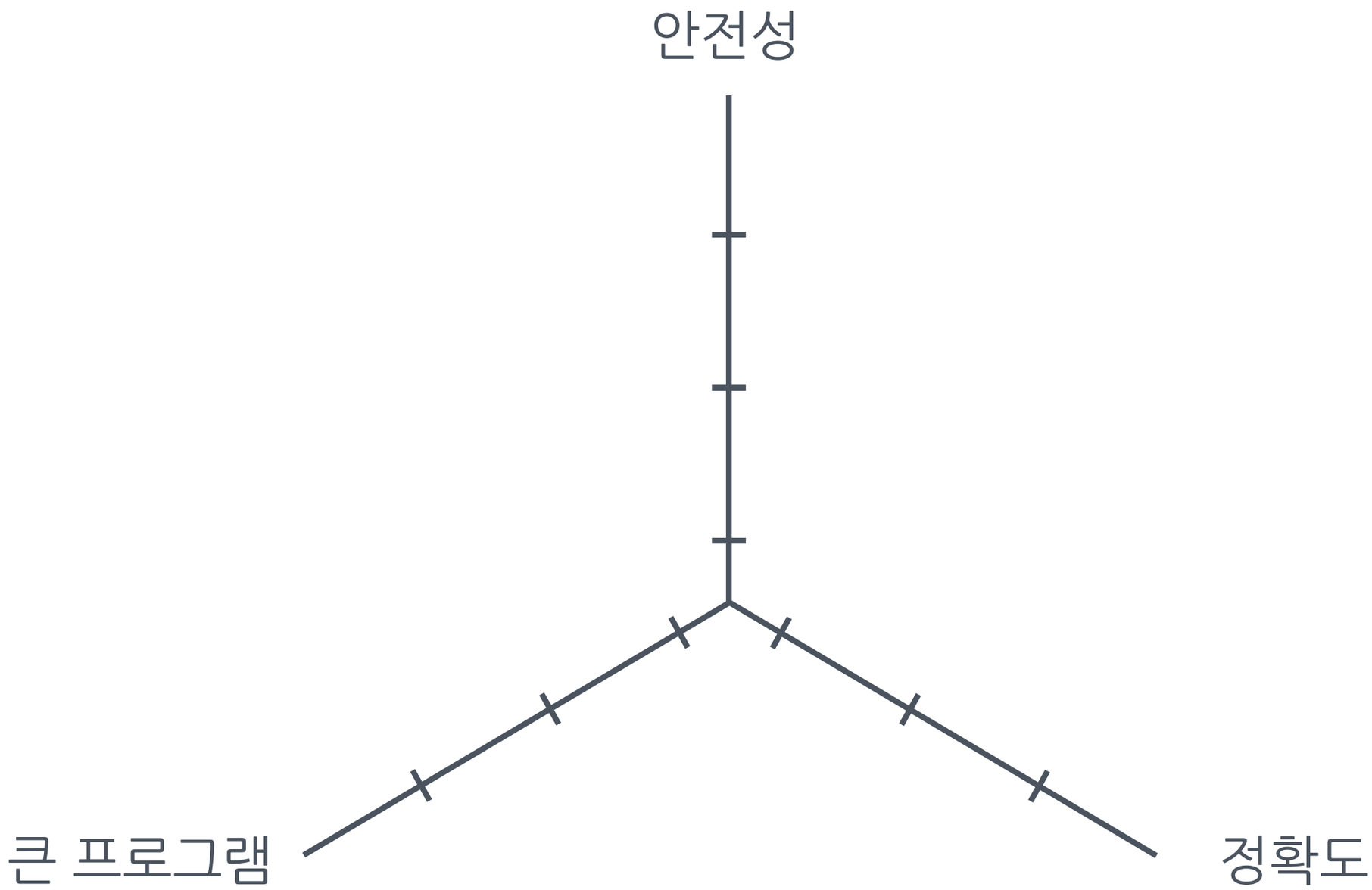
부분적으로 문맥을 구별하는 분석
(Partially Context-Sensitive Analysis)

오학주[1] 이원찬[1] 양홍석[2] 이광근[1]

