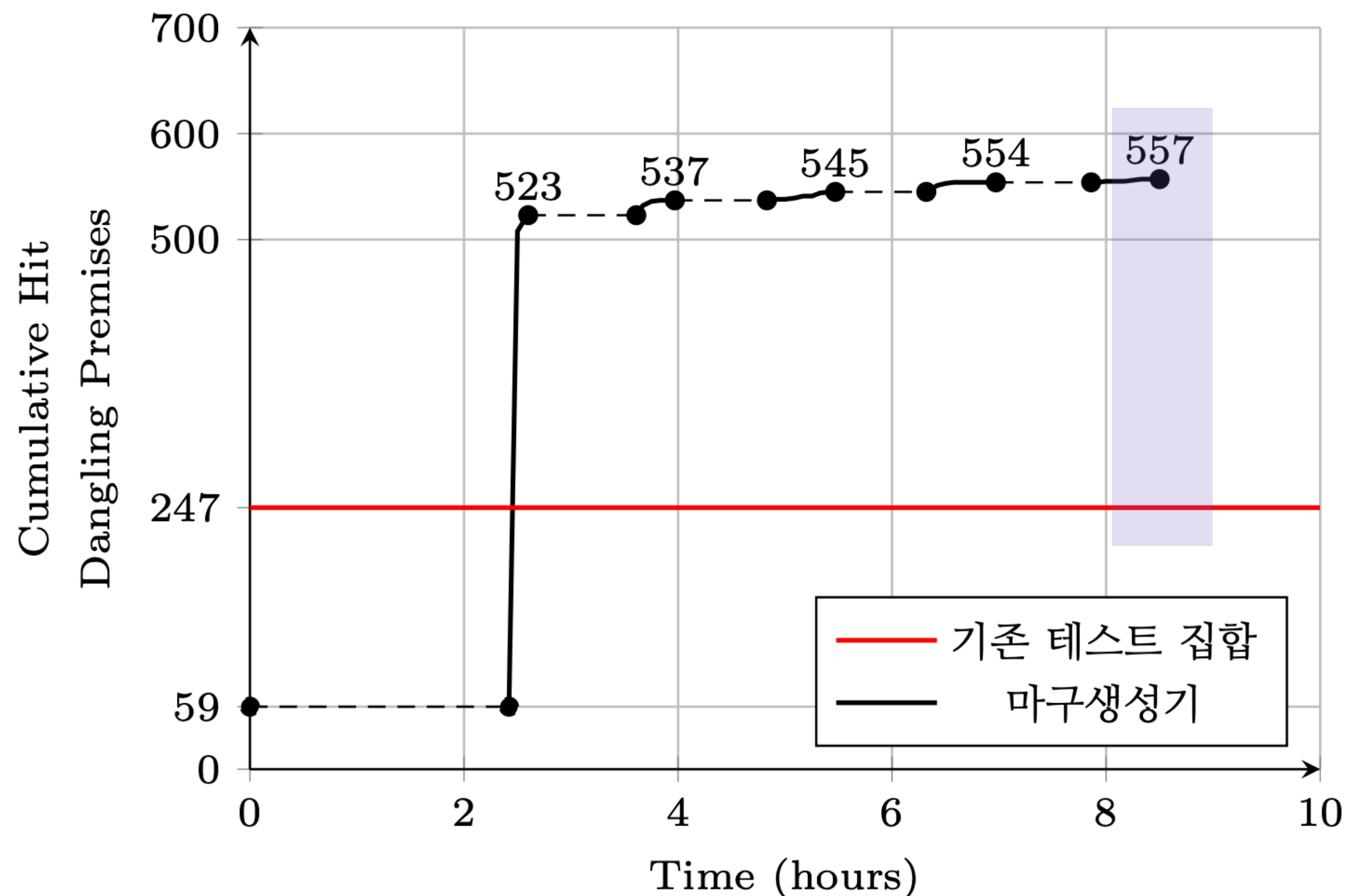
# 짜꿍 없는 조건 커버리지와 마구생성기 평가

1. 기존 타입 오류 테스트 집합 대비 더 높은 짜꿍 없는 조건 커버리지를 달성하는가?
2. 생성한 타입 오류 테스트 집합으로 공식 컴파일러의 결함을 발견할 수 있는가?
3. 명세 기반 짜꿍 없는 조건 커버리지가 컴파일러 소스코드 기반 분기 커버리지보다 타입 오류 테스트에 대해 더 민감한가?

