PL과 LLM의 상호보완적 결합을 통한 프로그램 합성 및 분석

한양대학교 ERICA 프로그래밍시스템 연구실

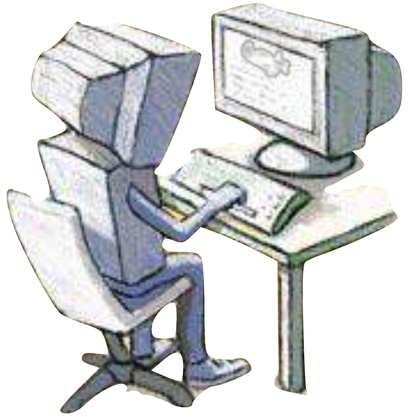
2025. 08. 22
SIGPL 2025 여름학교

한양대 ERICA 프로그래밍 시스템 연구실

- 지도교수: 이우석
- 박사과정
  - 이제형
  - 조한결
  - 김진상
- 석사과정
  - 왕오
  - 주강대
  - 이정훈
  - 박준성
- 방문연구
  - Alexis Just (CY-TECH, France)

우리 연구실의 주된 관심

프로그램 합성과 프로그램 분석

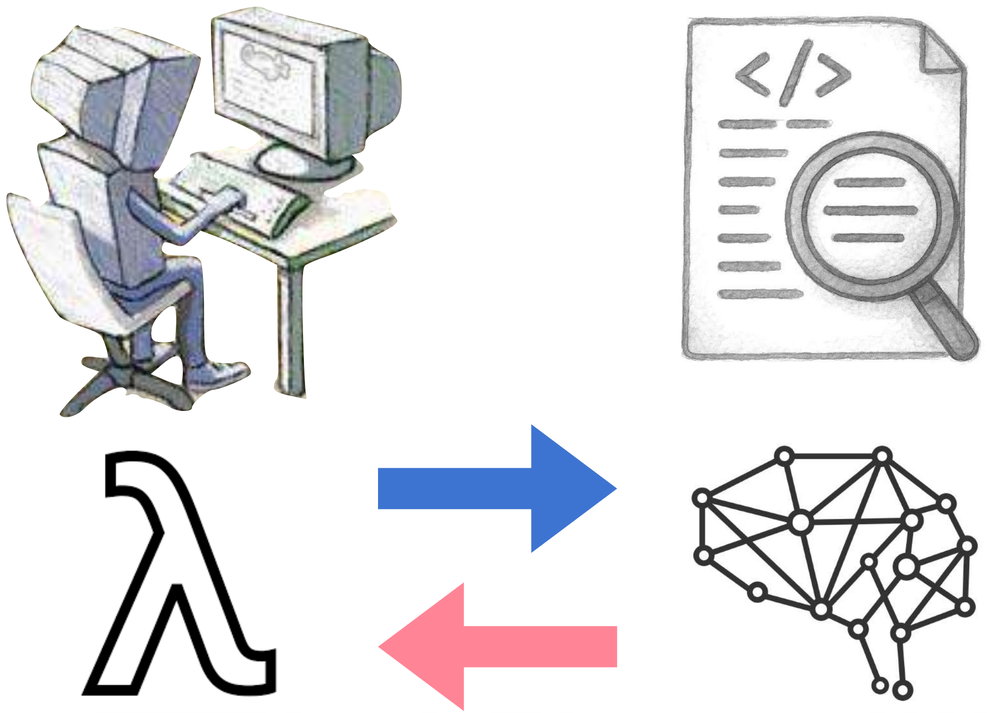


- **프로그램 합성 가속화 기술**
  - 함수형 재귀호출 프로그램 합성 (POPL'23, JFP'25)
  - 프로그램분석 기반 합성 가속화 (PLDI'23)
  - 통계모델 기반 합성 가속화 (PLDI'18)
  - 양방향 탐색 전략 (POPL'21)

- **프로그램 분석 관련**
  - 프로그램 간소화 (CCS'18)
  - Datalog 분석 합성 (FSE'18)
  - 인스턴트앱 자동생성 (ASEJ'23)
  - 코드보안을 보장하는 분석 (IEEE'22)

- **프로그램 합성 응용**
  - 동형암호 프로그램 최적화 (PLDI'20, TOPLAS'23)
  - 프로그램 합성 기반 패치생성 (FSE'21)
  - 프로그램 역난독화 (CCS'23)

우리 연구실의 최근 관심



프로그래밍언어이론과 신경망의 상호보완적 결합

프로그램 합성

- 사용자가 **원하는** (생김새 + 행동) 프로그램을 **자동** 생성하는 기술

- LLM과 차이점?
  - 신경망추론이 아닌 다양한 (PL이론기반) 탐색 알고리즘 수행
  - (끝난다면) 요구조건 100% 만족 보장

프로그램 합성의 쓸모

- **스펙을 100% 만족 + 기존에 없던 새로운 코드 발견 가능**
- **프로그램 최적화**
  - 꼬마최적화(peephole optimization) 발견
  - 함수형언어에서 중간계산과정 없애기(deforestation)
  - 데이터가 올때마다 처리하기(stream processing)
- **프로그램 역난독화**
  - 표현식 간단히(Mixed Boolean-Arithmetic deobfuscation)
  - 분기문 간단히(control-flow deobfuscation)
- **프로그램 검증**
  - 반복문 불변식(loop invariant) 추론
  - 자료구조 성질(representation invariant) 추론
- **보안**
  - 부채널(side-channel)공격을 회피하는 회로 합성
  - 샌드박스에서 시스템콜 허용정책 자동생성
  - 동형암호회로 가속화

프로그램 합성의 한계

- **스펙을 100% 만족 + 기존에 없던 새로운 코드 발견 가능**
- **프로그램 최적화**
  - 꼬마최적화(peephole optimization) 발견
  - 함수형언어에서 중간계산과정 없애기(deforestation)
  - 데이터가 올때마다 처리하기(
- **프로그램 역난독화**
  - 표현식 간단히(Mixed Boole
  - 분기문 간단히(control-flow deobfuscation)
- **프로그램 검증**
  - 반복문 불변식(loop invariant) 추론
  - 자료구조 성질(representation invariant) 추론
- **보안**
  - 부채널(side-channel)공격을 회피하는 회로 합성

템콜 허용정책 자동생성

화

큰 프로그램 생성은 어려움

LLM 기반 코드 생성

- 큰 코드도 잘 작성하나 **틀릴 수 있음** (아래 예: Meta LLaMA 3)

```
#include <stdio.h>
#include <string>
#include <map>
using namespace std;
string sort_numbers(string numbers){
  map<string, int> m;
  string s;
  for(i = 0; i<numbers.length(); i++)
   if (numbers[i] == ‘’){
        m[s] = stoi(s);
        s = “”;
…
```

구문오류 (‘’ -> ‘\0’)

```
let rec maxSum (arr: int list)
  (i: int) (k: int)
  (sum: int) (max: int) : int =
  match arr with
  | [] -> max
  | h::t -> if i = k then
      maxSum t (i+1) (k+1) (sum+h) (max)
    else
      let newSum = sum + h in
      let newMax = max (newSum) max in
      maxSum t (i+1) (k+1) newSum newMax
```

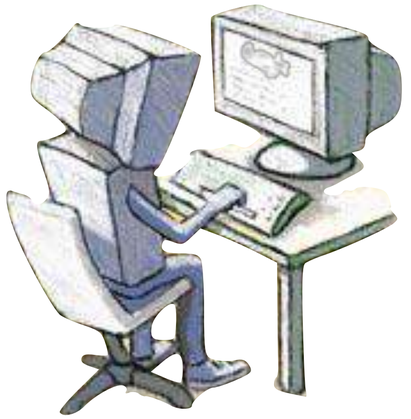
타입오류: 함수여야하는데 정수

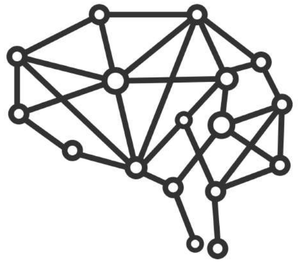
프로그램 합성의 두 방향

	탐색기반(Program synthesis)	LLM기반(Neural synthesis)
대두 시기	1960년대	2010년대
입력	입출력 예제, 논리식	자연어 기술, 입출력 예제
제약 조건 만족여부	100% 만족 보장	만족 못시킬 수 있음
생성 가능한 코드 크기	작음	큼
코드생성 방법	탐색 알고리즘 수행	신경망 추론
대표적 예	Flash(Fill, Extract, …), Bluepencil, …	GitHub Copilot, Cursor AI, …

프로그램 자동생성을 위한 LLM + 합성 결합

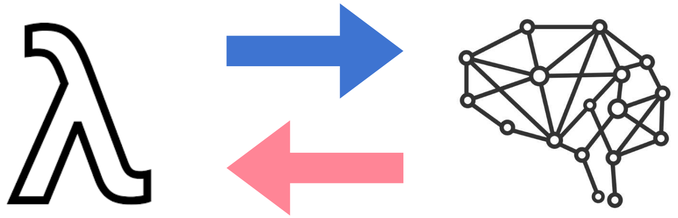
	제한된 코드생성 Constrained Decoding
조건만족?	기본적인 성질들 만족 (문법, 타입, …)
코드크기	큼





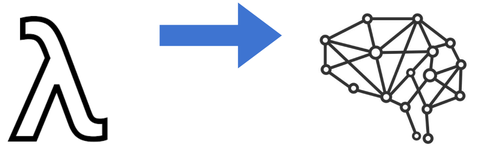
	신경망안내기반 합성 Neural-Guided Program Synthesis
조건만족?	100% 만족 보장
코드크기	보통

우리 연구실에서의 LLM + 합성 결합



- **LLM 생성 코드의 문법/타입/변수사용 올바름 보장**
  - LLM의 토큰 생성 과정을 감시하며 잘못된 생성 방지
  - 문법/**변수사용 올바름**: 임의의 언어에 대해
  - 타입 올바름: 강한 정적 타입 언어(예: OCaml)
- **LLM 이 안내하는 프로그램 합성**
  - LLM도 틀린 결과를 내고 탐색기반 합성은 성능문제로 못푸는 문제들에 대해서
  - 시너지: LLM의 솔루션을 힌트로 쓰는 프로그램 합성. 올바른 프로그램 빠르게 생성