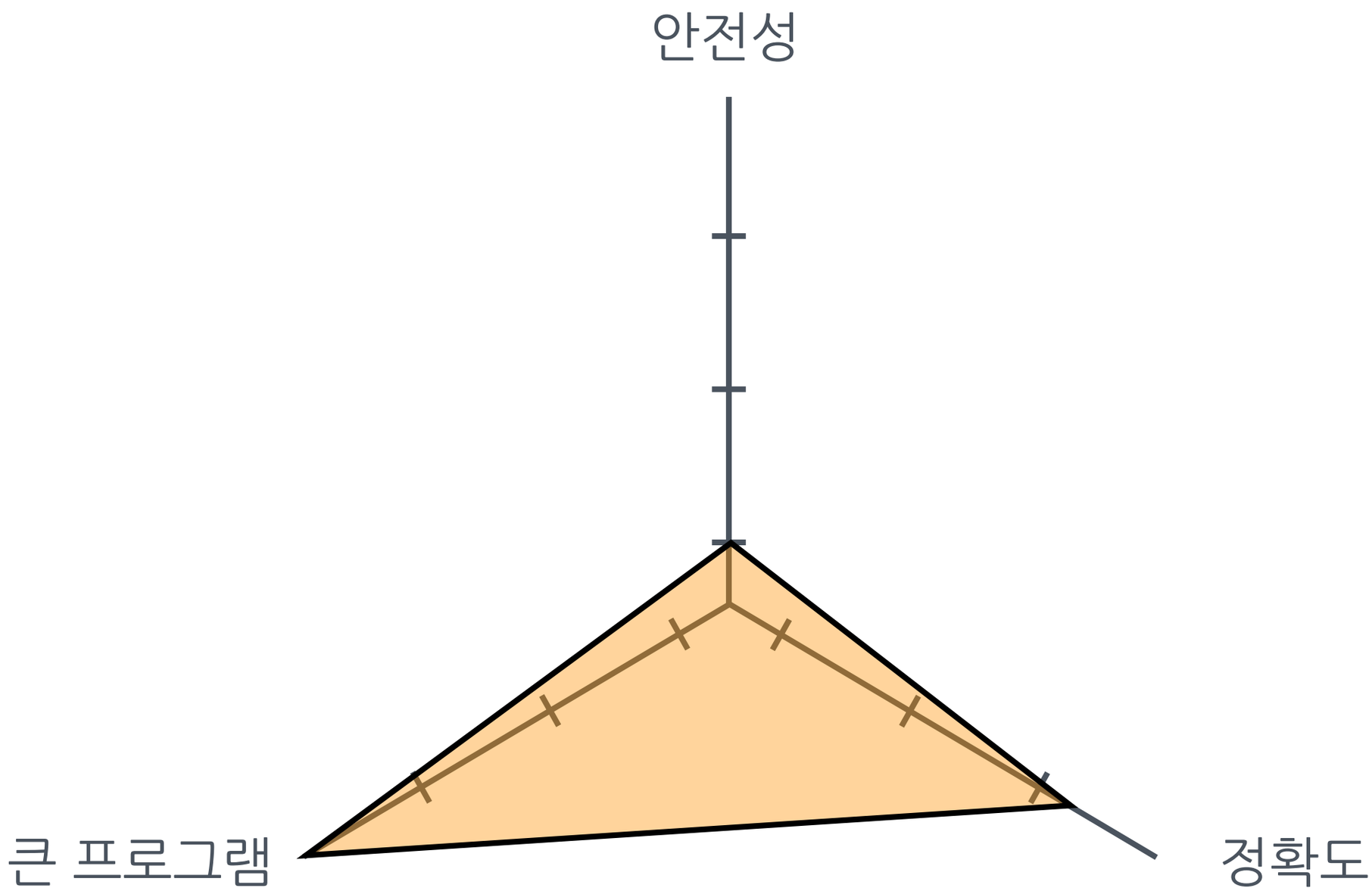
[1]서울대학교 [2]University of Oxford

SEOUL
NATIONAL
UNIVERSITY

정적분석의 난제

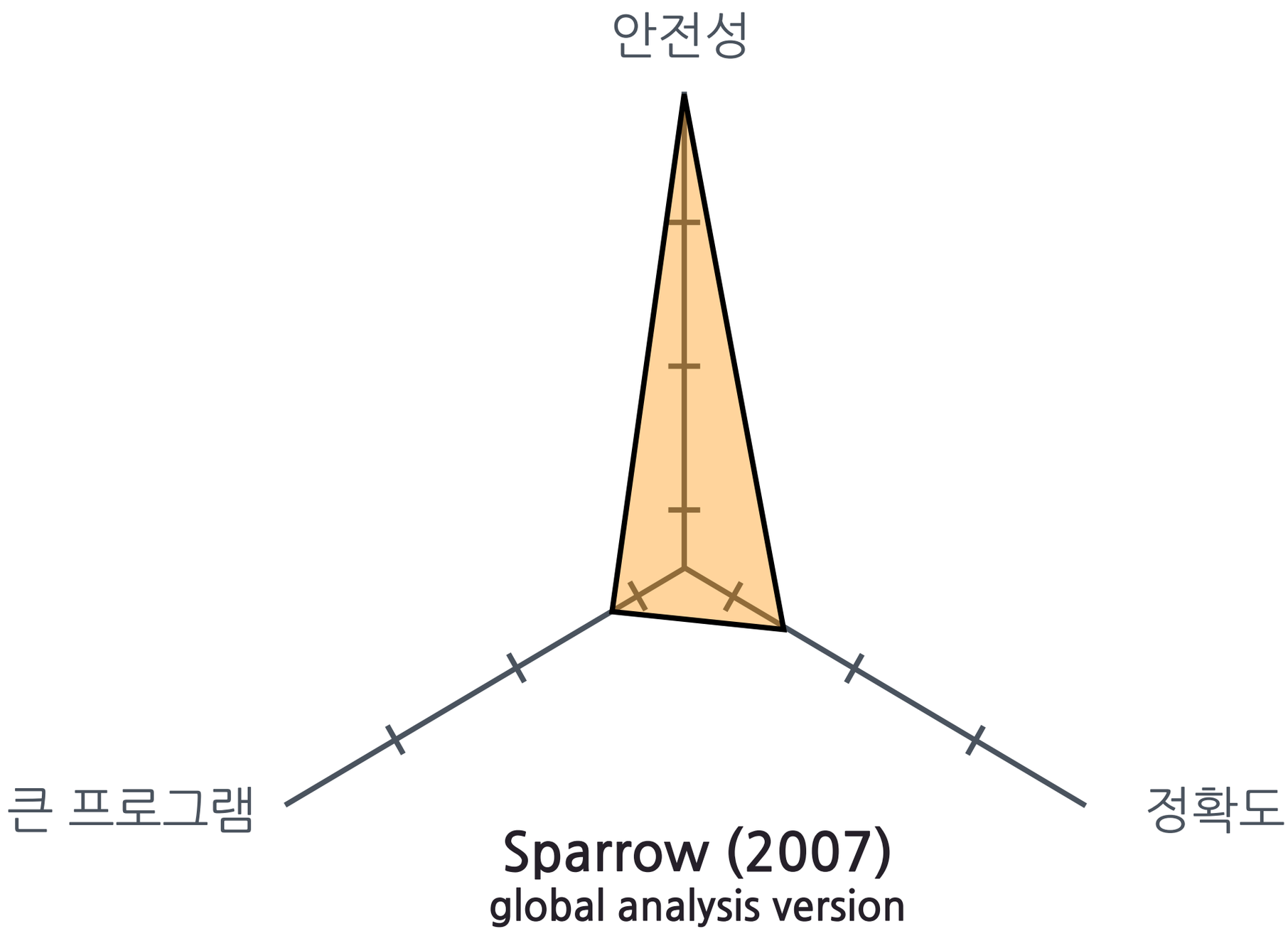


정적분석의 난제



버그 검출기

정적분석의 난제



정적분석의 난제

PLDI'12

Design and Implementation of Sparse Global Analyses for C-like Languages

Hakjoo Oh Kihong Heo Wonchan Lee Woosuk Lee Kwangkeun Yi
Seoul National University
{pronto,khheo,wclee,wslee,kwang}@ropas.snu.ac.kr

Abstract

In this article we present a general method for achieving global static analyzers that are precise, sound, yet also scalable. Our method generalizes the sparse analysis techniques on top of the abstract interpretation framework to support relational as well as non-relational semantics properties for C-like languages. We first use the abstract interpretation framework to have a global static analyzer whose scalability is unattended. Upon this underlying sound static analyzer, we add our generalized sparse analysis techniques to improve its scalability while preserving the precision of the underlying analysis. Our framework determines what to prove to guarantee that the resulting sparse version should preserve the precision of the underlying analyzer.

We formally present our framework; we present that existing sparse analyses are all restricted instances of our framework; we show more semantically elaborate design examples of sparse non-relational and relational static analyses; we present their implementation results that scale to analyze up to one million lines of C programs. We also show a set of implementation techniques that turn out to be critical to economically support the sparse analysis process.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis

Keywords Static analysis, abstract interpretation, sparse analysis

1. Introduction