# 1. 기존보다 더 높은 짝꿍 없는 조건 커버리지를 달성하는가?

명세로부터 짝꿍 없는 조건 939개 추출

공식 컴파일러의 타입-올바름 테스트 집합(1,166개)를 입력으로 10시간 동안 마구실행



- 타입 오류 테스트 422개 생성
- 기존 테스트 집합: 247/939 (26.30%)
- 생성한 테스트 집합: 557/939 (**59.32%**)

## 2. 공식 P4 컴파일러의 타입 검사기에서 결함을 발견할 수 있는가?

생성한 타입 오류 테스트 집합



P4 공식 컴파일러의 타입 검사기

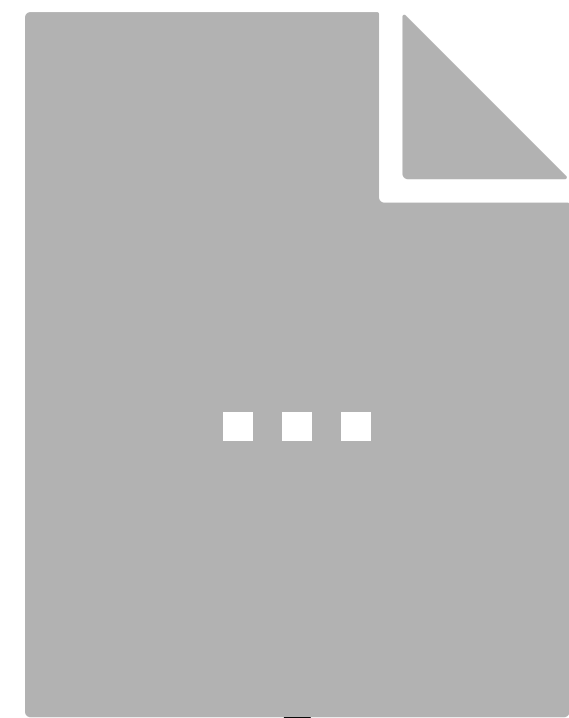


타입 안전성 버그 15개

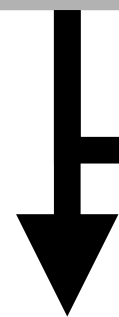
비정상 종료 버그 14개

### 3. 소스코드 기반 분기 커버리지와 짝꿍 없는 조건 커버리지 비교

기존 컴파일러 테스트 집합



- 타입-올바름 테스트 1,166개
- 타입 오류 테스트 254개



p4c

소스코드 분기 커버리지

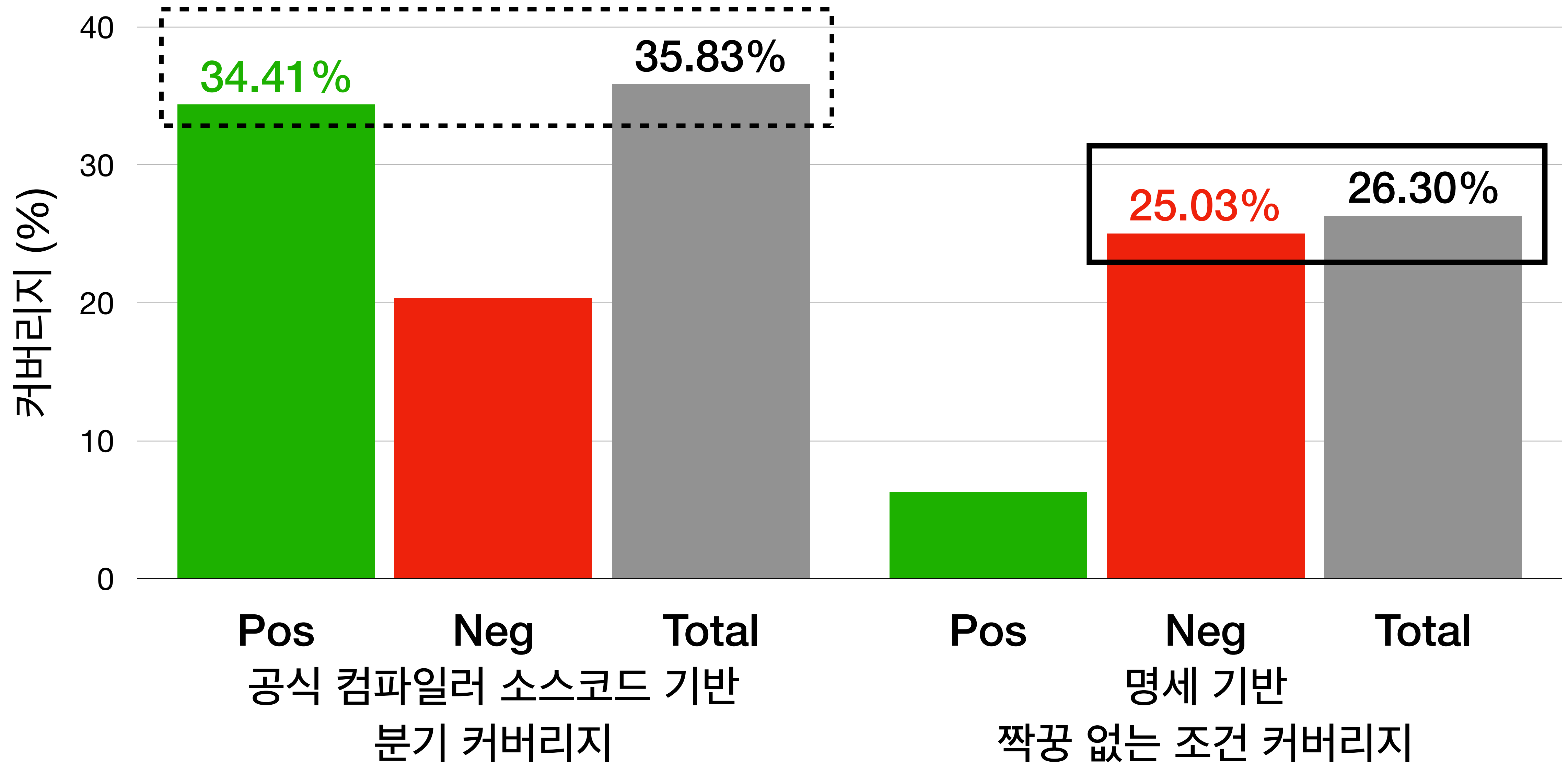
P4 기계화 명세



짝꿍 없는 조건 커버리지

각 커버리지 기준이 타입 오류 테스트에 대해서 민감한가?

### 3. 소스코드 기반 분기 커버리지와 짝궁 없는 조건 커버리지 비교



기계화 명세를 바탕으로 타입 오류 테스트 집합의 품질을 정량화하는  
짜깁 없는 조건 커버리지 (dangling coverage)를 정의하고,  
커버리지 기반 마구생성 (fuzzing)을 통해 P4 언어의 타입 오류 조건을  
최대한 많이 검사하는 타입 오류 테스트 집합 자동 생성

# 보조 자료

# 생김새 활용의 한계점: 거짓-짝꿍 없는 조건

P4 기계화 명세



거짓-짝꿍 없는 조건 (False Positive)

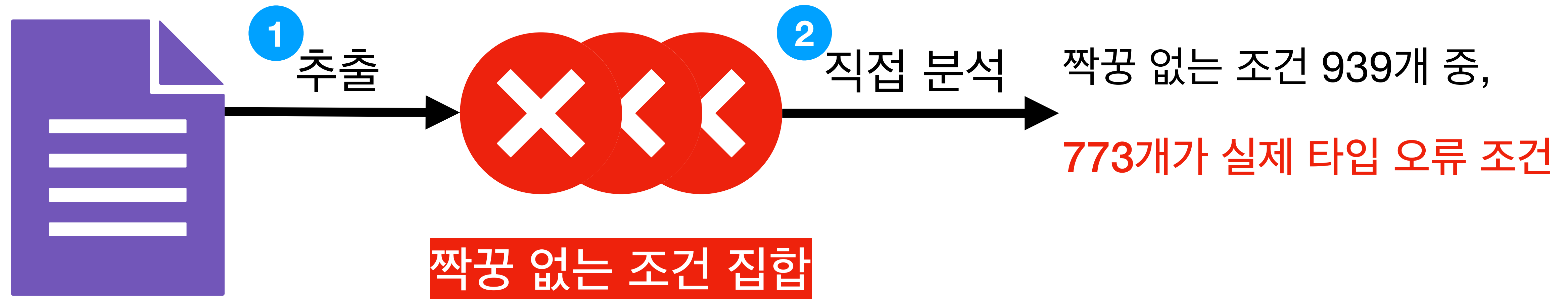
