정적 분석으로 코드 합성 보조하기

윤용호, 2021-12-17

내용

- 코드 합성이 무엇인지 간단히 소개
- 코드 합성에 정적 분석을 활용하는 방법
- 진행중인 정적 분석 설계

코드 합성?

조건을 만족하는 코드를 자동으로 만들기

```
f(1) = 3,
f(2) = 6,
 f(x) = ?
```

→

```
f(x) = ?
```

코드 합성?

조건을 만족하는 코드를 자동으로 만들기

```
f(1) = 3,
f(2) = 6,
 f(x) = ?
```

```
f(x) = 3 * x
```

코드 합성?

조건을 만족하는 코드를 자동으로 만들기

```
f(1) = 3,
f(2) = 6,
f(x) = ?
```

→

```
f(x) = 3 * x ?
f(x) = x * x + 2 ?
```

코드 합성?

조건을 만족하는 코드를 자동으로 만들기

```
f(1) = 3,
f(2) = 6,
f(3) = 9,
 f(x) = ?
```

→

```
f(x) = 3 * x
```

~~f(x) = x * x + 2 ?~~

코드 합성?

조건을 만족하는 코드를 자동으로 만들기

```
f(1) = 3,
f(2) = 6,
f(3) = 11,
f(x) = ?
```

→

~~f(x) = 3 * x~~

```
f(x) = x * x + 2 ?
```

입출력 예제 기반 코드 합성

- 합성 조건이 언제나 꼭 입출력 예제에만 국한되는 것은 아니지만, 꽤 유용함

- 이게 왜 되지?
  - 보통의 사람들이 프로그램을 짜고서 확인할 때도 대개
몇 가지 테스트 케이스에 대해 잘 동작하면 맞게 만들었나보다 생각함
  - “오컴의 면도날”

기계적으로 코드 합성하기

작은 부품부터 조립해나가기(Bottom up)

너무 많은 부품을 조립해봐야한다

...

`1+1`   `1+x`   `1+(-x)`   `x+(-1)`   `x*x`

`-1`   `-x`

`0`   `1`   `x`

$$
\begin{array}{rcl}
E & \rightarrow & 0 \\
 & | & 1 \\
 & | & x \\
 & | & E + E \\
 & | & E * E \\
 & | & -E
\end{array}
$$

기계적으로 코드 합성하기

상위 문법 규칙부터 나열해나가기(Top down)

```
              E

0    1    x    E+E   E*E    -E

     1+1  1+x  x+x    E*(-E)

     -E+E   E*E+E    E*(E+E)

              ...
```

중간중간 불가능한 후보를 잘라낼 수 있지만
여전히 너무 많은 공간을 탐색해야한다

$$
\begin{array}{rcl}
E & \rightarrow & 0 \\
 & | & 1 \\
 & | & x \\
 & | & E + E \\
 & | & E * E \\
 & | & -E
\end{array}
$$

기계적으로 코드 합성하기

양방향으로 합성하기(Bidirectional)*

```
                E
   0     1     x     E+E   E*E    -E
      1+1  1+x  x+x    E*(-E)
        -E+E  E*E+E    E*(E+E)
```

뼈대는 위에서 아래로

... 뼈대의 빈칸을 부품으로 채워본다

부품은 아래서 위로

```
 1+1   1+x   1+(-x)   x+(-1)   x*x

             -1     -x

           0     1     x
```

$$
\begin{array}{rcl}
E & \rightarrow & 0 \\
 & | & 1 \\
 & | & x \\
 & | & E + E \\
 & | & E * E \\
 & | & -E
\end{array}
$$

*(POPL'21) Woosuk Lee, "Combining the Top-Down Propagation and Bottom-Up Enumeration for Inductive Program Synthesis"

기계적으로 코드 합성 더 잘해볼 아이디어

정적 분석을 활용하여

- 글러먹은 뼈대를 빨리 판단해서 버리면 유리함
    - 글러먹은 = 아직 미완성 프로그램이지만 빈 칸에 어떤 부품을 끼워봐도 조건을 만족하지 못할 것이 확실한
    - 후보 부품 갯수가 많을 수록 버리기의 효과 큼
- 정적 분석으로 잘 할 수 있지 않을까?
    - 입력: 조건(입출력 쌍)과 미완성 프로그램
    - 출력: “정답일수도 있음” or “글러먹음”