Precise, sound, scalable yet global static analyzers have been unachievable in general. Other than almost syntactic properties, once the target property becomes slightly deep in semantics it's been a daunting challenge to achieve the four goals in a single static analyzer. This situation explains why, for example, in the static error detection tools for full C, there exists a clear dichotomy: either "bug-finders" that risk being unsound yet scalable or "verifiers" that risk being unscalable yet sound. No such tools are scalable to globally analyze million lines of C code while being sound and precise enough for practical use.

In this article we present a general method for achieving global static analyzers that are precise, sound, yet also scalable. Our approach generalizes the sparse analysis ideas on top of the abstract interpretation framework. Since the abstract interpretation framework [9, 11] guides us to design sound yet arbitrarily precise static analyzers for any target language, we first use the framework to have a global static analyzer whose scalability is unattended. Upon this underlying sound static analyzer, we add our generalized sparse analysis techniques to improve its scalability while preserving the precision of the underlying analysis. Our framework determines what to prove to guarantee that the resulting sparse version should preserve the precision of the underlying analyzer.

Our framework bridges the gap between the two existing technologies – abstract interpretation and sparse analysis – towards the design of sound, yet scalable global static analyzers. Note that while abstract interpretation framework provides a theoretical knob to control the analysis precision without violating its correctness, the framework does not provide a knob to control the resulting analyzer's scalability preserving its precision. On the other hand, existing sparse analysis techniques [6, 14, 15, 19, 20, 24, 40, 42, 44] achieve scalability, but they are mostly algorithmic and tightly coupled with particular analyses.[1] The sparse techniques are not general enough to be used for an arbitrarily complicated semantic analysis.

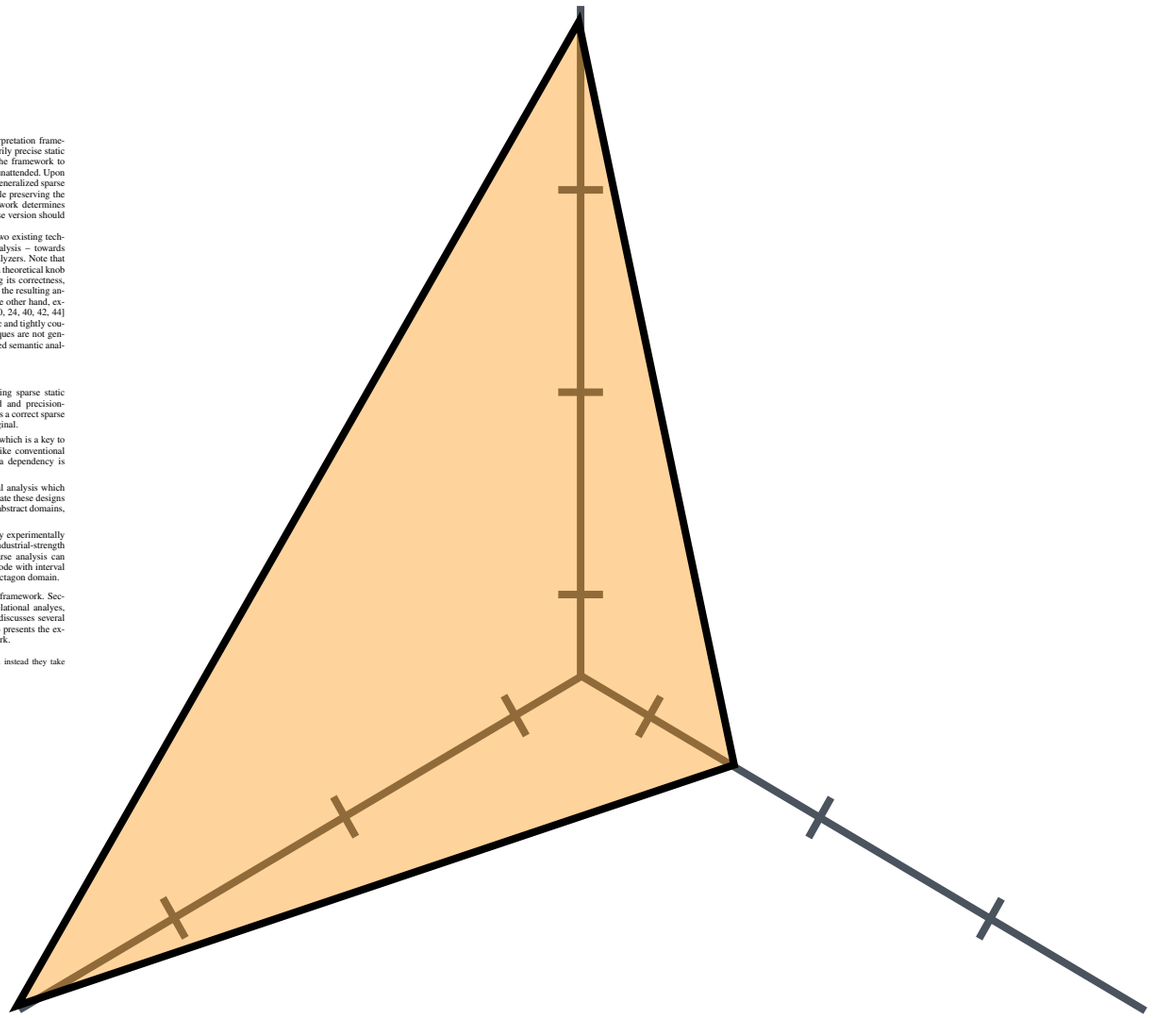
Contributions Our contributions are as follows.