LLM의 코드생성 과정

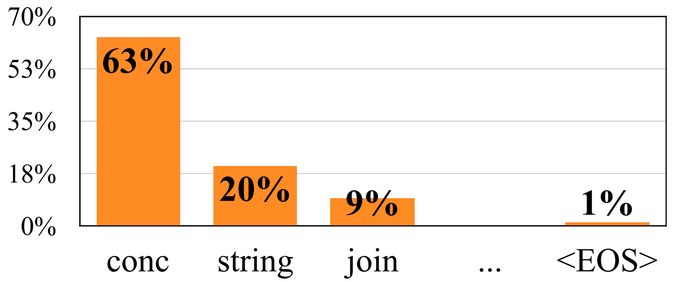


프롬프트

```
```Give python function that
concat two string```
def
```

LLM

토큰별 확률



Tokenizer

Tokens

	``	Give	…	def
ID	2	2331	…	4299

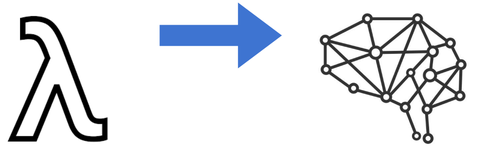
토큰 선택

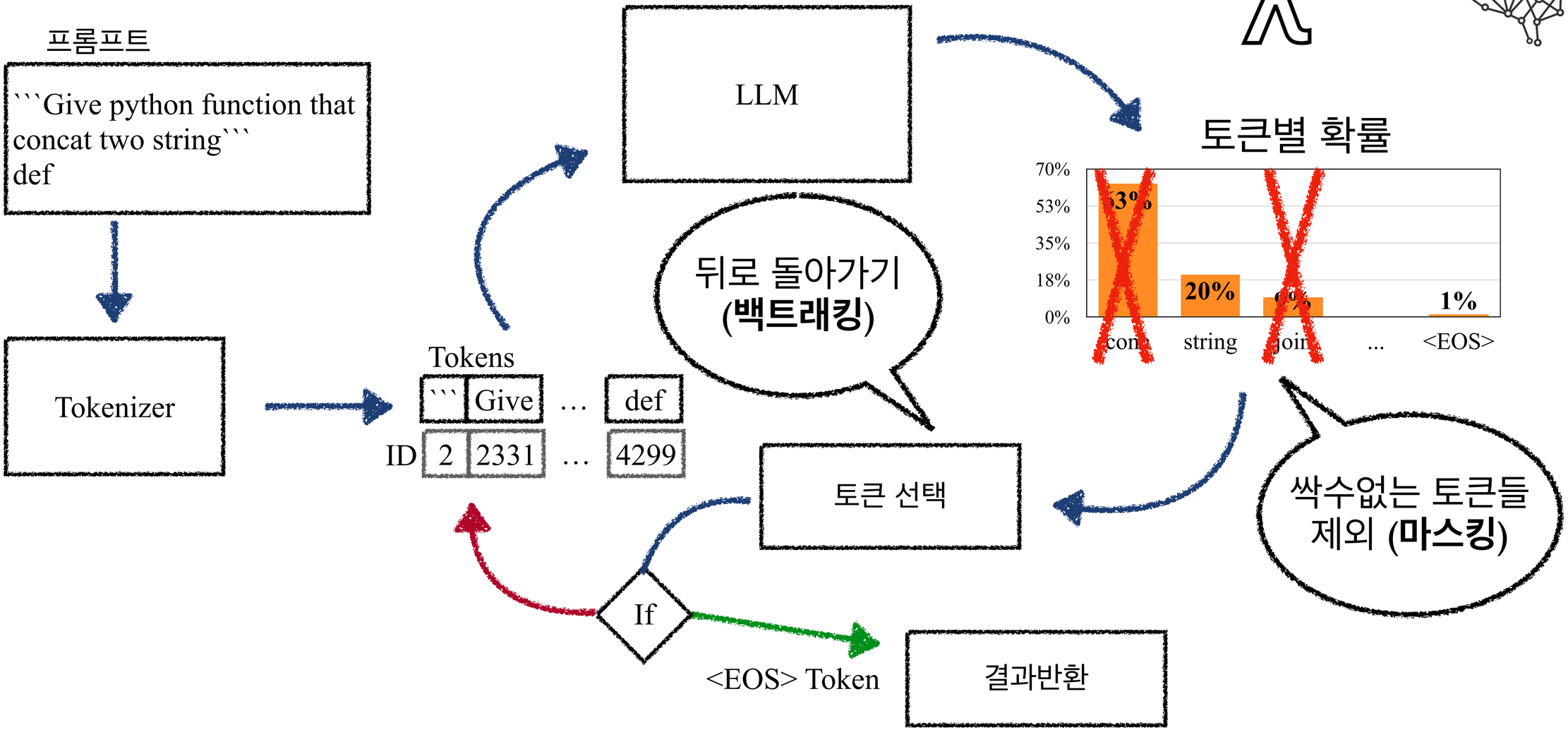
If

<EOS> Token

결과반환
```

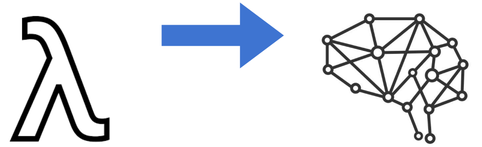
제한된 코드생성 (Constrained Decoding)





LLM 생성 코드의 올바름 보장 예

문법오류 수정



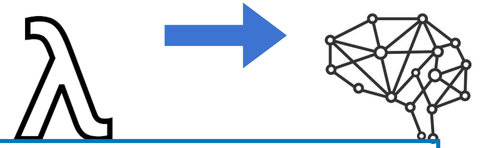
```
#include <stdio.h>
#include <string>
#include <map>
using namespace std;
string sort_numbers(string numbers){
  map<string, int> m;
  string s;
  for(i = 0; i<numbers.length(); i++)
   if (numbers[i] == ‘’){
        m[s] = stoi(s);
        s = “”;
…
```

구문오류 (‘’ -> ‘\0’)

```
#include <stdio.h>
#include <string>
#include <map>
using namespace std;
string sort_numbers(string numbers){
  map<string, int> m;
  string s;
  for(i = 0; i<numbers.length(); i++)
   if (numbers[i] == ‘0’){ temp=“zero”; }
   else if (numbers[i] == ‘1’){ temp=“one”; }
…
```

LLM 생성 코드의 올바름 보장 예

타입오류 수정



```
(* Write a function to …
let rec maxSum (arr: int list) (i: int) (k: int)
(sum: int) (max: int) : int =
  match arr with
  | [] -> max
  | h::t -> if i = k then
      maxSum t (i+1) (k+1) (sum+h) (max)
    else
      let newSum = sum + h in
      let newMax = max (newSum) max in
      maxSum t (i+1) (k+1) newSum newMax
```

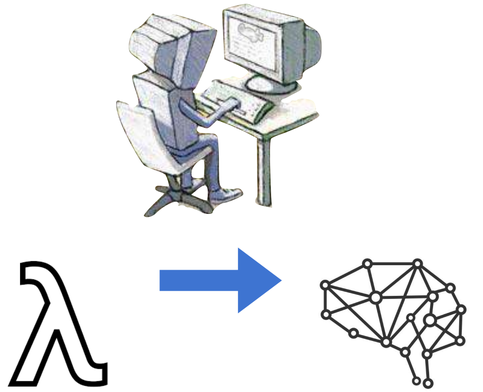
Function 자리에 int가 사용 :TYPE ERROR

```
(* Write a function to …
let rec maxSum (arr: int list) (i: int) (k: int)
(sum: int) (max: int) : int =
  match arr with
  | [] -> max
  | h::t ->  if i = k then
      maxSum t (i+1) (k+1) (sum+h) (max)
    else
      let newSum = sum + h in
      let newMax = maxSum t (i+1) (k+1) newSum (max) in
      let newMax = maxSum t (i+1) (k) newSum (max) in
      newMax
```

int list -> int -> int -> int -> int 올바른 타입의 함수

LLM 생성 코드의 올바름 보장 예

정의되지 않은 변수 사용 오류 수정



```
from typing import List, Tuple

def sum_product(numbers: List[int]) -> Tuple[int, int]:
    """ For a given list of integers,(10, 24)    """
    if not numbers:
        return (0,1)
    else:
        return (sum(numbers), reduce(lambda x, y: x*y, numbers))
```

```
from typing import List, Tuple

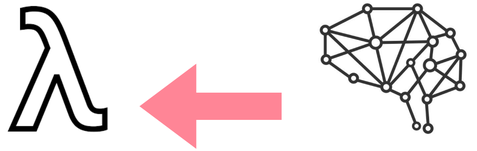
def sum_product(numbers: List[int]) -> Tuple[int, int]:
    """ For a given list of integers,(10, 24)    """
    if not numbers:
        return (0,1)
    else:
        return (sum(numbers), numbers[0] * sum_product(numbers[1:])[1])
```

정의되지 않은 reduce 함수명 사용

(∵ from functools import reduce 없이 사용)

라이브러리 함수 대신 **재귀호출로 대체**

LLM이 안내하는 프로그램 합성



```
let rec f x y =
    match x with
    | [] ->
        match y with
        | [] -> 0
        | n2 :: rest2 -> n2
    | n1 :: rest1 ->
        match y with
        | [] -> n1
        | n2 :: rest2 ->
            (compare n1 n2) + (f rest1 rest2)
```

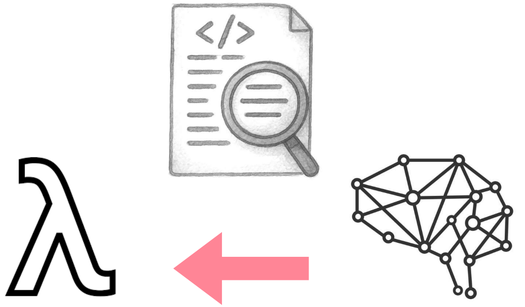
LLM이 생성한
오답코드를 참고하여

```
let rec f x y =
    match x with
    | [] ->
        match y with
        | [] -> 0
        | n2 :: rest2 -> n2 + (f [] rest2)
    | n1 :: rest1 ->
        match y with
        | [] -> n1 + (f rest1 [])
        | n2 :: rest2 ->
            match (compare n1 n2) with
            | EQ -> n1 + (f rest1 rest2)
            | GT -> n1 + (f rest1 rest2)
            | LT -> n2 + (f rest1 rest2)
```

정답코드
빠르게 합성

정적 분석 마의 삼각형

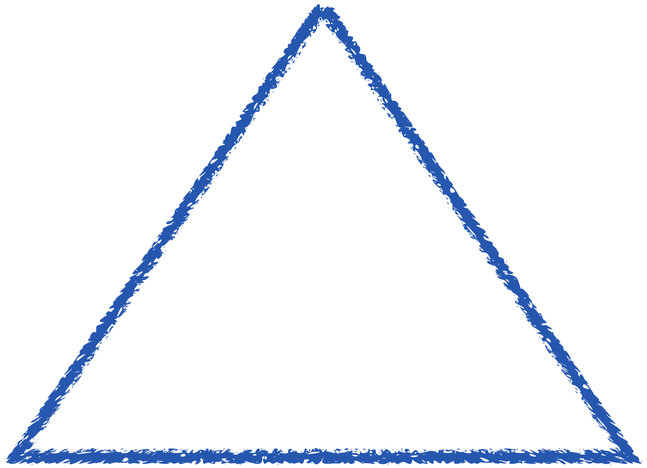
단 둘만 가능



안전성 (실제 가능성 모두 포섭)

정확도

(드문 허위정보)

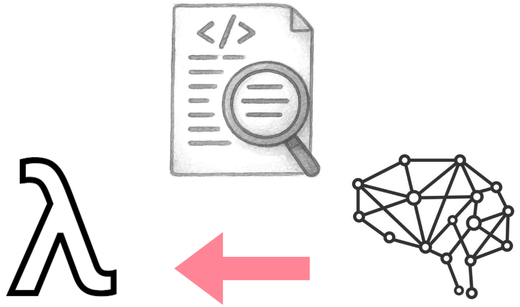


분석속도

(대규모 SW분석 가능)

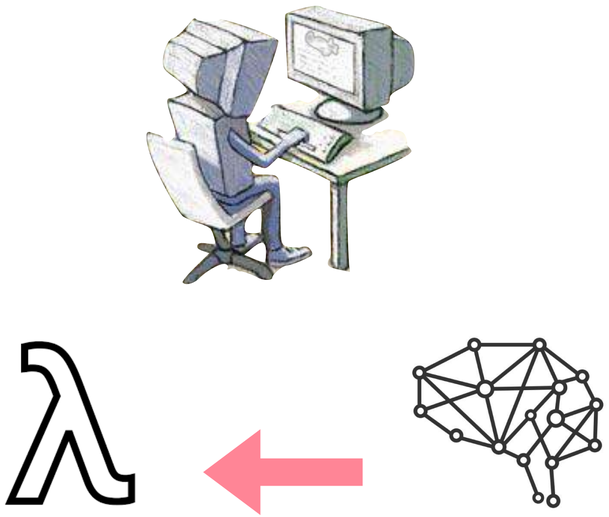
- 한계극복 위해 사람들이 LLM을 정적분석에 사용. But 단순 사용은 별로 효과적이지 않음
  - 단순사용: 정적 분석 경보와 코드 보여주고 “해당 경보의 진위여부를 알려줘.”
  - **토큰 제한**: 최대입력토큰수 제한. 코드는 더 거대
  - **환각**: 믿을 수 없고 일관적이지 않은 답변 — 긴 답변의 진위여부를 체크해야.