합성에 정적 분석을 활용할 수 있는 예

조건:
f(x) = “x가 8개의 비트 나열일 때,
x의 가장 낮은 자리부터 처음으로 0이 나오기 직전까지
쭉 연속된 1만 그대로 남기고 그 위는 모두 0으로 채운 값”

정답:
((x + 1) ^ x) >> 1

입출력 예제:

```
f(b_11001111) = b_00001111
f(b_01010111) = b_00000111
```

후보 미완성 프로그램:
([] | x) >> 1

정방향 분석(x=b_11001111):

```
[]              : TTTTTTTT
x               : 11001111
[] | x          : 11TT1111
([] | x) >> 1 : T11TT111
```

T = 임의의 비트가 가능
. = 어떤 비트도 불가능
0 = 이 위치엔 0만 가능
1 = 이 위치엔 1만 가능

합성에 정적 분석을 활용할 수 있는 예

조건:
f(x) = “x가 8개의 비트 나열일 때,
x의 가장 낮은 자리부터 처음으로 0이 나오기 직전까지
쭉 연속된 1만 그대로 남기고 그 위는 모두 0으로 채운 값”

정답:
((x + 1) ^ x) >> 1

입출력 예제:
f(b_11001111) = b_00001111
f(b_01010111) = b_00000111

후보 미완성 프로그램:
([] | x) >> 1

정방향 분석(x=b_11001111):

```
[]              : TTTTTTTT
x               : 11001111
[] | x          : 11TT1111
([] | x) >> 1 : T11TT111
```

역방향 분석(x=b_11001111):

```
([] | x) >> 1 : T11TT111   ∧ 00001111 = 0..01111 <- 걸러먹음
[] | x          : 11TT1111
x               : 11001111
[]              : TTTTTTTT
```

합성에 정적 분석을 활용할 수 있는 예

조건:
f(x) = “x가 8개의 비트 나열일 때,
x의 가장 낮은 자리부터 처음으로 0이 나오기 직전까지
쭉 연속된 1만 그대로 남기고 그 위는 모두 0으로 채운 값”

정답:
((x + 1) ^ x) >> 1

입출력 예제:
```
f(b_11001111) = b_00001111
f(b_01010111) = b_00000111
```

후보 미완성 프로그램:
```
([] & x) >> 1
```

정방향 분석(x=b_11001111):

```
[]              : TTTTTTTT
x               : 11001111
[] & x          : TT00TTTT
([] & x) >> 1 : TTT00TTT
```

역방향 분석(x=b_11001111):

```
([] & x) >> 1 : TTT00TTT   ∧ 00001111 = 0000.111 <- 글러먹음
[] & x          : TT00TTTT
x               : 11001111
[]              : TTTTTTTT
```

합성에 정적 분석을 활용할 수 있는 예

조건:
f(x) = “x가 8개의 비트 나열일 때,
x의 가장 낮은 자리부터 처음으로 0이 나오기 직전까지
쭉 연속된 1만 그대로 남기고 그 위는 모두 0으로 채운 값”

정답:
((x + 1) ^ x) >> 1

입출력 예제:

```
f(b_11001111) = b_00001111
f(b_01010111) = b_00000111
```

후보 미완성 프로그램:
([] / x) >> 1

정방향 분석(x=b_11001111):

```
[]              : TTTTTTTT
x               : 11001111
[] / x          : TTTTTTTT
([] / x) >> 1 : TTTTTTTT
```

역방향 분석(x=b_11001111):

```
([] / x) >> 1 : TTTTTTTT  ∧ 00001111 = 00001111
[] / x          : TTTTTTTT  ∧ 0001111T = 0001111T (30 or 31)
x               : 11001111  ∧ 0000TTTT ([0,8]) = ..001111 <- 글러먹음
[]              : TTTTTTTT
```