- We propose a general framework for designing sparse static analysis. Our framework is semantics-based and precision-preserving. We prove that our framework yields a correct sparse analysis that has the same precision as the original.
- We present a new notion of data dependency, which is a key to the precision-preserving sparse analysis. Unlike conventional def-use chains, sparse analysis with our data dependency is fully precise.
- We design sparse non-relational and relational analysis which are still general as themselves. We can instantiate these designs with a particular non-relational and relational abstract domains, respectively.
- We prove the practicality of our framework by experimentally demonstrating the achieved speedup of an industrial-strength static analyzer [23, 26, 28, 35–38]. The sparse analysis can analyze programs up to 1 million lines of C code with interval domain and up to 100K lines of C code with octagon domain.

Outline Section 2 explains our sparse analysis framework. Section 3 and 4 design sparse non-relational and relational analyes, respectively, based on our framework. Section 5 discusses several issues involved in the implementations. Section 6 presents the experimental studies. Section 7 discusses related work.

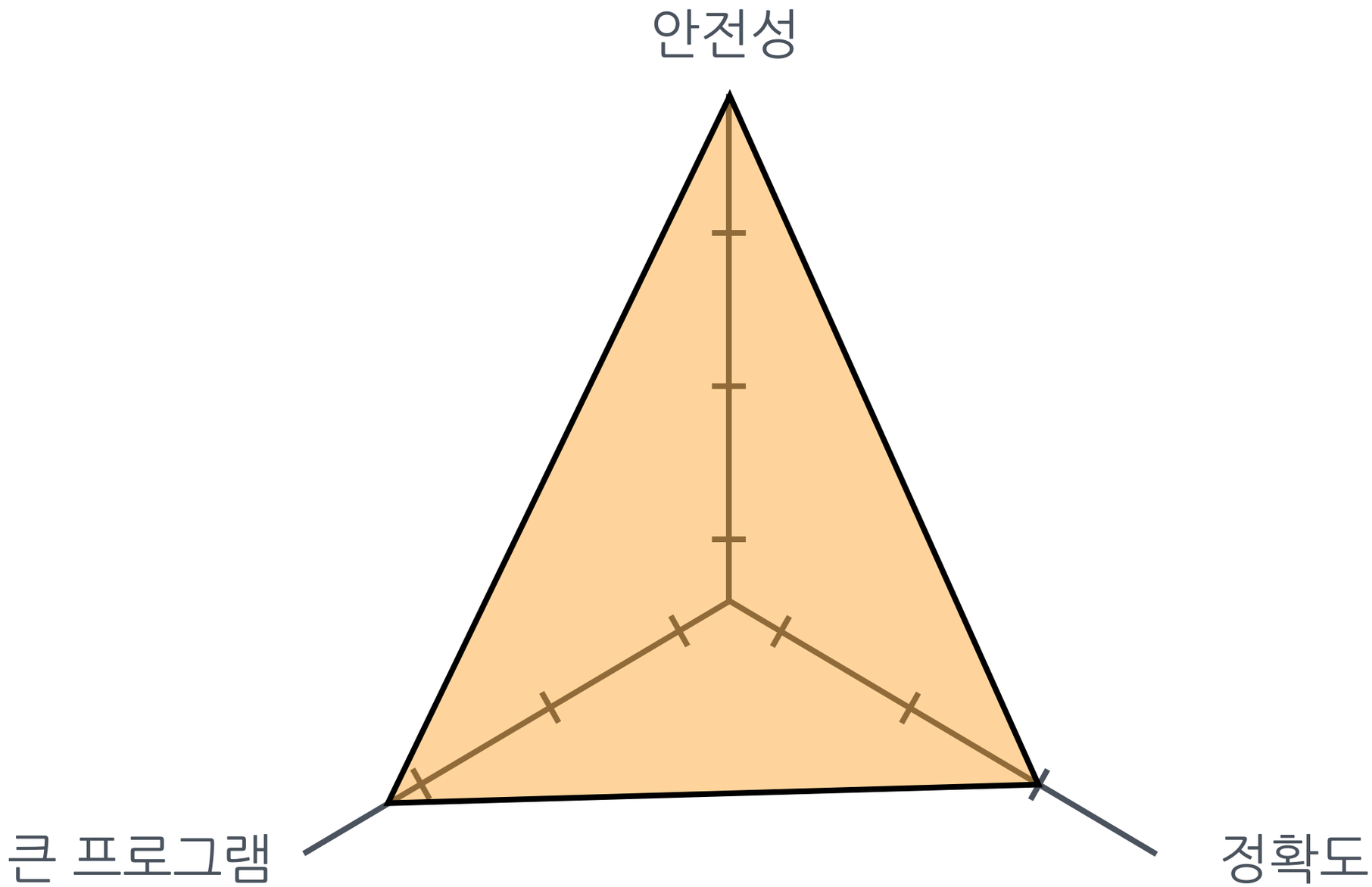
[1] A few techniques [7, 39] are in general settings but instead they take coarse-grained approach to sparsity.