LLM 상호작용을 통한 허위경보 제거



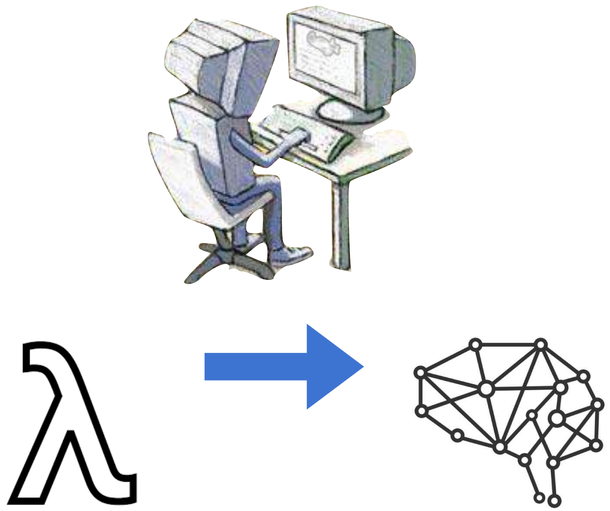
- 먼저 정적 분석의 결론 도출 과정(provenance)을 획득
- 결론 도출 과정 중, 틀렸을 가능성이 있는 일부 단계들에 대해서 LLM에 질의
- 전체 오류 경보의 진위여부를 묻는 것 보다
  - 질문 및 대답 양 적음 — 검증을 위한 노력 감소, **토큰 제한 문제 해결**
  - 질문의 난이도 감소 — **환각으로 인한 오답 확률 감소**

뒤 이어



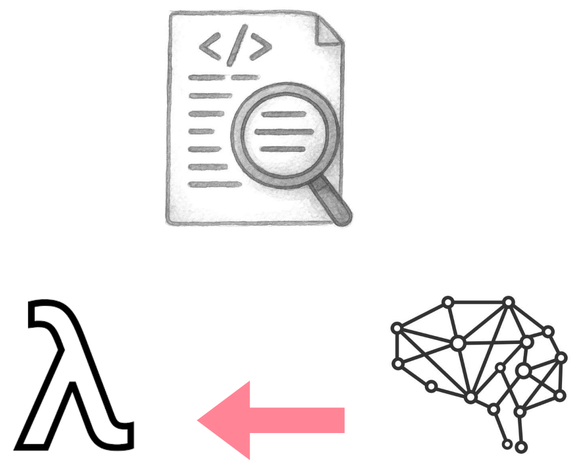
LLM 오답을 이용한
양방향탐색 안내

(조한결 박사과정)



LLM이 생성하는 코드에서
정의되지 않은 변수사용
금지하기

(박준성, 김진상 석사과정)

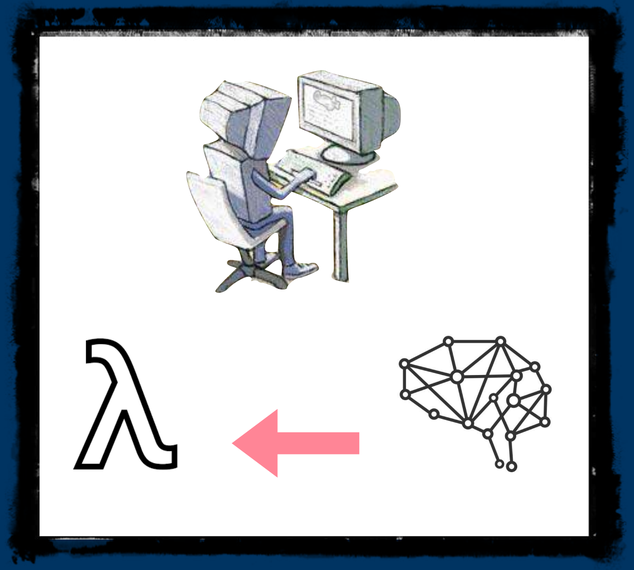


LLM과 상호작용을 통한
정적분석 허위경보 제거

(주강대 석사과정)

LLM 오답을 이용한 양방향탐색 안내

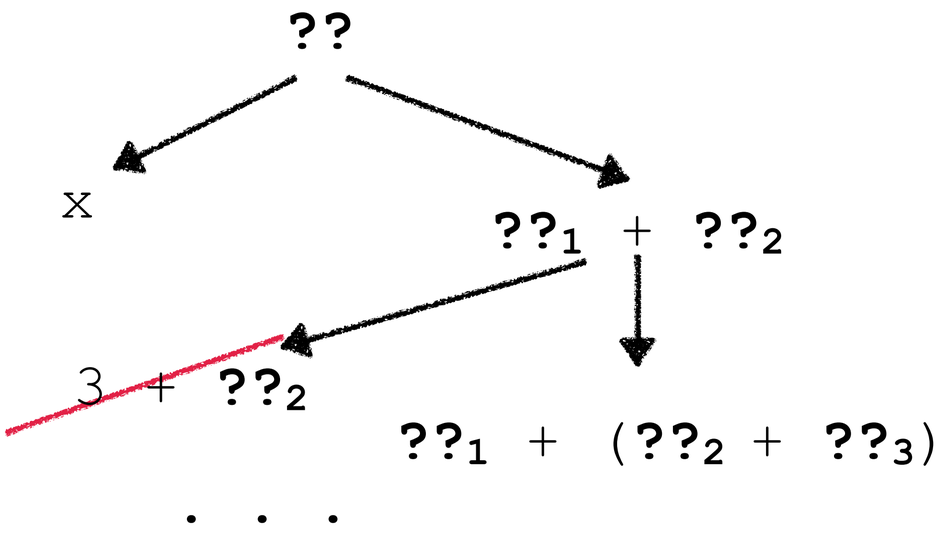
조한결 박사과정



합성을 위한 두 탐색 전략

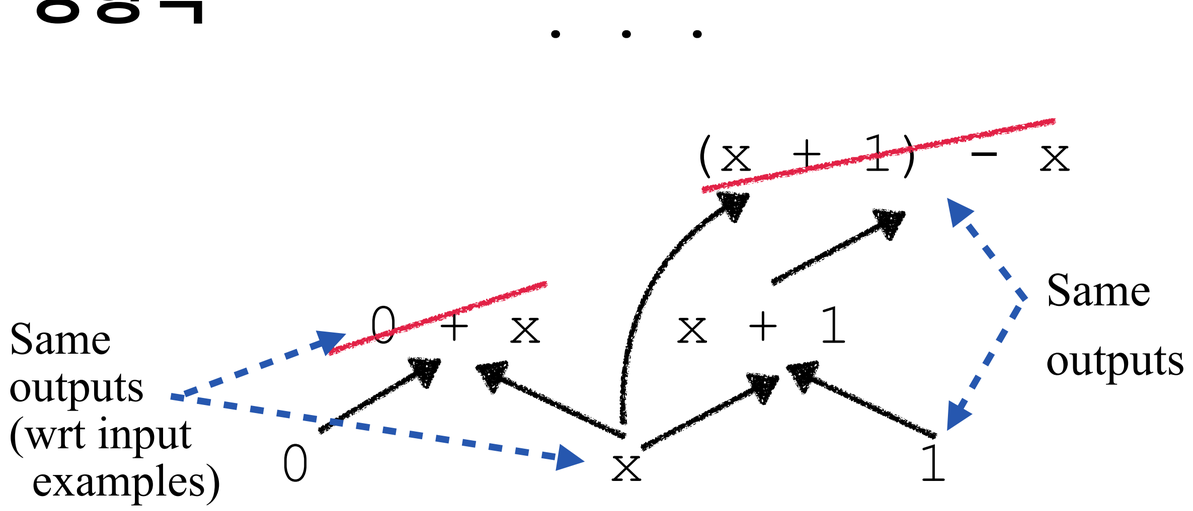
`function f x = ??` (spec: `0 ⟼ 0, 1 ⟼ 2`)

하향식



- 빈 프로그램에서 시작, 구멍을 점차 채움
- 싹수없는*infeasible* 프로그램 조기 제거

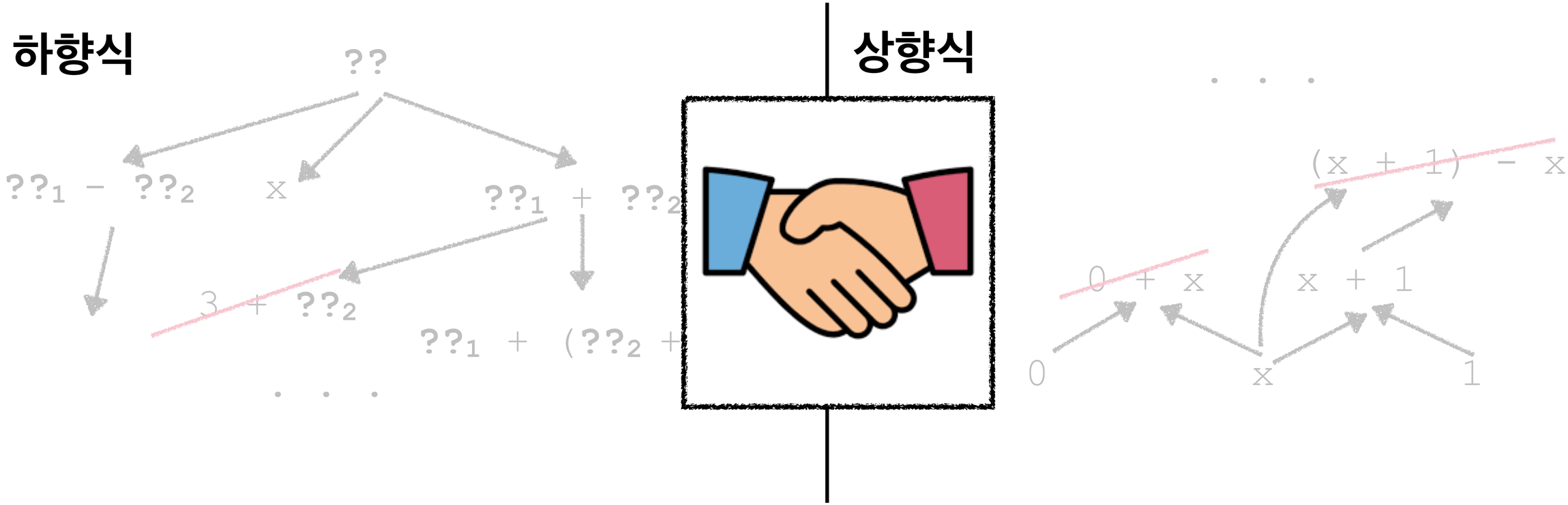
상향식



- 작은식들을 조합, 점차 큰 프로그램 생성
- 실행결과가 같은 중복*redundant* 식들 제거 (*observational equivalence pruning*)

양방향 탐색 전략

- 하향식으로 뼈대 생성 + 상향식으로 구멍을 채울 식 생성†
  - 싹수없는 뼈대 제거 + 중복되는 식 제거
- 다양한 합성기가 사용중 (Duet[POPL'21], Simba[PLDI'23], Flashfill++[POPL'23], Synthphonia[PLDI'25])



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

양방향 합성 알고리즘

양방향탐색 (문법 G, 스펙 Φ)

1 : C := ∅, $n := 1$

2 : repeat

3 : C := **부품식추가**(G, n)

4 : Q := { □ }

5 : while Q ≠ ∅ do

6 : P := **후보선택**(Q)

7 : if P 만족 Φ then return P

8 : if P 싹수없음 then continue

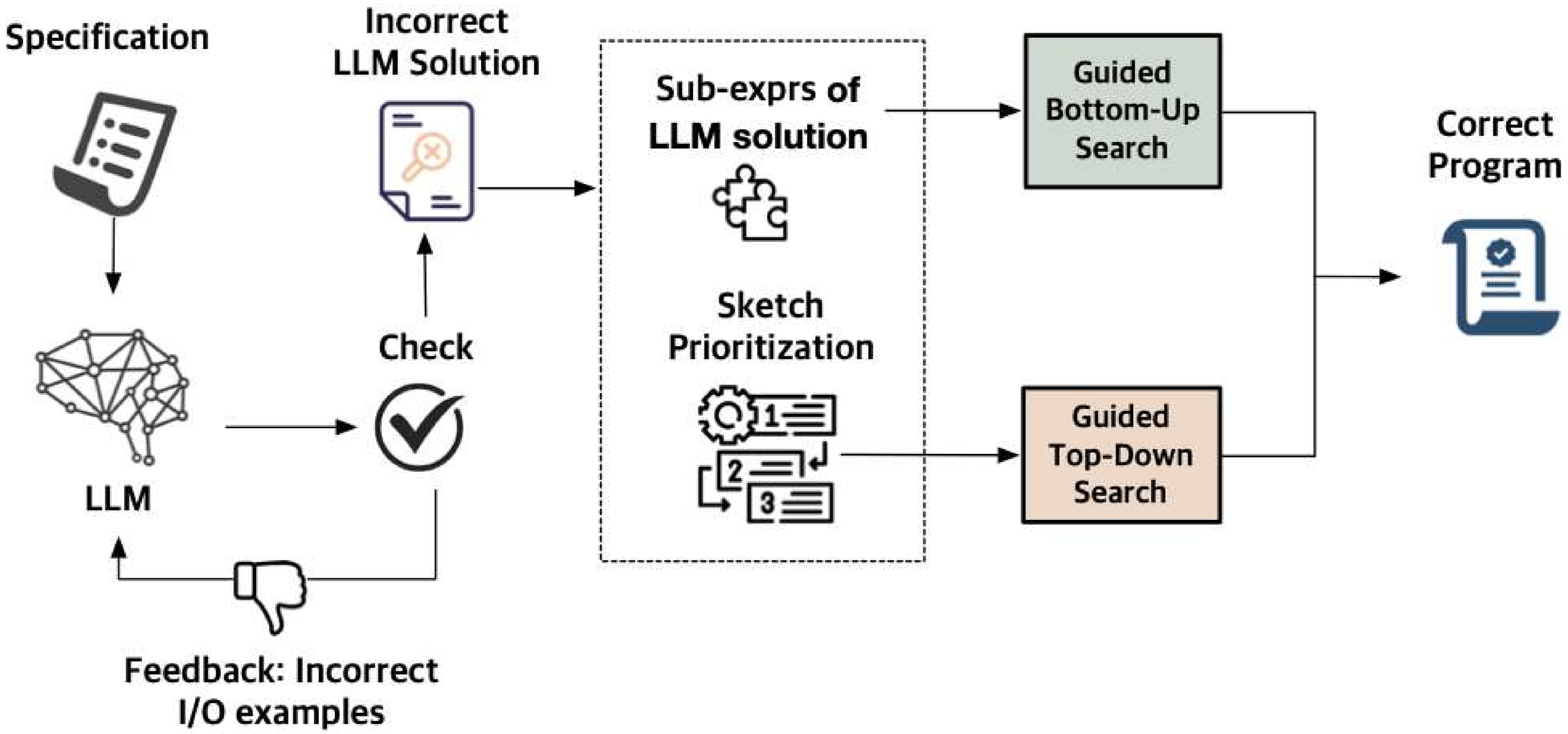
9 : h := **구멍선택**(P)