합성에 정적 분석을 활용할 수 있는 예

조건:
f(x) = “x가 8개의 비트 나열일 때,
x의 가장 낮은 자리부터 처음으로 0이 나오기 직전까지
쭉 연속된 1만 그대로 남기고 그 위는 모두 0으로 채운 값”

정답:
((x + 1) ^ x) >> 1

입출력 예제:
f(b_11001111) = b_00001111
f(b_01010111) = b_00000111

후보 미완성 프로그램:
([] ^ x) >> 1

정방향 분석(x=b_11001111):

```
[]             : TTTTTTTT
x              : 11001111
[] ^ x         : TTTTTTTT
([] ^ x) >> 1  : TTTTTTTT
```

역방향 분석(x=b_11001111): 정답일수도 있음

```
([] ^ x) >> 1  : TTTTTTTT  ∧ 00001111 = 00001111
[] ^ x         : TTTTTTTT  ∧ 0001111T = 0001111T
x              : 11001111
[]             : TTTTTTTT  ∧ 1101000T = 1101000T
```

합성에 활용하기 좋은 정적 분석이 되려면

- 미완성 프로그램을 분석할 수 있어야
- 역방향 분석(Backward Analysis)이 필수이자 핵심: 정방향 분석은 미완성 부분에 의해 Top이 많이 발생
- 최대한 빠르고 정확한 분석으로 불가능한 후보를 쳐내야
    - 불가능한 후보 = 요약 값에 bottom이 존재하면 (왜? 구체화 해보면 가능한 값이 없게 되므로)
    - **최대한 정확한 분석: 도메인과 (역방향)요약 연산을 정교하게 정의해야**
    - 최대한 빠른 분석: 온갖 부품을 마구 끼워서 검사해보는 것보다 미완성 뼈대를 분석해서 버리는 게 더 비싸면 손해
- 희망
    - 대개 합성 대상 프로그램 크기가 비교적 작음 -> 분석 비용이 싸다
    - 아직은 합성 대상 언어에 반복문이 없음 -> 분석 정확도를 높이기 편리하다

미완성 프로그램 정적 분석 대상 언어

- SyGuS (Syntax-Guided Synthesis) 의 합성 대상 언어 중 BitVector 관련 부분
  - SyGuS는 합성 분야의 사실상 표준
  - BitVector 에서 잘 풀리면 String 등으로 확장 예정

- 값: 64비트 정수(64개의 비트 나열)
- 연산: 널리 쓰이는 산술 및 비트 연산
- [ ]: 프로그램의 아직 완성되지 않은 부분
- 모든 변수는 입력값 (let 변수 선언 없음)

$$
\begin{array}{rcl}
E & \rightarrow & 0 \mid 1 \mid -1 \\
 & \mid & x \\
 & \mid & \diamond E \\
 & \mid & E \circ E \\
 & \mid & \texttt{if}\ B\ E\ E \\
 & \mid & [] \\
B & \rightarrow & E = E \\
 & \mid & \texttt{not}\ B \\
 & \mid & B\ \texttt{and}\ B \\
 & \mid & B\ \texttt{or}\ B \\
 & \mid & [] \\
\diamond & \rightarrow & \sim \mid - \\
\circ & \rightarrow & \wedge \mid \vee \mid \oplus \mid \ll \mid \gg \mid \ggg \\
 & & \mid + \mid - \mid * \mid / \mid \% \mid /_s \mid \%_s
\end{array}
$$

상호 보완하는 복합 도메인

Reduced Product Domain

- 부호 없는 구간, 부호 있는 구간, 요약 비트 나열 도메인을 모두 활용
  - 구체화 함수는 각 부분 도메인의 구체화 함수 결과의 교집합
  - 예: {b_11111100, b_11111110} 을 요약하면 ([252, 254], [-4, -2], 111111T0)
- 각 부분 도메인이 서로의 정확도를 보완 (Reduction Operator)
  - 비트 연산 결과에서는 구간 값, 산술 연산 결과에서는 비트 값의 정확도가 떨어지는데, 떨어진 정확도를 정확도 높은 도메인의 결과를 이용해 보완할 수 있다
  - 예: 요약 연산 결과가 (Top, Top, T0001T1T) 이었다면 ([10, 143], [-118, 15], T0001T1T) 로 구간 값을 살릴 수 있다