PLDI'12, June 11–16, 2012, Beijing, China.



Sparse Sparrow (2012)

목표: 정확도 향상



허위경보 주요 원인들

- 함수 호출 문맥 뭉치기
- 변수 관계 잊기
- 배열 및 버퍼 내용 뭉치기
- 모르는 라이브러리 함수
- ...

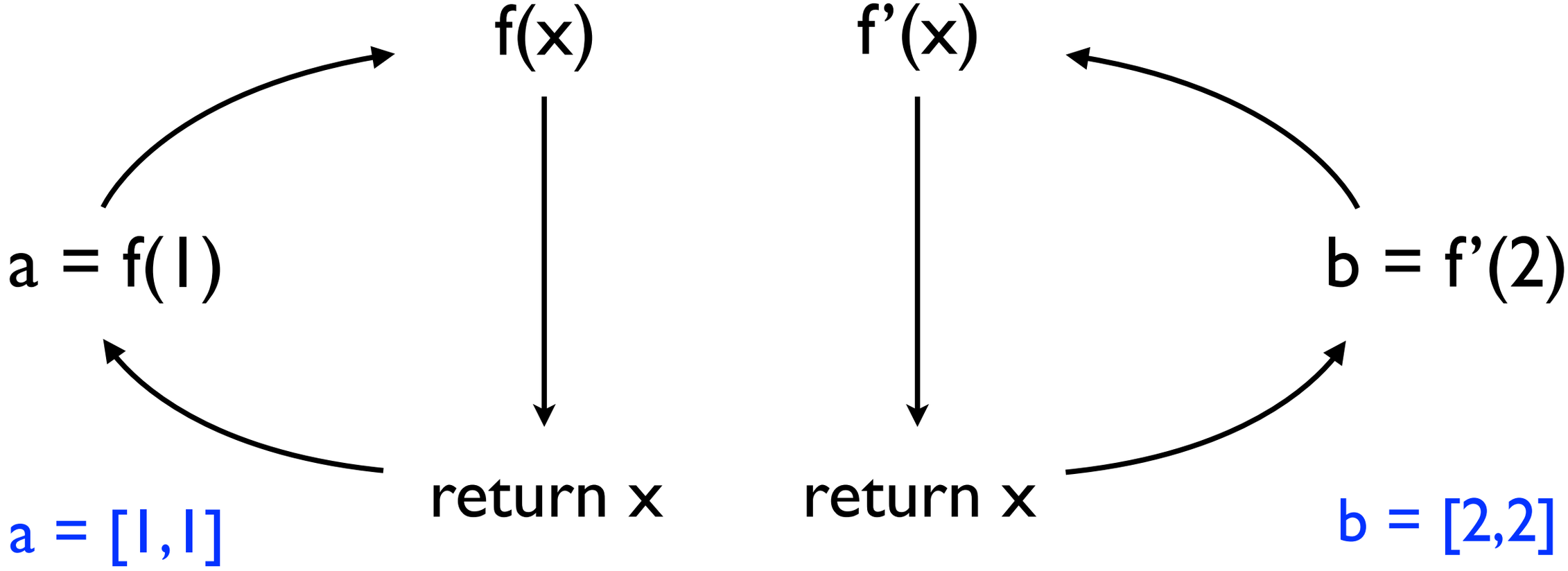
첫단계: 문맥 구분

- Sparse Sparrow 의 경우,
  - 문맥을 뭉쳐서 생기는 허위경보가 가장 많음
  - make와 tar의 경우 허위경보의 50%이상

문맥을 구분하는 분석
(context-sensitive analysis)

```
f (int x) { return x; }

a = f(1);
b = f(2);
```