10: Q := Q ∪ **구멍채우기**$_P$(h, G, C)

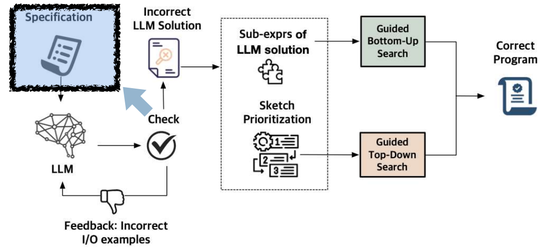
11 : done; $n := n + 1$

12: until 시간제한 초과

LLM 오답을 이용한 양방향탐색 개요



예제 문제



- 두 정수 리스트를 받아서, 각 인덱스 마다 원소들을 비교하여 더 큰 것들의 합을 반환하는 함수 만들기. 리스트 길이가 다를 경우, 더 긴 리스트의 나머지 부분을 결과에 더해서 반환
- 입출력 예제:

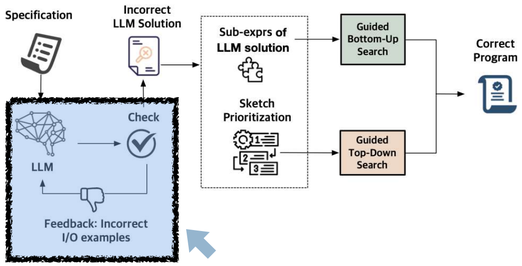
```
[ ]          [ 0 ]          ->   0
[1, 2, 3]    [0, 1]         ->   6
[1, 2]       [0, 4, 2, 3]   ->   10
 …
```

- SOTA 합성기들 모두 2분내 합성 실패

정답

```
let rec f x y =
    match x with
    | [] ->
        match y with
        | [] -> 0
        | n2 :: rest2 -> n2 + (f [] rest2)
    | n1 :: rest1 ->
        match y with
        | [] -> n1 + (f rest1 [])
        | n2 :: rest2 ->
            match (compare n1 n2) with
            | EQ -> n1 + (f rest1 rest2)
            | GT -> n1 + (f rest1 rest2)
            | LT -> n2 + (f rest1 rest2)
```

LLM에 질의



- GPT-4o-mini 사용. 틀릴 경우 피드백 제공. 3회까지 기회주기
- 그럼에도 오답 생성

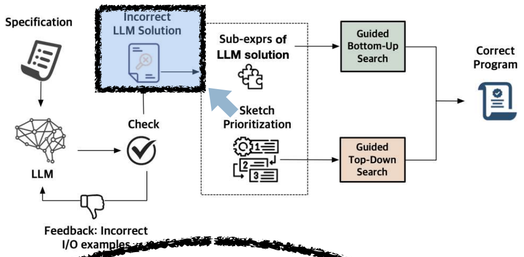
```
1    let rec f x y =
2        match x with
3        | [] ->
4            match y with
5            | [] -> 0
6            | n2 :: rest2 -> n2
7        | n1 :: rest1 ->
8            match y with
9            | [] -> n1
10           | n2 :: rest2 ->
11               (compare n1 n2) + (f rest1 rest2)
12
13
14
```

LLM오답

```
let rec f x y =
    match x with
    | [] ->
        match y with
        | [] -> 0
        | n2 :: rest2 -> n2 + (f [] rest2)
    | n1 :: rest1 ->
        match y with
        | [] -> n1 + (f rest1 [])
        | n2 :: rest2 ->
            match (compare n1 n2) with
            | EQ -> n1 + (f rest1 rest2)
            | GT -> n1 + (f rest1 rest2)
            | LT -> n2 + (f rest1 rest2)
```

정답

틀렸지만 유용



- 전체적인 구조 유사
- 정답의 중요부분이 그대로 존재하거나, 약간 다르게 존재

```
1    let rec f x y =
2        match x with
3        | [] ->
4            match y with
5            | [] -> 0
6            | n2 :: rest2 -> n2
7        | n1 :: rest1 ->
8            match y with
9            | [] -> n1
10           | n2 :: rest2 ->
11               (compare n1 n2) + (f rest1 rest2)
12
13
14
```

match 구조 유사

정답에 존재하는 식

```
let rec f x y =
    match x with
    | [] ->
        match y with
        | [] -> 0
        | n2 :: rest2 -> n2 + (f [] rest2)
    | n1 :: rest1 ->
        match y with
        | [] -> n1 + (f rest1 [])
        | n2 :: rest2 ->
            match (compare n1 n2) with
            | EQ -> n1 + (f rest1 rest2)
            | GT -> n1 + (f rest1 rest2)
            | LT -> n2 + (f rest1 rest2)
```

LLM오답에 없으나 (f rest1 rest2)와 유사

LLM 오답을 이용한 양방향 탐색 유도 알고리즘

양방향탐색 (문법 G, 스펙 Φ, LLM 오답 P_{LLM})

1 : C := ∅, n := 1

2 : repeat

3 : C := **부품식추가**(G, n, P_{LLM})

4 : Q := { □ }

5 : while Q ≠ ∅ do

6 : P := **후보선택**(Q, P_{LLM})

7 : if P 만족 Φ then return P

8 : if P 싹수없음 then continue

9 : h := **구멍선택**(P)

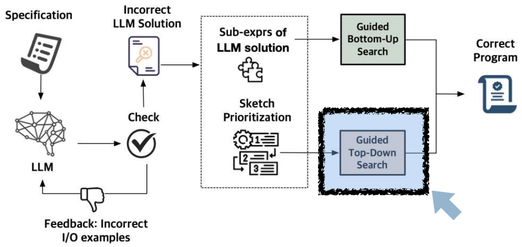
10: Q := Q ∪ **구멍채우기**$_P$(h, G, C)

11 : done; n := n + 1

12: until 시간제한 초과

LLM 오답과 비슷한 후보 먼저 탐색

하향식 탐색 유도



- 공통패턴 뽑기*anti-unification*: 두 프로그램의 공통부분만 남기고 불일치되는 부분을 변수로

```
let rec f x y =
    match x with
    | [] ->
        match y with
        | [] -> 0
        | n2 :: rest2 -> n2 + (f [] rest2)
    | n1 :: rest1 ->
        match y with
        | [] -> n1 + (f rest1 [])
        | n2 :: rest2 ->
            match (compare n1 n2) with
            | EQ -> n1 + (f rest1 rest2)
            | GT -> n1 + (f rest1 rest2)
            | LT -> n2 + (f rest1 rest2)
```

⊔

```
1   let rec f x y =
2       match x with
3       | [] ->
4           match y with
5           | [] -> 0
6           | n2 :: rest2 -> n2
7       | n1 :: rest1 ->
8           match y with
9           | [] -> n1
10          | n2 :: rest2 ->
11              (compare n1 n2) + (f rest1 rest2)
12
13
14
```

=

```
let rec f x y =
    match x with
    | [] ->
        match y with
        | [] -> 0
        | n2 :: rest2 -> X
    | n1 :: rest1 ->
        match y with
        | [] -> Y
        | n2 :: rest2 -> Z
```

패턴 변수

- 후보 프로그램 A와 LLM오답 사이 거리: LLM오답에서 패턴 변수에 해당하는 부분 **크기의 합**
- 거리가 짧은 순으로 후보 프로그램들 탐색 — 구멍은 특별히 취급 (어떤 프로그램도 될 수 있음)
- 복잡도 $O(n)$ 으로 $O(n^2)$ 이상의 복잡도의 문자열/트리수정거리 보다 많은 후보 탐색에 유리

LLM 오답을 이용한 양방향 탐색 유도 알고리즘

양방향탐색 (문법 G, 스펙 Φ, LLM 오답 P_{LLM})

1 : C := ∅, n := 1

2 : repeat

3 : C := **부품식추가**(G, n, P_{LLM})

4 : Q := { □ }

5 : while Q ≠ ∅ do

6 : P := **후보선택**(Q, P_{LLM})

7 : if P 만족 Φ then return P

8 : if P 싹수없음 then continue

9 : h := **구멍선택**(P)

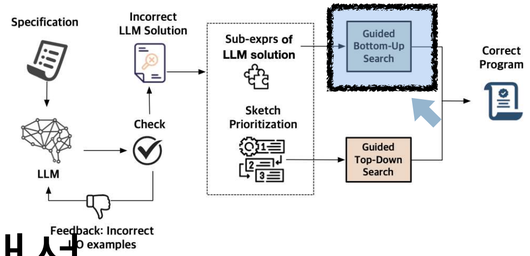
10: Q := Q ∪ **구멍채우기**$_P$(h, G, C)

11 : done; n := n + 1

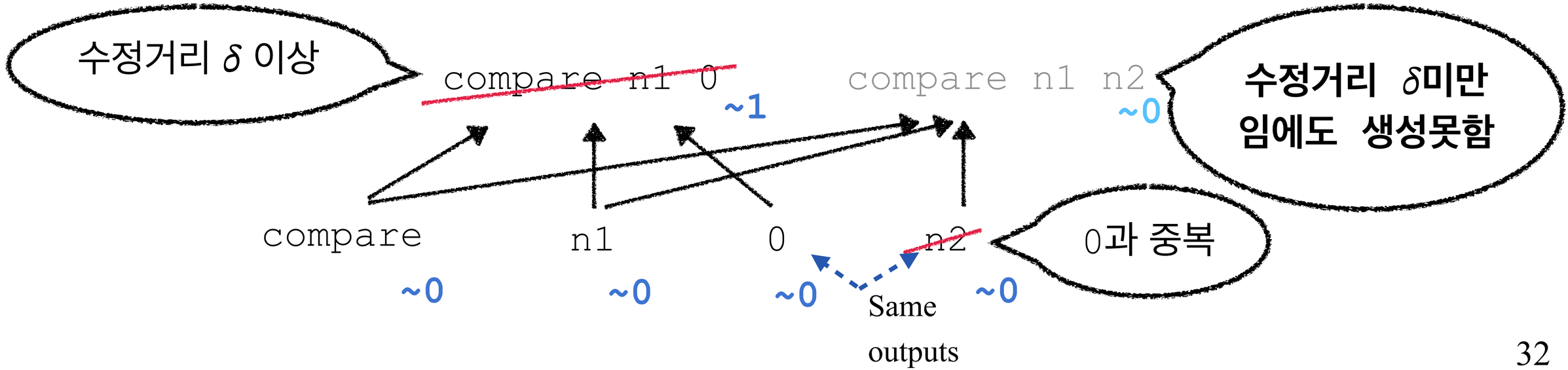
12: until 시간제한 초과

LLM 오답과 비슷한 부품식 추가

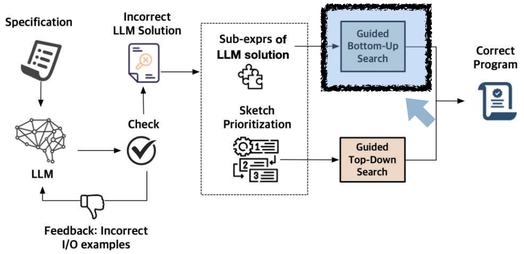
상향식 탐색 유도



- LLM 오답의 부분표현식들 중, 수정거리가 δ 미만인 것이 있는 표현식들 생성
  - δ : LLM에 대한 믿음의 정도 (낮을수록 LLM오답이 정답에 가까울거라 믿음)
- 중복 표현식 제거 시 수정거리가 δ 미만인 식을 생성하지 못하는 경우 발생
  - 예: δ = 1인 상황 (n1, n2 가 각각 첫째 둘째 인자 리스트의 첫원소를 지칭,
    ~k : LLM오답의 부분표현식 중 가장 유사한 것과의 수정거리가 **k**)