상호 보완 함수

Reduction Operator

- 비트 나열 도메인
  - 부호 없는 구간 도메인에서
    - 양끝값을 비트로 표현한 후 최상위 위치부터 순서대로 동일한 비트들은 그 값으로, 처음으로 달라지는 위치부터 이후 더 낮은 자리는 모두 T 으로
  - 부호 있는 구간 도메인에서
    - 양끝값의 부호가 같으면 부호 없는 구간 도메인과 같은 방법
    - 양끝값 부호가 다르면 활용 포기 (구간에 111…1 과 000…0 을 포함하게 되므로)

상호 보완 함수

Reduction Operator

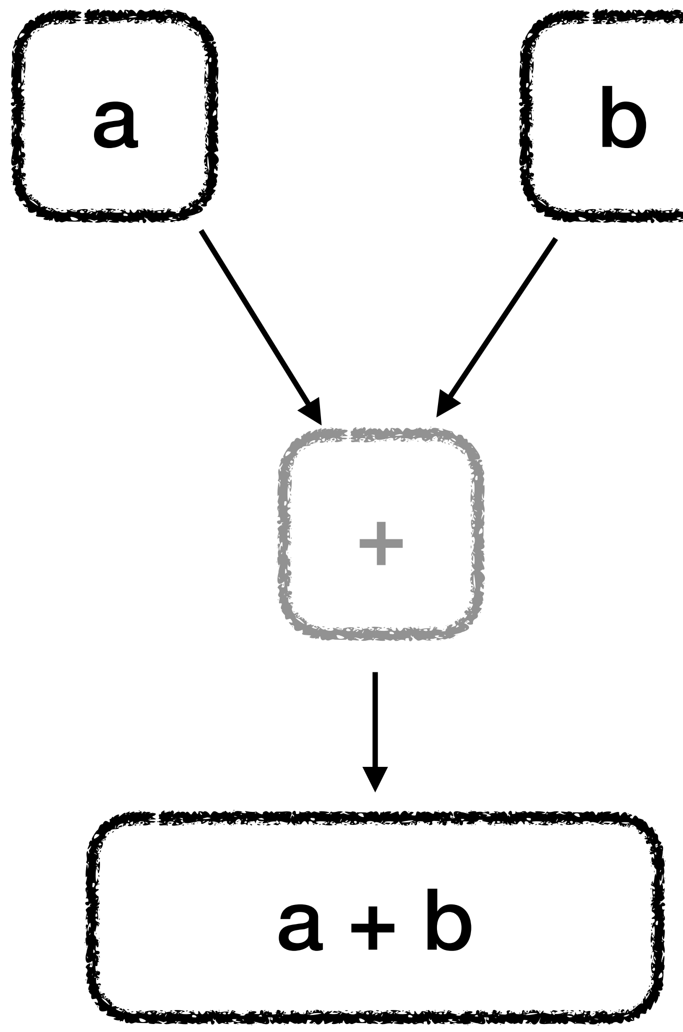
- 부호 없는 구간 도메인
  - 비트 나열 도메인에서: T 비트를 모두 0으로 만들면 최소, 1로 만들면 최대
  - 부호 있는 구간 도메인에서
    - 양끝값 부호가 동일하면 양끝값의 비트 표현을 부호없이 해석해서 사용
    - 양끝값 부호가 다르면 활용 포기 (구간에 111…1 과 000…0 을 포함하게 되므로)

상호 보완 함수
Reduction Operator

- 부호 있는 구간 도메인
  - 비트 나열 도메인에서
    - 최상위 비트가 0이나 1이면 부호 없는 구간 도메인과 똑같이
    - 최상위 비트가 T이면
      - 최소: 최상위비트는 1, 나머지 T 비트는 0으로 채운 것
      - 최대: 최상위 비트는 1, 나머지 T 비트는 1로 채운 것
  - 부호 없는 구간 도메인에서
    - 양끝값을 비트로 표현했을 때 최상위 비트가 동일하면: 양끝값 비트 표현을 부호 있는 것으로 해석하여 사용
    - 최상위 비트가 다르면 활용 포기 (구간에 100….0 과 011…1 을 포함하게 되므로)