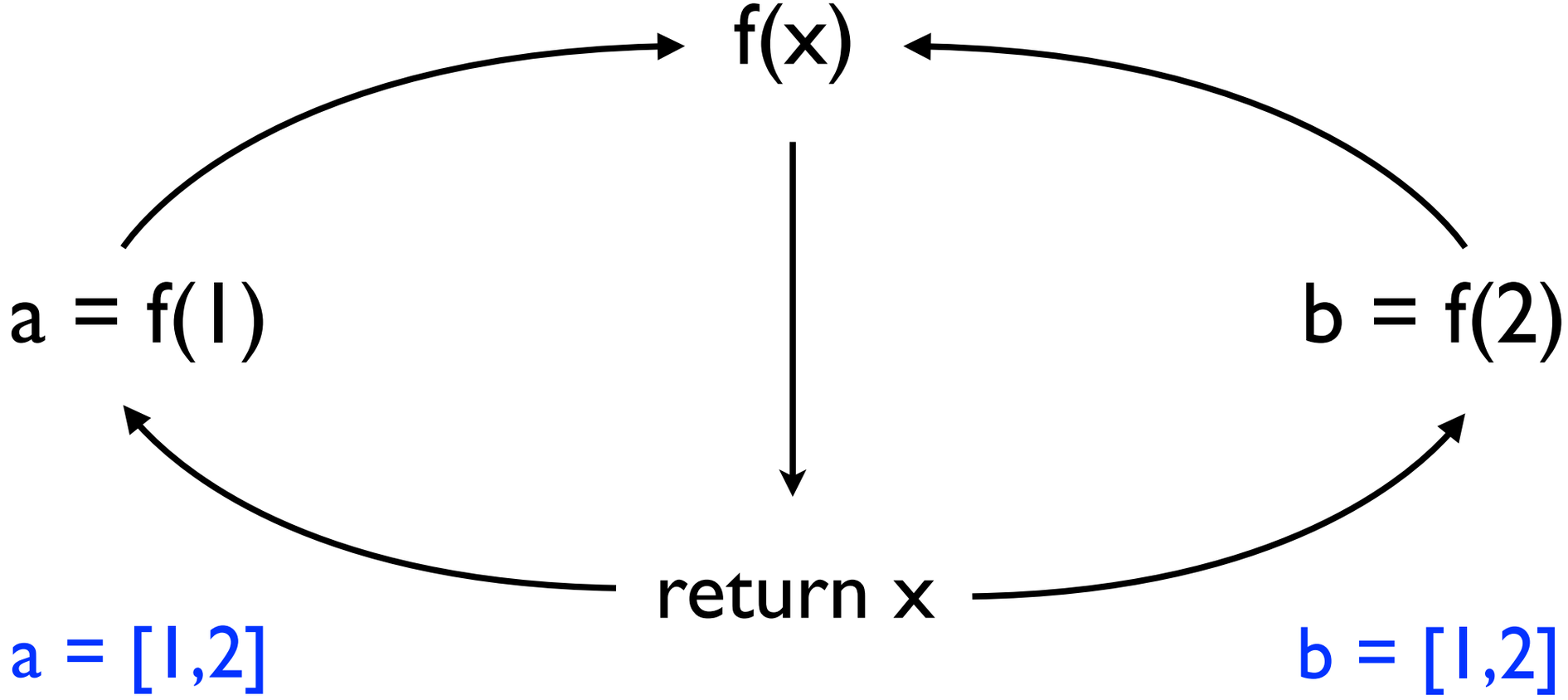


현재 Sparrow

문맥을 뭉치는 분석
(context-insensitive analysis)

```
f (int x) { return x; }

a = f(1);
b = f(2);
```