상향식 탐색 유도



- 상향식 나열되는 각 후보와 LLM오답의 모든 부분표현식들과의 **수정거리 리스트** 계산
  - LLM오답의 부분표현식들: [`0, n2, n1, x, y, …`]
  - `n2`의 수정거리 리스트: [1, 0, 1, 1, 1, …]
  - `n1`의 수정거리 리스트: [1, 1, 0, 1, 1, …]

`n1`와 `y` 사이 거리

- 중복된 표현식이라도 (즉, 이미 생성된 다른 표현식과 출력이 같더라도) 수정거리 리스트가 파레토 최적이면 제거하지 않고 더 큰 표현식 조립을 위해 사용
  - 수정거리 리스트 L1, L2 에 대해, 모든 L1 원소가 같은 위치의 L2 원소이하면 L1 ≤ L2
  - 어떤 표현식 e에 대해서, 지금까지 나열된 다른 표현식들의 수정거리 리스트 중 e의 것 이하인 수정거리 리스트가 없다면 e는 파레토 최적

LLM오답을 이용한 부품식추가

부품식추가 (문법 G, 제한크기 n)

1 : C := 최소부품, $k := 1$

2 : repeat

3 : for all 생성규칙 따라 생성

4 : e := 새부품 후보

5 : if (|e| > k) ∨ (δ 만족못함) then continue

6 : if (새로운출력 e) ∨ (e 가 파레토최적) then

7 : C := C ∪ {e}

8 : done

9: $k := k + 1$

10: until $k <$ n

11: C := 최적화-하나만남기기(C)

$\delta = 1$

C

0 x y compare

LLM오답을 이용한 부품식추가

부품식추가 (문법 G, 제한크기 n)

1 : C := 최소부품, $k := 1$

2 : repeat

3 : for all 생성규칙 따라 생성

4 : e := 새부품 후보

5 : if (|e| > k) ∨ (δ 만족못함) then continue

6 : if (새로운출력 e) ∨ (e 가 파레토최적) then

7 : C := C ∪ {e}

8 : done

9: $k := k + 1$

10: until $k <$ n

11: C := 최적화-하나만남기기(C)

$\delta = 1$

$k := 2$

C

0 x y compare

n1 rest1 n2 rest2

LLM오답을 이용한 부품식추가

부품식추가 (문법 G, 제한크기 n)

1 : C := 최소부품, $k := 1$

2 : repeat

3 : for all 생성규칙 따라 생성

4 : e := 새부품 후보

5 : if (|e| > k) ∨ (δ 만족못함) then continue

6 : if (새로운출력 e) ∨ (e 가 파레토최적) then

7 : C := C ∪ {e}

8 : done

9: $k := k + 1$

10: until $k <$ n

11: C := 최적화-하나만남기기(C)

$\delta = 1$

$k := 2$

C

0	x	y	compare
n1	rest1	n2	

중복된 출력이지만, C에 있는 부품들보다 작거나 같은 수정거리 리스트를 가져 추가

(n2 수정거리 리스트 ≤ 부품들 수정거리 리스트)

수정거리 리스트

n2: [1, **0**, 1, 1, 1, …]

0 : [0, 1, 1, 1, 1, …]

n1: [1, 1, 0, 1, 1, …]

…

LLM오답을 이용한 부품식추가

부품식추가 (문법 G, 제한크기 n)

1 : C := 최소부품, $k := 1$

2 : repeat

3 : for all 생성규칙 따라 생성

4 : e := 새부품 후보

5 : if (|e| > k) ∨ (δ 만족못함) then continue

6 : if (새로운출력 e) ∨ (e 가 파레토최적) then

7 : C := C ∪ {e}

8 : done

9: $k := k + 1$

10: until $k <$ n

11: C := 최적화-하나만남기기(C)

$\delta = 1$

$k := 4$

C

0 x y compare

n1 rest1 n2 rest2

~~compare n1 0~~

compare n1 n2

수정거리 리스트 중 하나가
δ 보다 작은 경우 없으면 넘어가기

LLM오답을 이용한 부품식추가

부품식추가 (문법 G, 제한크기 n)

1 : C := 최소부품, $k := 1$

2 : repeat

3 : for all 생성규칙 따라 생성

4 : e := 새부품 후보

5 : if (|e| > k) ∨ (δ 만족못함) then continue

6 : if (새로운출력 e) ∨ (e 가 파레토최적) then

7 : C := C ∪ {e}

8 : done

9: $k := k + 1$

10: until k < n

11: C := 최적화-하나만남기기(C)

compare n1 n2 가 만들어지고

중복된 출력을 가진 n2 는 최적화됨

C

0 x y compare

n1 rest1 ~~n2~~ rest2

compare n1 n2

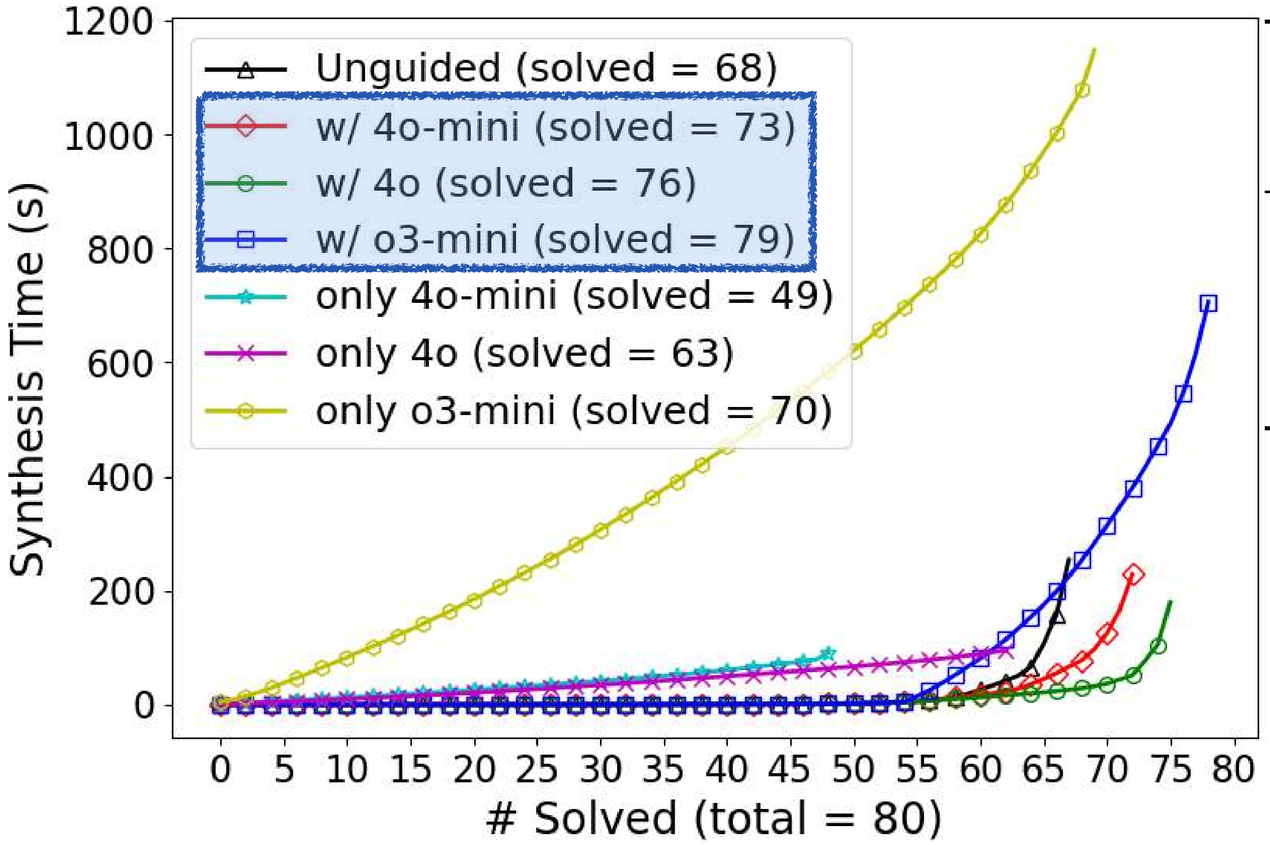
- 세가지 성질을 만족
  - **중복 없음(no redundancy)**
  - **근접성(proximity)**
  - **완전성(completeness)**

실험

- 재귀호출함수 합성기 Trio [POPL'23] 위에 구현, 80개 문제 대상
  - 기존 문제 60개 + 새로 만든 어려운 20개 문제
- 세팅
  - 1초 Trio 돌려보고 못풀면 LLM에 질의
  - LLM이 오답 생성 시 틀린부분 지적해주며 재생성 요구 (3번까지)
  - 끝내 LLM이 정답 생성 못할 경우, 3번 시도 중 가장 괜찮은 오답으로 탐색 유도
- LLM 모델 : GPT-4o-mini (경량), 4o, o3-mini (추론)

실험

- 성능비교



- 상향/하향 유도 효과

Methods	**# Solved**		
	4o-mini	4o	o3-mini
UNGUIDED	68	68	68
+ Top-Down Guidance	70 (↑ **2**)	72 (↑ **4**)	77 (↑ **9**)
+ Both Guidance	73 (↑ **5**)	76 (↑ **8**)	79 (↑ **11**)

- **LLM이나 합성기를 단독으로 쓸 때 못푸는 문제들 다수 해결**

실험

- 설계선택*Design choice* 정당화를 위한 실험
- $O(n)$인 공통패턴뽑기 대신 $O(n^3)$ 인 RTED (트리수정거리) 쓸 경우
  - 73 → 62 (↓9) (w/ 4o-mini)　　76 → 70 (↓6) (w/ 4o)
- LLM에게 100개의 답을 만들게 한 후 PCFG(probabilistic context-free grammar) 를 학습, A* 알고리즘으로 탐색 시
  - 73 → 67 (↓6) (w/ 4o-mini)　　76 → 67 (↓9) (w/ 4o)
  - 기존 논문들에서 흔히 LLM + 합성으로 사용하는 방법

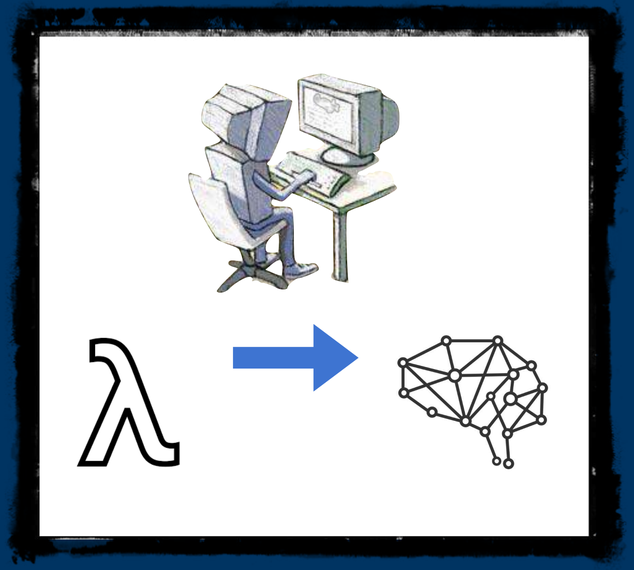
[7] Y. Li, J. Parsert, and E. Polgreen, "Guiding enumerative program synthesis with large language models," in *International Conference on Computer Aided Verification* Springer, 2024, pp. 280–301.

[8] S. Barke, E. Anaya Gonzalez, S. R. Kasibatla, T. Berg-Kirkpatrick, and N. Polikarpova, "Hysynth: Context-free llm approximation for guiding program synthesis," *Advances in Neural Information Processing Systems*, vol. 37, pp. 15 612–15 645, 2024.

[10] Y. Li, J. W. d. S. Magalhães, A. Brauckmann, M. F. O'Boyle, and E. Polgreen, "Guided tensor lifting," *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 1984–2006, 2025.