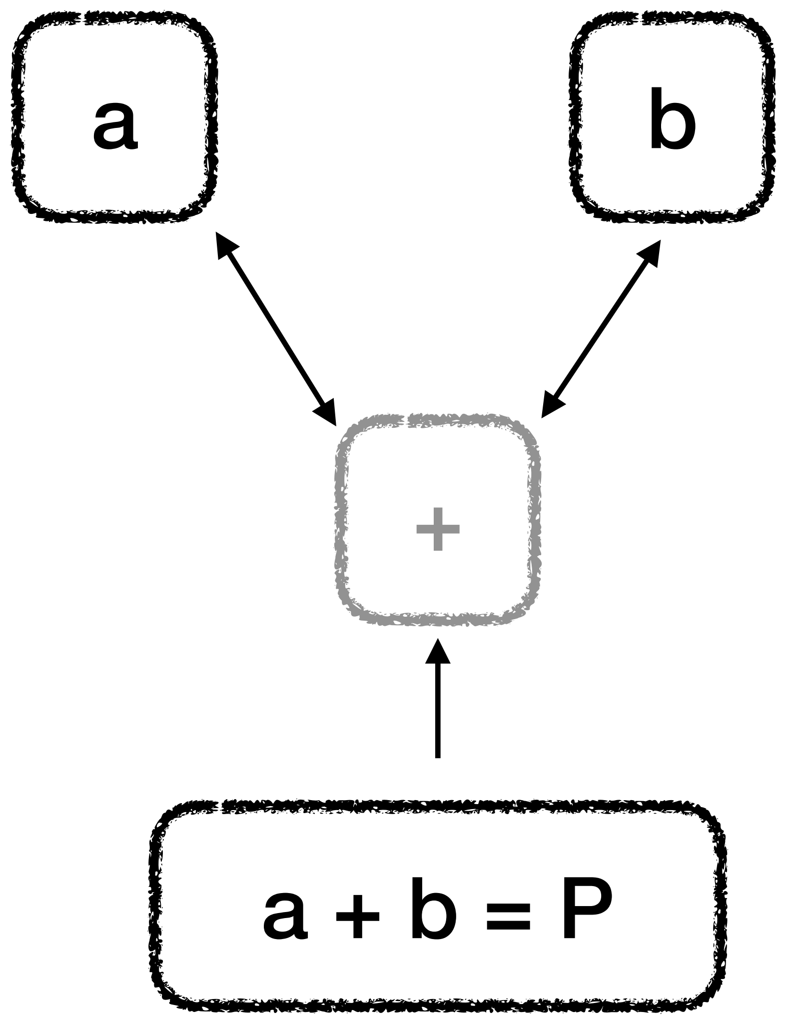
정방향/역방향 분석

정방향



a의 요약을 계산하고
b의 요약을 계산하고
그것으로 a+b의 요약을 계산

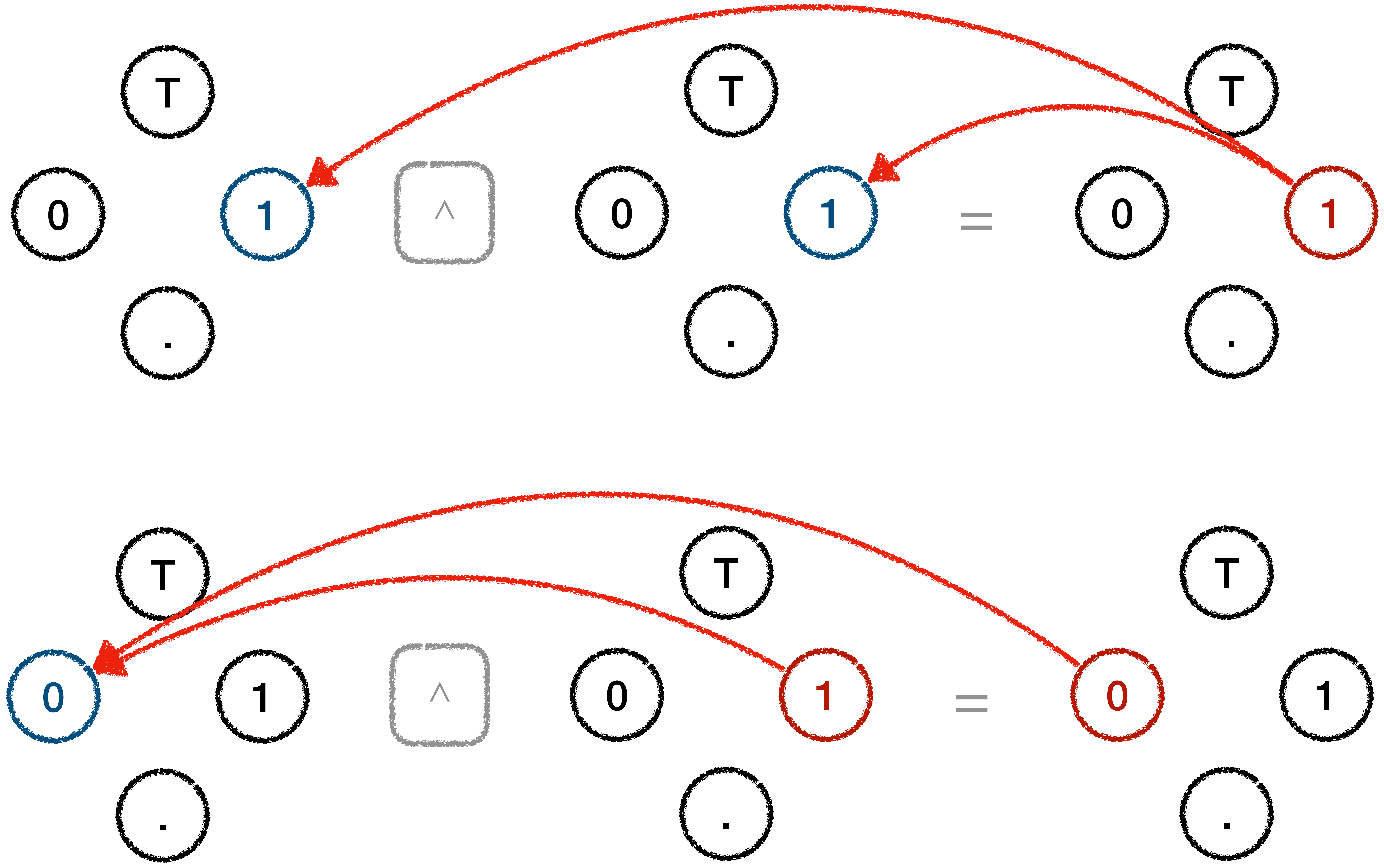
역방향



a+b가 만족해야할 조건 P가 주어졌을 때
a, b 각각의 요약이 만족해야할 조건을 계산
(a, b의 정방향 요약이 계산되어있으면 활용 가능)

역방향 분석 연산: and(∧)