모든 문맥 구분은 불가능

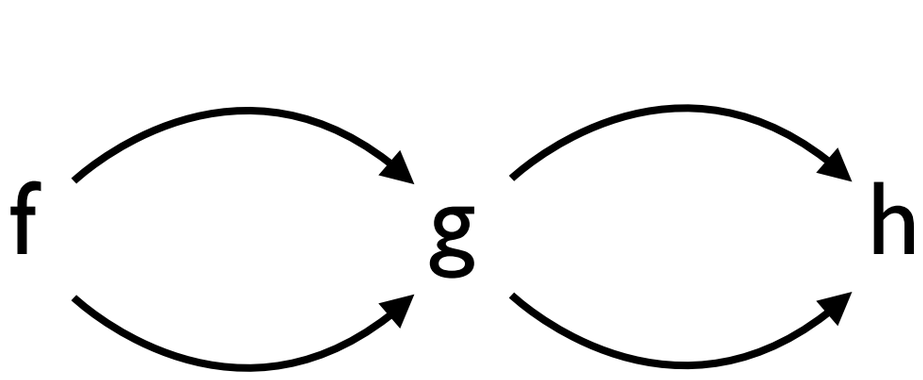
parts of less-382

문맥 요약은 필수,
어떻게 요약하는가가 이슈

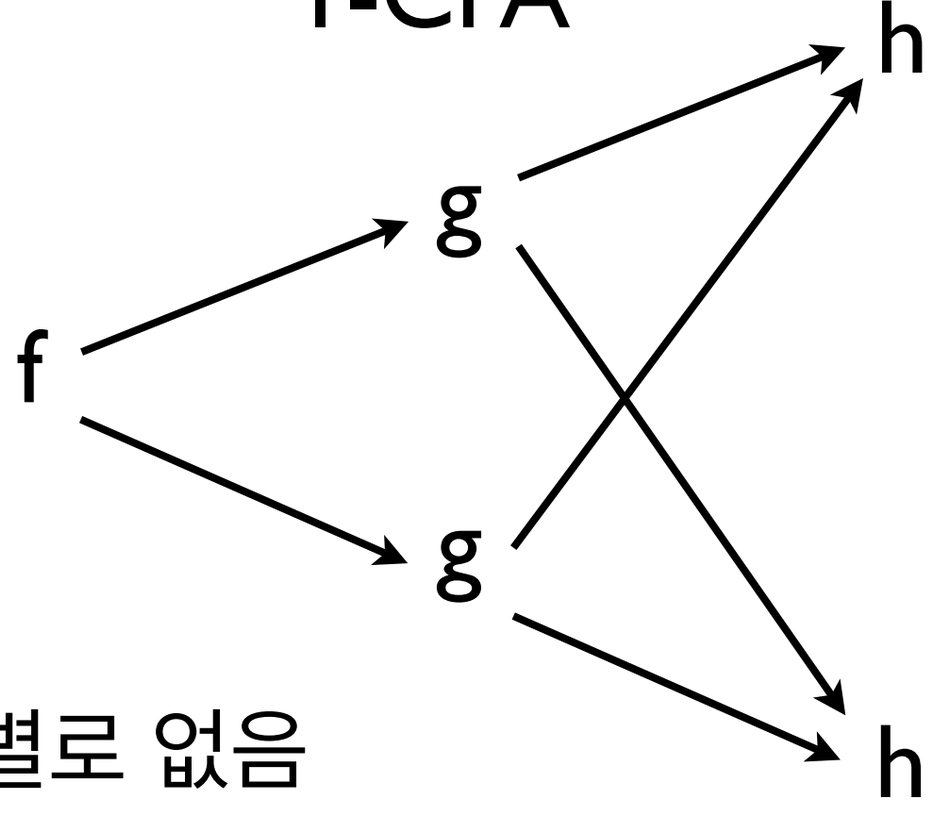
기존 문맥 요약의 문제: k-CFA

- 각 함수의 문맥을 k단계까지 일괄적으로 구분

0-CFA



1-CFA



- k가 작으면: 정확도 향상이 별로 없음
- k가 크면: 비용이 감당 안됨

목표: “좋은” 문맥 요약 찾기

구분된 문맥에서 항상 정확도 향상이 있음

= 정확도 향상이 없을 문맥은 구별하지 않음

예 1

```
char * xmalloc (int size)
{
  char *result = (char *) malloc (size);
  if (result == 0)
    fatal ("virtual memory exhausted");
  return result;
}

int main()
{
  path = xmalloc(10);        // (1) : O
  *path = ...;               // alarm

  ptr = xmalloc(unknown);    // (2) : X
  *ptr = 2;                  // alarm
}
```

예 2

```
void multi_glob (int size) {
  while (...) {
    while (...) {
      char *elt = xmalloc(size); // (1) : O
      *elt = ...;   // alarm
    }
  }
}

int main()
{
  multi_glob (8);   // (2) : O
  multi_glob (16);  // (3) : O
  multi_glob (4);   // (4) : O
}
```

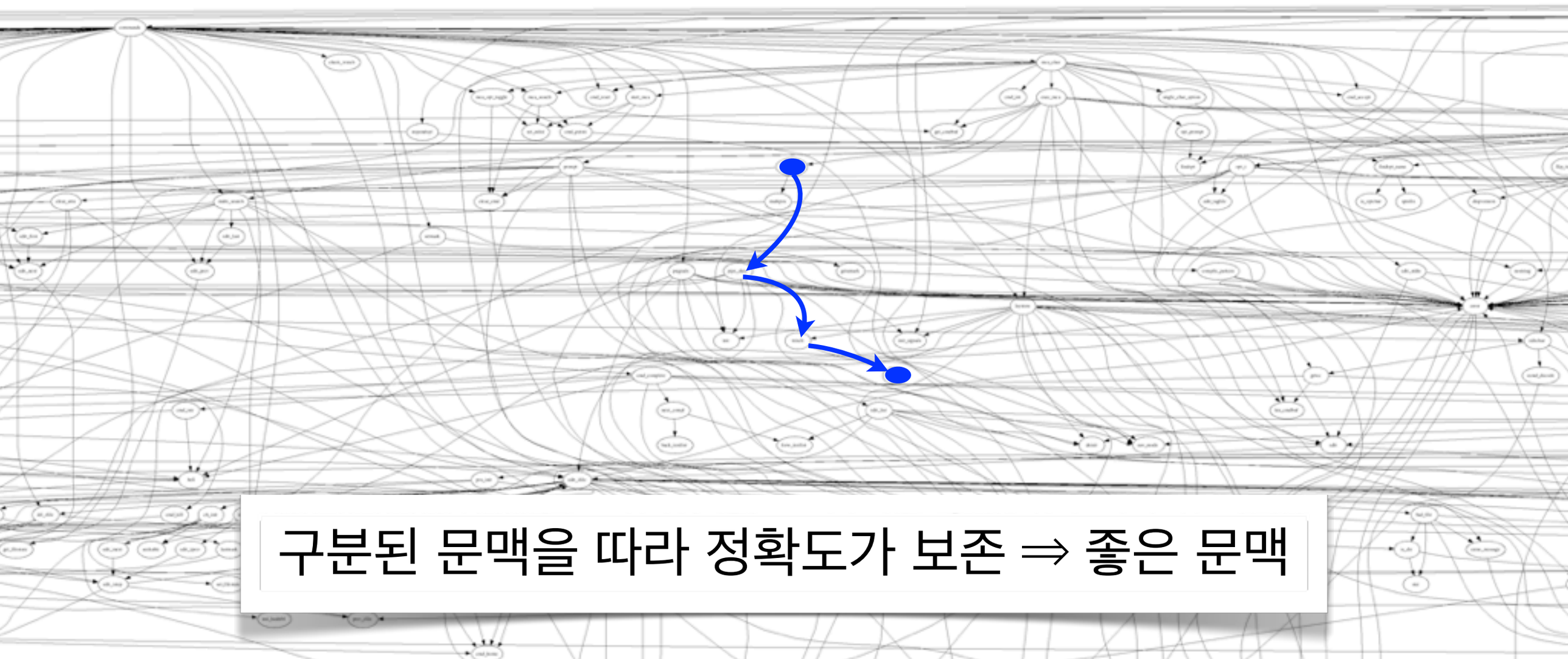
예 3

```
int atoi(const char *str)
{
    int i = 0;
    while(*str) {
        i = (i<<3) + (i<<1) + (*str - '0');
        str++;
    }
    return i;
}

int main()
{
    char *p = "10";
    int n = atoi (p);          // (1) : X
    char *buf = xmalloc (n); // (2) : X
    *buf = ...;
}
```