LLM이 생성하는 코드에서 정의되지 않은 변수 사용 금지하기

박준성 석사과정, 김진상 박사과정

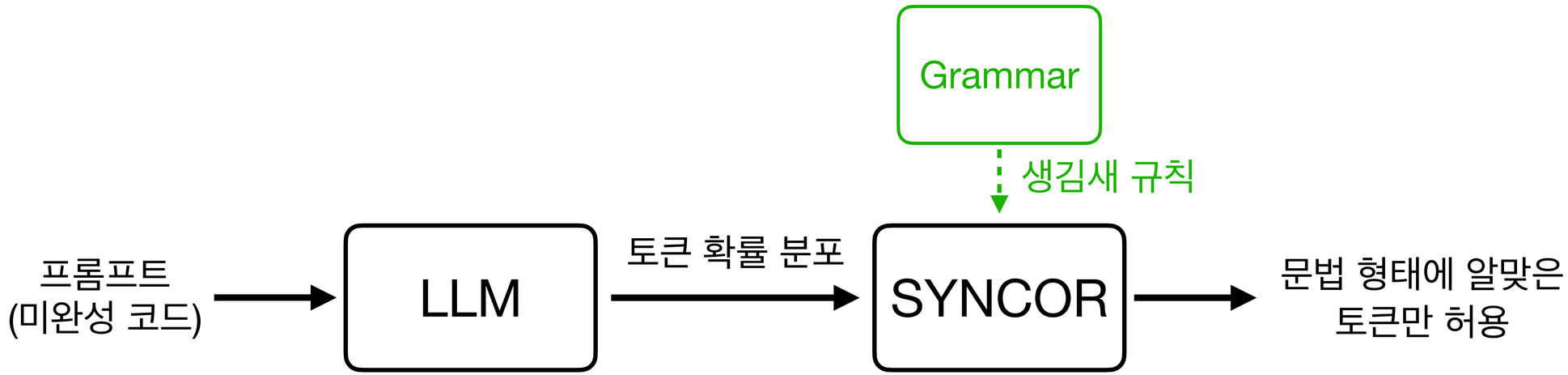


배경지식

- 제한된 코드생성 (Constrained Decoding)
  - LLM이 토큰을 생성할 때 특정 규칙이나 제약을 만족하도록 다음 토큰 선택을 제한
- SYNCOR (이전에 구현)
  - 문법의 생김새 규칙을 기반으로 **문법적 올바름을 강제하는 제한된 코드생성기**

배경지식

- 제한된 코드생성 (Constrained Decoding)
  - LLM이 토큰을 생성할 때 특정 규칙이나 제약을 만족하도록 다음 토큰 선택을 제한
- SYNCOR (이전에 구현)



배경지식

- 제한된 코드생성 (Constrained Decoding)
  - LLM이 토큰을 생성할 때 특정 규칙이나 제약을 만족하도록 다음 토큰 선택을 제한
- SYNCOR (이전에 구현)
  - 문법의 생김새 규칙을 기반으로 **문법적 올바름을 강제하는 제한된 코드생성기**
  - **그러나 문법적으로 올바르다고 해서 미정의 변수 사용 오류가 없는 것은 아님**

목차

1. 문법규칙만으로는 정의되지 않은 변수명 오류를 해결할 수 없음
   → Attribute Grammar: 어떤 변수명이 정의/참조 중 무엇으로 사용되는지 확인

2. 변수명 하나가 여러 토큰에 걸쳐 완성될 수 있음
   → 변수명이 더 길어질 수 있다면 가능한 변수명의 prefix인지 검사

3. 미완성 코드에서 변수명이 정의로 사용될지 참조로 사용될지 미리 파악할 수 없음
   → 변수명 오류 발생을 처리할 2가지 알고리즘

문법규칙의 한계 해결하기

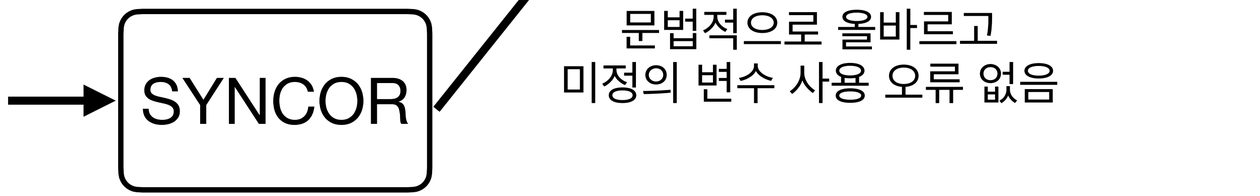
문법규칙만으로는 정의되지 않은 변수명 오류를 해결할 수 없음

- 예제: 다음 생김새 규칙을 기반으로 문법적으로 올바른 코드를 생성하는 SYNCOR

예제를 위한 쉬운 문법: ’변수 값 할당 반복’

```
NAME    := [a-z]+
CONST   := [0-9]+
EQUAL   := “=”
PLUS    := “+”

start   := start stmt | stmt
stmt    := def
def     := NAME EQAUL use
use     := NAME | CONST | NAME PLUS use
```



문법규칙의 한계 해결하기

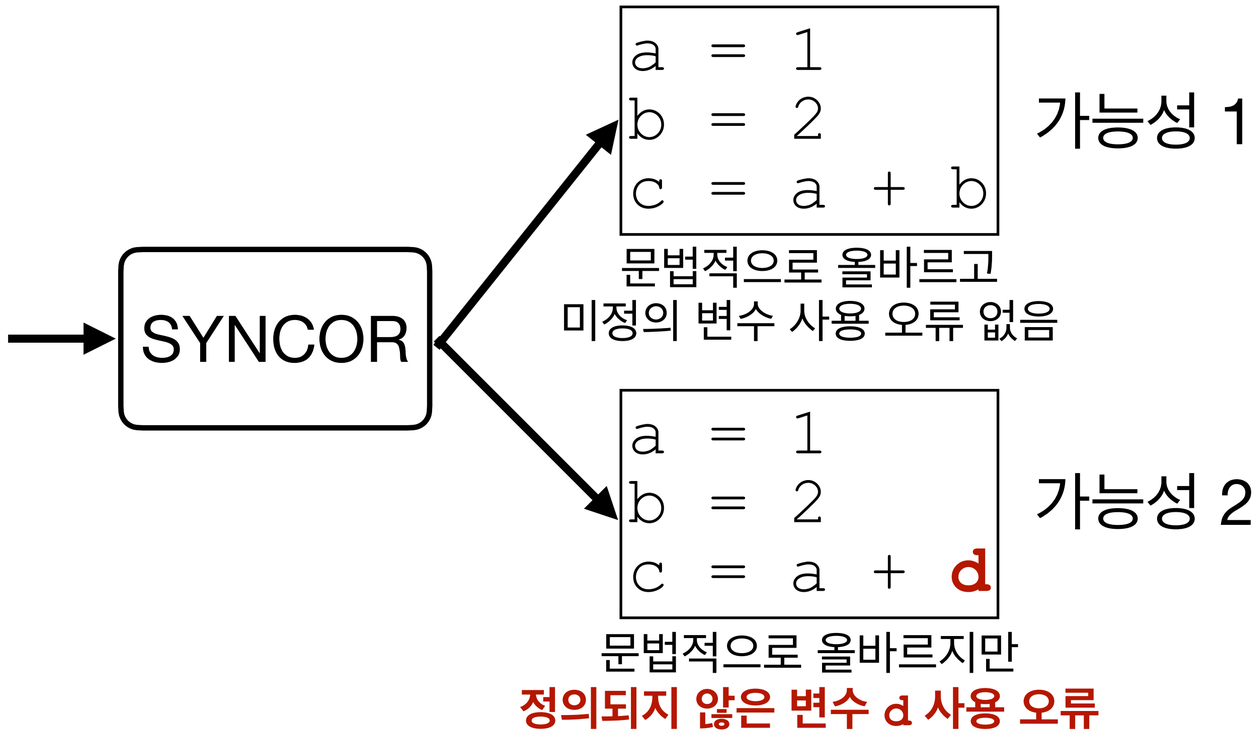
문법규칙만으로는 정의되지 않은 변수명 오류를 해결할 수 없음

- 문법의 생김새 규칙만 알고 있기 때문에,
  어떤 자리에 오는 변수명이 ‘정의’를 위한 변수명인지 ‘참조’를 위한 변수명인지 알 수 없음

예제를 위한 쉬운 문법: ’변수 값 할당 반복’

```
NAME    := [a-z]+
CONST   := [0-9]+
EQUAL   := "="
PLUS    := "+"

start   := start stmt | stmt
stmt    := def
def     := NAME EQAUL use
use     := NAME | CONST | NAME PLUS use
```



문법규칙의 한계 해결하기

태그 속성을 통해 변수명 터미널에 추가적인 의미를 부여하기 (1/2)

- **‘속성 달린 문법’(Attribute Grammar)를 도입**

  어느 위치에서 사용되는 변수명이 ‘정의’/‘참조’인지 확인하기 위한 속성(이하, 태그) 추가

```
NAME    := [a-z]+
CONST   := [0-9]+
EQUAL   := “=”
PLUS    := “+”

start   := start stmt | stmt
stmt    := def
def     := NAME@FREE EQAUL use
use     := NAME@BOUND | CONST | NAME@BOUND PLUS use
```

문법규칙의 한계 해결하기

태그 속성을 통해 변수명 터미널에 추가적인 의미를 부여하기 (2/2)

- 태그는 일종의 ‘출신을 따지는 속성’(inherited attribute)
- 동일한 터미널이라고 할지라도, 어떤 규칙의 일부인지에 따라 태그의 값이 달라짐

```
def := NAME@FREE EQUAL use
```

```
NAME.tag = FREE
EQUAL.tag = None
use.tag = None
```

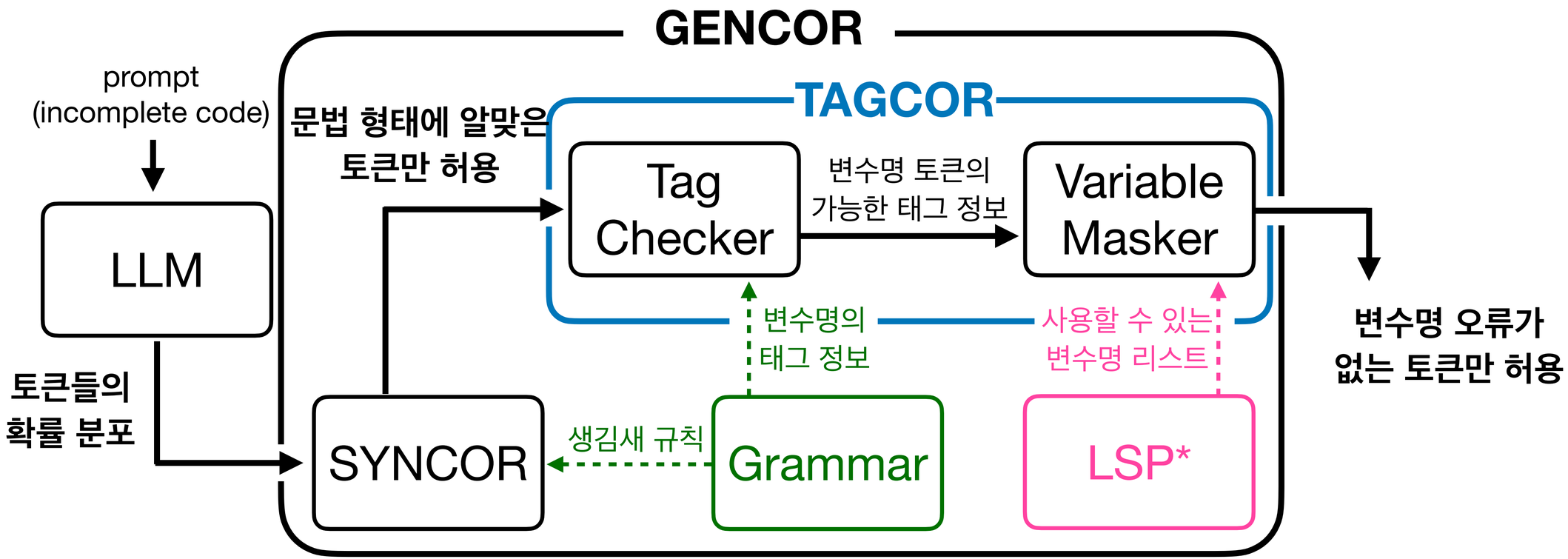
```
use := NAME@BOUND PLUS use
```

```
NAME.tag = BOUND
PLUS.tag = None
use.tag = None
```