```
-- if binop ∈ { SHL, SHR }
```

```
-- if binop = PLUS ∨ binop = MINUS
```

실제로는 짝꿍이지만, 생김새가 달라서 걸러내지 못함

# 짜깁 없는 조건 중 실제 타입 오류 조건 개수는?

P4 기계화 명세



# 프로그램을 어떻게 변형할 것인가?

어떤 프로그램의 어떤 부분(AST 노드)을 변형할지 찾은 후에, 셋 중 하나 적용:

- ▶ 명세-타입 변환 (Meta-type driven)

명세 상의 문법 정의에 따라 값 변형, e.g., **BitT** (type) → **IntT** (type)

- ▶ 리스트 변환

리스트에 중복 원소 삽입 / 원소 삭제 / 순서 뒤바꾸기, e.g., [ 1; 2; 3 ] → [ 2; 3; 1 ]

- ▶ 끼리끼리 변환 (Constructor)

서로 비슷한 명세 문법끼리 변환, e.g., **IntE** 42 → **BitE** 42

# 1. 기존보다 더 높은 짝꿍 없는 조건 커버리지를 달성하는가?

생성한 테스트 집합: 557/939 (59.32%)

324

233

...14

기존 테스트 집합: 247/939 (26.30%)

## 2. 타입 안전성 버그 예시

### 8.12.1. Explicit casts

The following casts are legal in P4:

- `int<W>` → `bit<W>`

너비가 W로 같을 때만

`int<W>`에서 `bit<W>`으로 형변환을 허용한다.

```
const bit<32> two = 32w2;  
const bit<6> twofour = ((bit<6>) (((bit<0>) (((int<32>) (two)))))));
```

⋮

`int<32>`에서 `bit<0>`으로 형변환을 잘못 허용함

### 3. 짝꿍 없는 조건 커버리지와 소스코드 기반 분기 커버리지 비교