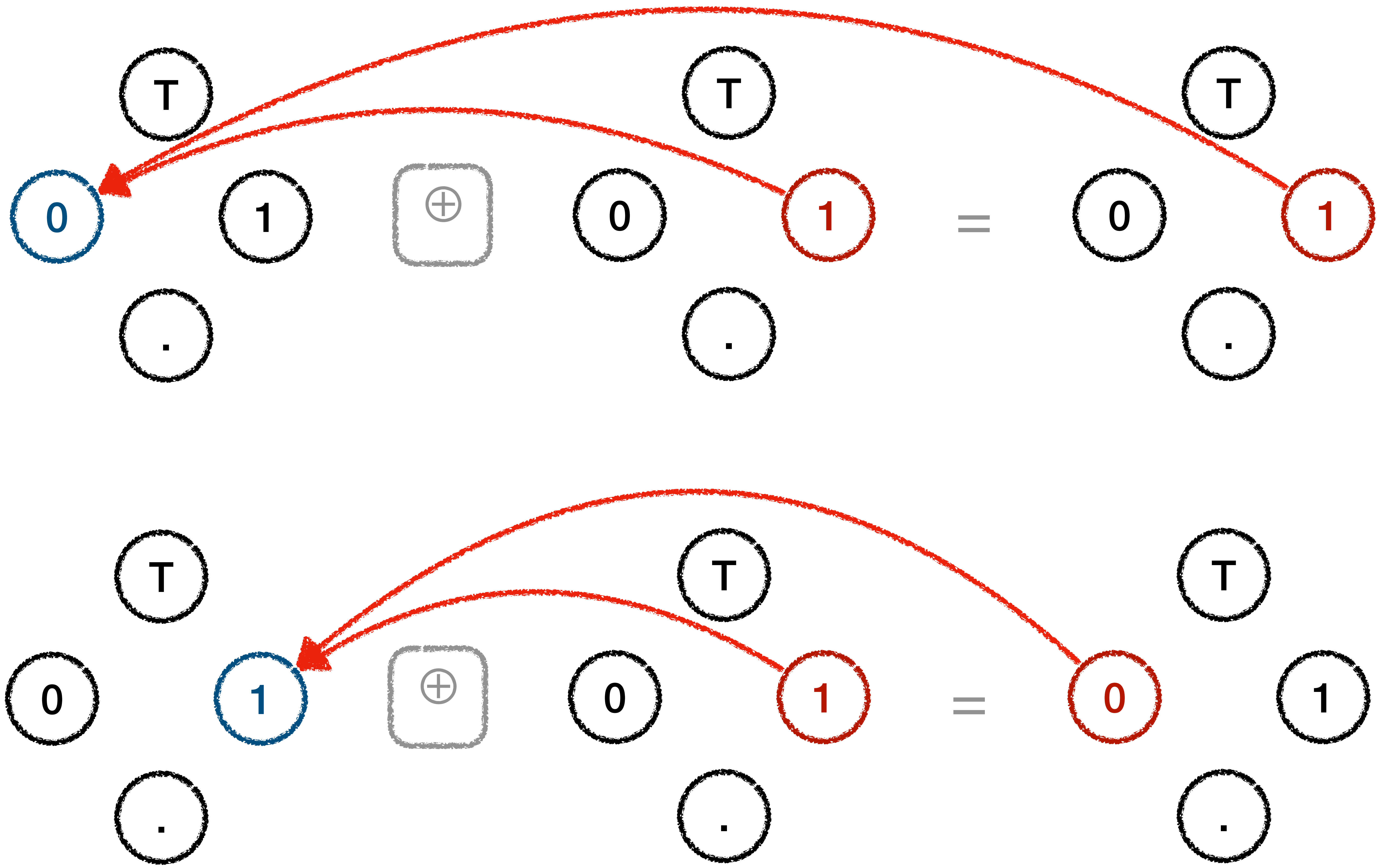
각 비트 자리마다



정방향 결과	역방향 결과
TTTTTTTT	T10T0T1T
TT10101T	T110101T
	T100T01T

역방향 분석 연산: xor(⊕)

각 비트 자리마다



정방향 결과	역방향 결과
T0T0TT1T	T010TT1T
TT10100T	T110100T
	T100T01T

역방향 분석 연산: lshr(>>>)

- a >>> b = P 가 주어졌을 때
- P의 1이 아닌 연속된 상위 비트 갯수로 b의 최댓값 제한
  - b가 그보다 크다면 그 위치에 1이 올 수 없음
- b를 구체화한 값들 만큼씩 P를 왼쪽으로 shift 한 결과들을 모두 뭉친 결과로 a의 비트 나열을 보완

정방향 결과	역방향 결과
TTTTTTTT	T11TTTTT
[2,4]	[2,3]
00TTTTTT	000111T0

```
a >> 2 = P  0111T0TT
a >> 3 = P  111T0TTT
```

진행 상황

- 한 일
    - 도메인과 상호 보완 함수 정의
    - 모든 연산에 대한 정방향 분석 정의
    - 비트 연산에 대한 역방향 분석 정의
- 할 일
    - 산술 및 논리 연산에 대한 역방향 분석 정의
    - 성능 측정과 보완 (충분히 많이 쳐내는가 / 충분히 효율적인가)