문법규칙의 한계 해결하기

제한된 코드생성기 전체 구조

- TAGCOR: **문법적 올바름** + 태그 정보를 참고하여 **변수명 올바름을 보장**

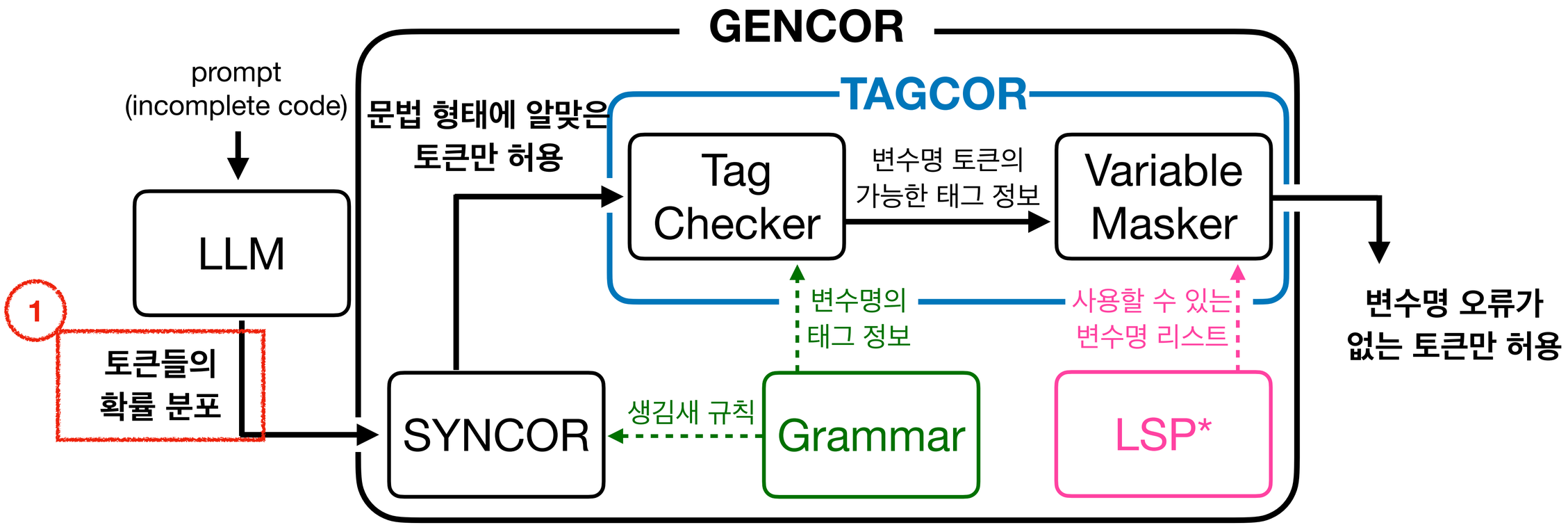


*Language Server Protocol

문법규칙의 한계 해결하기

제한된 코드생성기 전체 구조

- TAGCOR: **문법적 올바름** + 태그 정보를 참고하여 **변수명 올바름을 보장**

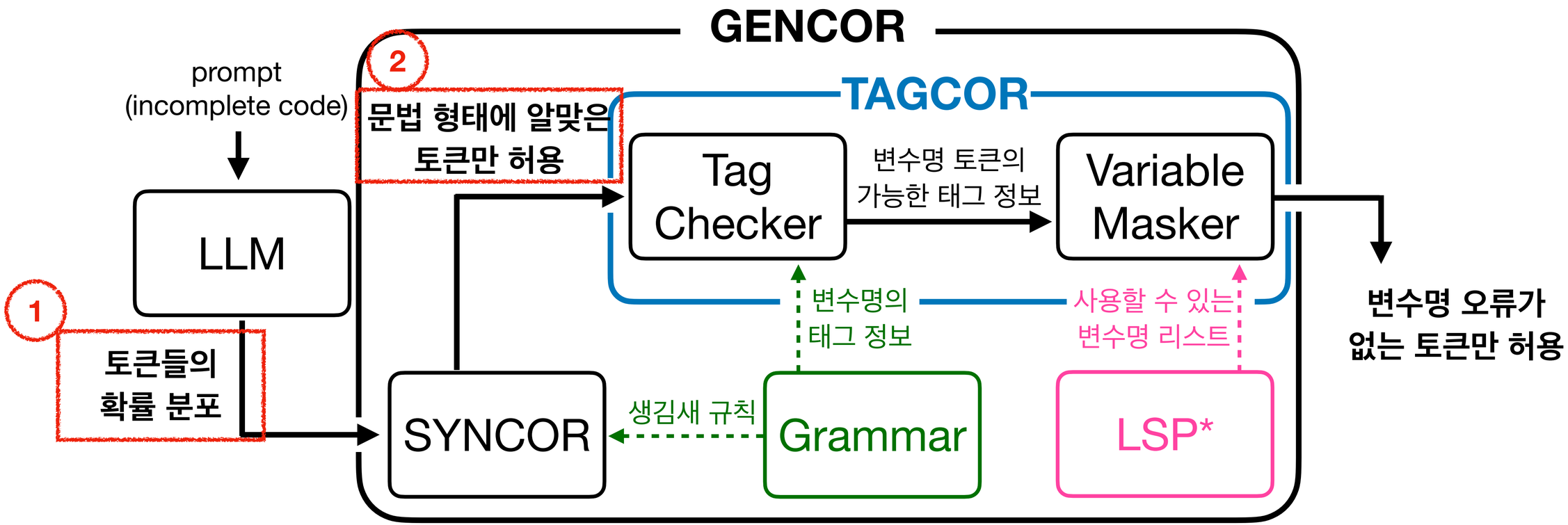


*Language Server Protocol

문법규칙의 한계 해결하기

제한된 코드생성기 전체 구조

- TAGCOR: **문법적 올바름** + 태그 정보를 참고하여 **변수명 올바름을 보장**

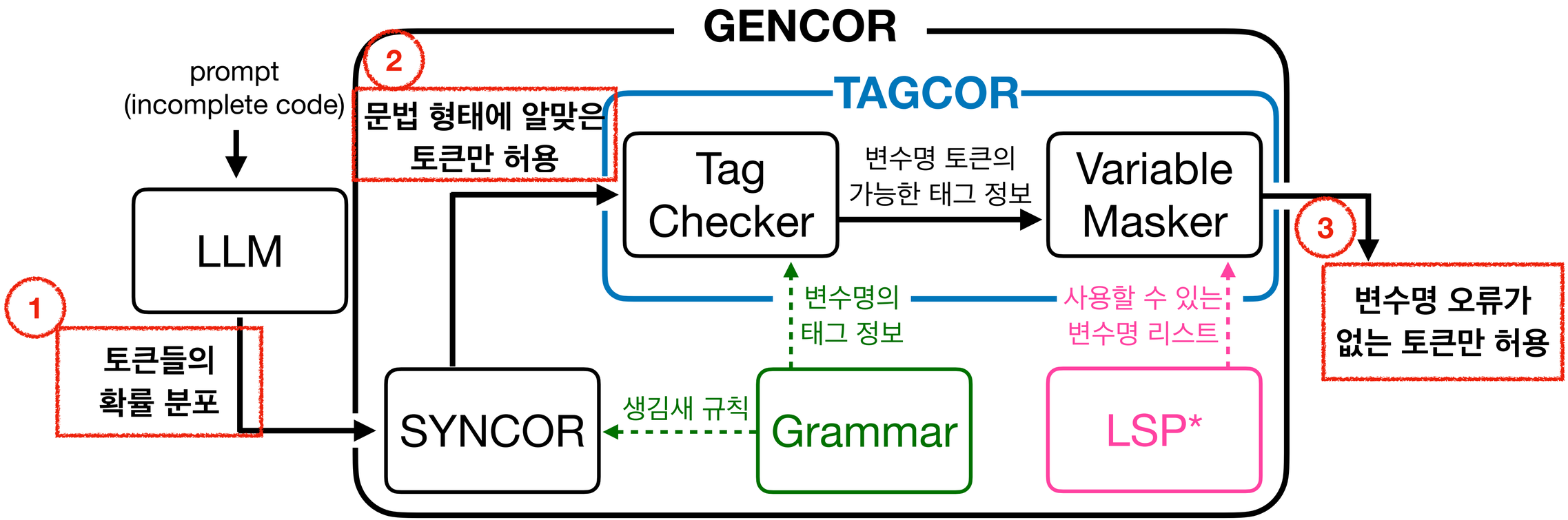


*Language Server Protocol

문법규칙의 한계 해결하기

제한된 코드생성기 전체 구조

- TAGCOR: **문법적 올바름** + 태그 정보를 참고하여 **변수명 올바름을 보장**



*Language Server Protocol

실험

“문법적 올바름만으로는 변수명 오류를 방지할 수 없고 TagCor가 이를 해결”

- LLM: CodeLlama-7b-python-hf, Llama3-8B
- 실험 케이스: MBPP·HumanEval·BigCodeBench 벤치마크에서 SYNCOR 적용 코드 중에서 변수명 오류를 포함한 사례
- 제한 시간: 180s

실험

변수명 오류 교정 성공 사례

- **모델 - 벤치마크: 교정 성공/변수명 오류 케이스**
- CodeLlama-HumanEval: 6/8 (75%)
- MetaLlama3-HumanEval: 3/6 (50%)
- MetaLlama3-MBPP: 4/4 (100%)
- CodeLlama-BigCodeBench: 14/25 (56%)

성공하지 못한 케이스는 주어진 시간내에 코드 생성을 완성하지 못한 경우(Time Out)

실험

교정 성공 사례

HumanEval/8, Llama-3-8B, SYNCOR

```
from typing import List, Tuple

def sum_product(numbers: List[int]) -> Tuple[int, int]:
    """ For a given list of integers,(10, 24)    """
    if not numbers:
        return (0,1)
    else:
        return (sum(numbers), reduce(lambda x, y: x*y, numbers))
```

- 정의되지 않은 reduce 함수명 사용

```
from functools import reduce
```

라이브러리 함수 미사용 → **재귀로 대체**

HumanEval/8, Llama-3-8B, TAGCOR

```
from typing import List, Tuple

def sum_product(numbers: List[int]) -> Tuple[int, int]:
    """ For a given list of integers,(10, 24)    """
    if not numbers:
        return (0,1)
    else:
        return (sum(numbers), numbers[0] * sum_product(numbers[1:])[1])
```

실험

교정 성공 사례

BigCodeBench/242, CodeLlama-7b-Python-hf, SYNCOR

```
import cv2
import matplotlib.pyplot as plt

def task_func(image_path, kernel_size):
    """    Applies a blur effect to an image using a specified kernel size,"""

    if not isinstance(kernel_size, int):
        raise ValueError("kernel_size must be an integer.")
    if kernel_size <=0:
        raise ValueError("kernel_size must be a positive integer.")

    if not isinstance(image_path, str):
        raise ValueError("image_path must be a string.")

    if not os.path.exists(image_path):
        raise FileNotFoundError("The specified image file does not exist.")

    img = cv2.imread(image_path)
    img_blurred = cv2.blur(img, (kernel_size, kernel_size))
```

BigCodeBench/242, CodeLlama-7b-Python-hf, TAGCOR

```
import cv2
import matplotlib.pyplot as plt

def task_func(image_path, kernel_size):
    """    Applies a blur effect to an image using a specified kernel size"""

    if not isinstance(kernel_size, int):
        raise ValueError("kernel_size must be an integer.")
    if kernel_size <=0:
        raise ValueError("kernel_size must be a positive integer.")

    if not isinstance(image_path, str):
        raise ValueError("image_path must be a string.")

    if not cv2.haveImageReader(image_path):
        raise FileNotFoundError("The specified image file does not exist.")

    img = cv2.imread(image_path)
    img_blurred = cv2.blur(img, (kernel_size, kernel_size))
```

- 임포트하지 않은 모듈 os 사용

  사용가능한 라이브러리의 함수 중 가장 비슷한 의미(cv2.haveImageReader)의 함수 사용

실험

교정했지만 의미가 나아지지 않음

BigCodeBench/781, CodeLlama-7b-Python-hf, SYNCOR

```
import os
from datetime import datetime

def task_func(filepath: str) -> dict:
    """Determine the size and date of the last modification of a file."""
    size = os.path.getsize(filepath)
    last_modified =
datetime.fromtimestamp(os.path.getmtime(filepath)).strftime('%Y-%m-%d %H:%M:
%S')
    return {'size': f_size(size), 'last_modified': last_modified}
```

BigCodeBench/781, CodeLlama-7b-Python-hf, TAGCOR

```
import os
from datetime import datetime

def task_func(filepath: str) -> dict:
    """Determine the size and date of the last modification of a file."""
    size = os.path.getsize(filepath)
    last_modified =
datetime.fromtimestamp(os.path.getmtime(filepath)).strftime('%Y-%m-%d %H:%M:
%S')
    return {'size': frozenset([frozenset([('size', size), ('unit',
'bytes')])]), 'last_modified': last_modified}
```