아이디어

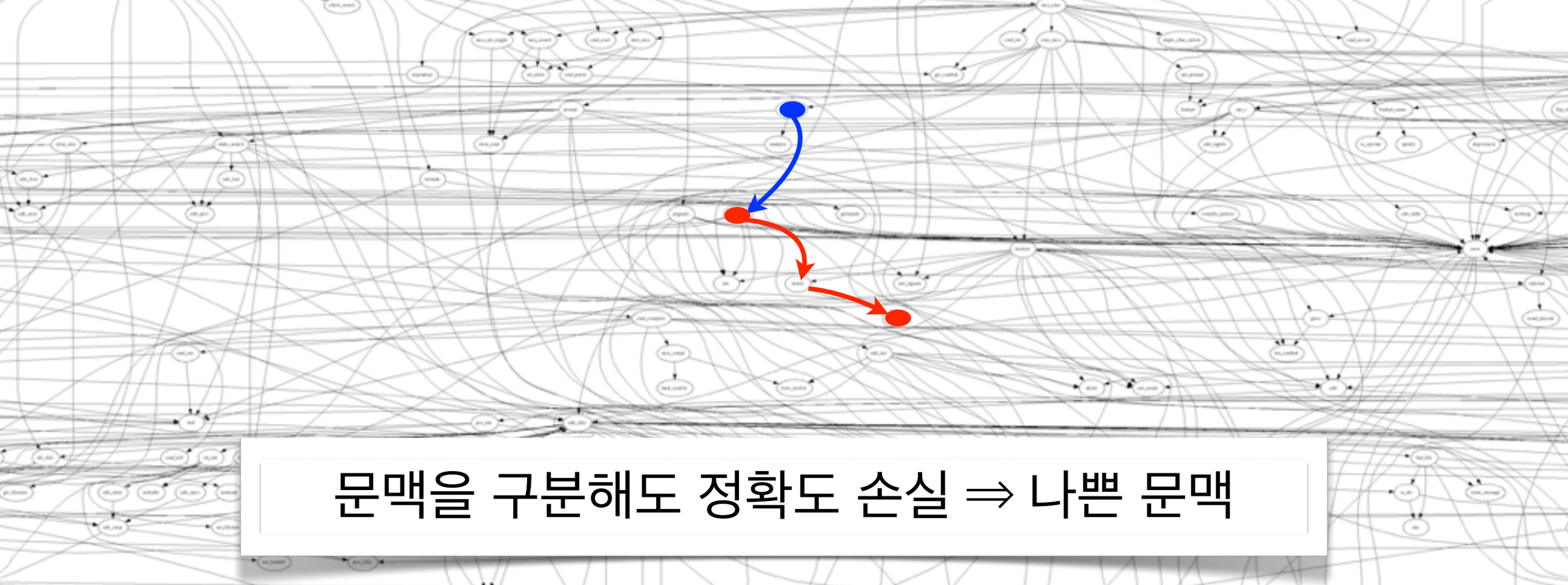
- 문맥을 모두 구분하는 분석을 실제로 해보면 “좋은” 문맥과 “나쁜” 문맥을 구분할 수 있다.



구분된 문맥을 따라 정확도가 보존 ⇒ 좋은 문맥

아이디어

- 문맥을 모두 구분하는 분석을 실제로 해보면 “좋은” 문맥과 “나쁜” 문맥을 구분할 수 있다.



문맥을 구분해도 정확도 손실 ⇒ 나쁜 문맥

아이디어

$A_{\perp}$ 문맥을 모두 뭉치는 분석

ㄴㅐ

$A_{\top}$ 문맥을 모두 구분하는 분석

아이디어

$A_\perp$

ㄴㅐ

$A_\top \longrightarrow \pi$ 좋은 문맥 요약

아이디어

$A_\perp$

네

좋은 문맥만 구별하는 분석

$A_\top \longrightarrow \pi$

A_π

아이디어

$A_\perp$

ㄴㅐ

$A_\top \longrightarrow \pi$

1단계: 전분석

A_π

2단계: 본분석

아이디어

$A_\perp$

$\sqcup\!\!\sqcup$

$A_\top \longrightarrow \pi$ A_π

$\sqcup\!\!\sqcap$

$\hat{A}_\top \longrightarrow \hat{\pi}$ $A_{\hat{\pi}}$

$\hat{\pi}$ 도 "좋은" 요약

$$A_\top \longrightarrow \pi$$

$$\sqcap \qquad\qquad \sqcup$$

$$\hat{A}_\top \longrightarrow \hat{\pi}$$

= 구분된 문맥에서는 항상 정확도 향상이 있음

≠ 정확도 향상이 있는 문맥은 모두 구분함

실험결과

프로그램	문맥에 둔감한 분석		부분적으로 문맥에 민감한 분석				Δ경보	Δ시간	Δ크기
	경보	분석시간	경보	전분석	본분석	구분된 호출수			
spell	58	0.6	30	0.3	0.9	25/124 (20.2%)	48.00%	1.8x	1.47x
bc	606	15.6	483	6.5	15.3	29/777 (3.7%)	20.0%	1.4x	1.03x
tar	940	43.8	799	11.8	43.5	56/1,218 (4.6%)	15.00%	1.3x	1.05x
less	654	131.1	561	11.9	184.7	59/1,522 (3.9%)	14.00%	1.5x	1.22x
make	1,500	89.3	1,002	20.3	124.2	85/1,050 (8.3%)	33.00%	1.6x	1.20x
wget	1,307	72	905	29.9	126.1	111/1,973 (5.6%)	30.0%	2.2x	1.74x
a2ps	3,682	125	2,004	205.3	343.6	263/2,450 (10.7%)	46.00%	4.4x	1.93x

결론

- 문맥 요약을 잘하면 적은 추가비용으로 높은 정확도 향상을 이룰 수 있다.
- 모든 문맥을 고려하는 전처리 분석으로 좋은 문맥 요약을 찾을 수 있다.