정의되지않은 변수 **f_size** 대신 **frozenset**을 사용했지만 그렇다고 목적에 맞는 코드를 생성한 것은 아님

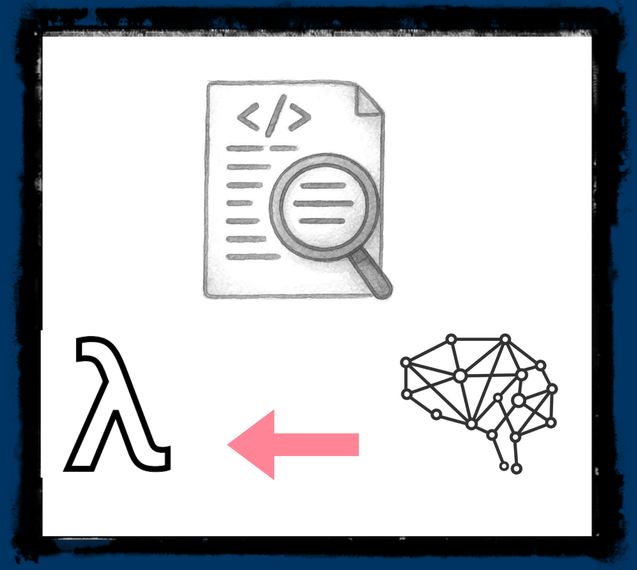
실험

토큰당 소요 시간

모델	토큰 사전 크기	LLM추론시간	SYNCOR 소비시간	TAGCOR 소요시간
meta-llama/Meta-Llama-3-8B	128,256	92ms	127ms	203ms
codellama/CodeLlama-7b-Python-hf	32,000	42ms	78ms	31ms

LLM과 상호작용을 통한 정적분석 허위경보 제거

주강대 석사과정



LLM과의 상호작용을 통해 정적분석 허위 경보를 줄이는 방법

Filtering Static Analysis False Alarms using LLMs

주강대, 이우석

Programming system Lab.

Hanyang university

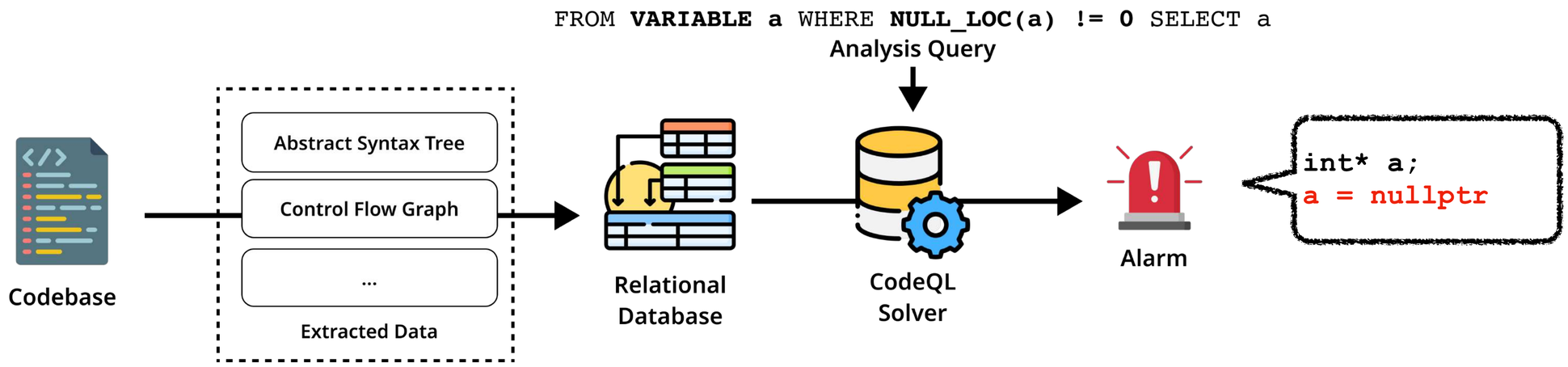
Alexis Just

CY Tech, France

What is CodeQL

본 연구에서는 **정적분석 도구** 중 하나이자 **Github**의 공식 **코드 분석 도구**인 **CodeQL**을 사용

- **Datalog**와 유사한 **선언형 언어**를 이용, **임의의 정적분석**을 사용자가 직접 기술할 수 있음
- 대상 **프로그램**을 **관계형 DB** 형태로 추출하고, **코드**를 마치 **데이터**처럼 **쿼리**하는 방식으로 **정적분석**을 수행



Motivation

- **CodeQL**과 같은 **정적분석** 도구들은 경우에 따라 수 많은 **허위 경보**를 만들어냄
- **허위 경보**를 식별하는 작업은 **노동 집약적**이고 **많은 시간**이 소요
- 최근, **정적분석**의 **허위 경보 문제 개선**을 위해 **LLM을 적용하는 연구**[1][2]에 대한 관심 증가

→ 본 연구는 정적분석의 허위 경보 식별을 위해 LLM을 적용하되
보다 적은 비용으로, 보다 효과적이고 정교하게
자동화하는 것을 목표

[1] Li et al. "Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach" OOPSLA. 2024
[2] Chapman et al. “Interleaving Static Analysis and LLM Prompting." SOAP. 2024

Research Hypothesis

Hypothesis : LLM을 통한 **허위 경보 식별**의 성능과 신뢰도는 **'질문 방식'**의 영향을 크게 받을 것

- **직접 질문 방식(Baseline)**: 문제를 그대로 전달하여, **LLM**에게 **최종 판단**을 요구
  - **“이 경보가 진짜인가?”**

    → **복잡한 추론** 과정 필요하며 **환각(Hallucination)** 및 **부정확성** 증가 할 것

- **우리의 아이디어 (Ours) :** 문제를 더 쉽게 **분해**하여, **LLM**에게 **단순한 사실 확인**만 요구
  - **“이 사실(fact)가 유효한가?”**

    → 훨씬 **정확하고 일관된** 답변을 하며 더 **효율적인** 문제 해결 기대

    → 효과적 문제 해결을 위해 **핵심 사실** 선택이 중요함

Two case studies

우선 아래 두 가지 사례의 문제 해결에 집중:

- **Case 1: 널 접근 (Null Dereference) 분석**
  - 널 접근 분석에서는 불완전성 문제 (Completeness issue)
    → 실제 프로그램에 **존재하지 않는 fact**를 **존재하는 것으로 간주**하여 **허위 경보**가 발생

- **Case 2: 메모리 누수 (Memory Leak) 분석**
  - 메모리 누수 분석에서는 불안전성 문제 (Soundness issue)
    → 실제 프로그램에 **존재하는 fact**를 **존재하지 않는 것으로 간주**하여 **허위 경보**가 발생

실험 대상 및 환경 : Baseline VS Ours

실험 환경 및 대상

실시간 분산 시스템에서 데이터를 효율적으로 교환하기 위한 DDS기반 통신 미들웨어(Middleware)

분석 대상 : FastDDS(C++) / **분석 도구 :** CodeQL / **LLM 모델 :** GPT-4o-mini

- **널 접근 분석 :** 선별된 **17개의 실제 허위 경보 대상**
- **메모리 누수 분석 :** 선별된 **59개의 실제 허위 경보 대상**

실험 방법

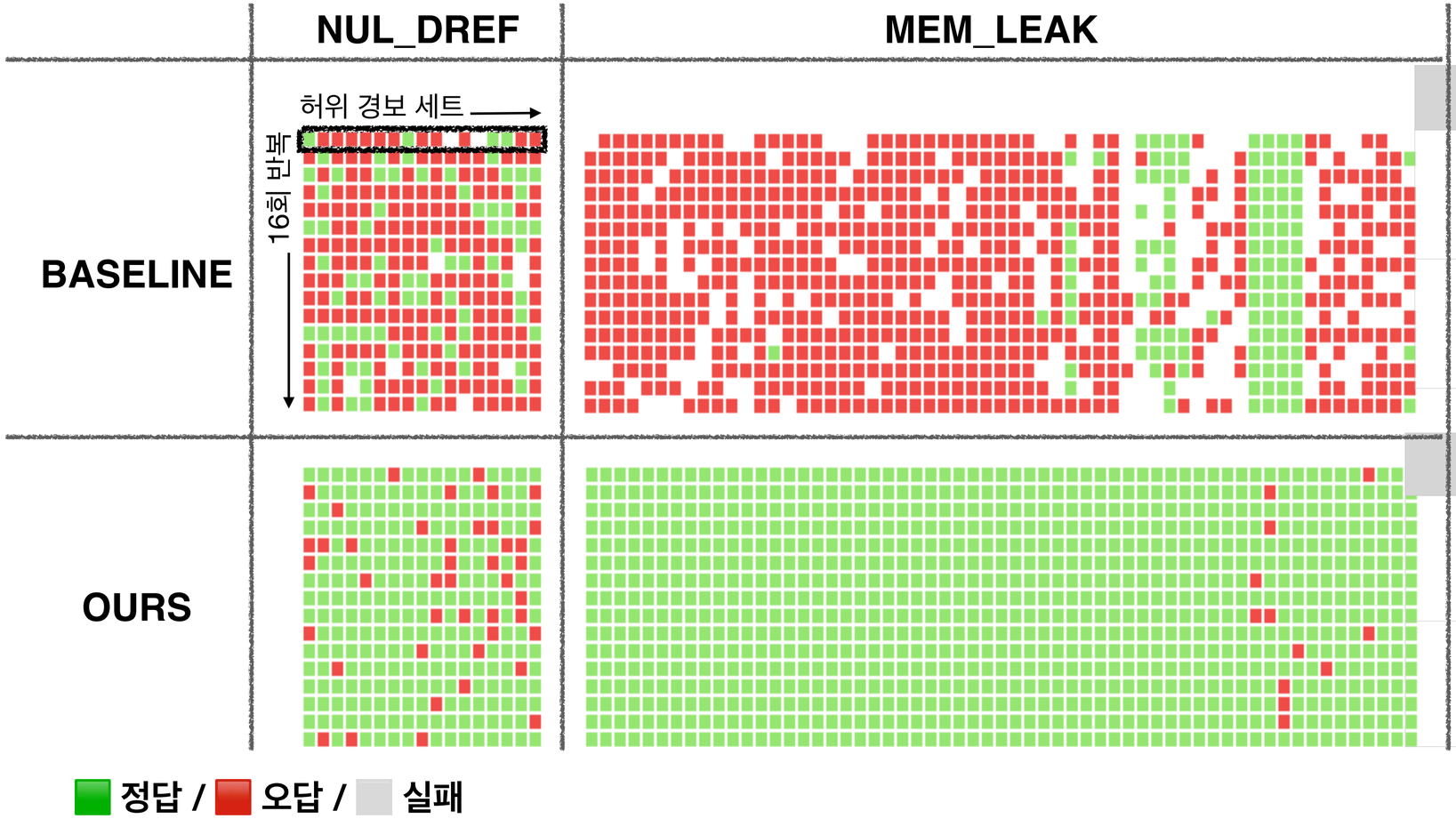
- **Baseline**과 **우리 방식** 의 **허위 경보** 식별 성능 비교
- LLM 답변의 **정답 여부** 및 **일관성** 평가**, 16회 반복**
- 결과의 **응답 분포**를 측정하고 **Entropy** 값으로 수치화 (값이 클수록 답변 일관성↓)

$$H(X) := -\sum_{x \in \mathcal{X}} p(x) \log p(x)$$

Baseline

- **단순하게** 해당 분석 **경보**의 **진위 여부**를 LLM에게 질문
- LLM이 추가적으로 필요한 **코드 요청시 제공** (특정 함수 정의 등)
- **최대 4번**의 **요청/응답**을 반복 후 **결론** 내림

실험 결과 : Baseline VS Ours



허위 경보 식별 수		avg
NUL_DREF (total 17)	Base	4.81
	Ours	14.31
MEM_LEAK (total 59)	Base	7.19
	Ours	58.25

Entropy		avg
NUL_DREF	Base	0.60
	Ours	0.32
MEM_LEAK	Base	0.40
	Ours	0.03

실제 허위 경보를 대상으로 **Baseline**과 성능을 비교한 결과 → **fact 기반 쉬운 질문인 우리 방식**이
허위 경보 제거 성능(정답 수)과 **답변 일관성(Entropy)** 측면에서 훨씬 우수하다는 것을 알 수 있음

향후 과제

현재 우리 방식은 **특정 버그** 또는 **특정 분석에 특화된** 형태로 **일반화**가 많이 **부족한 한계**를 가짐

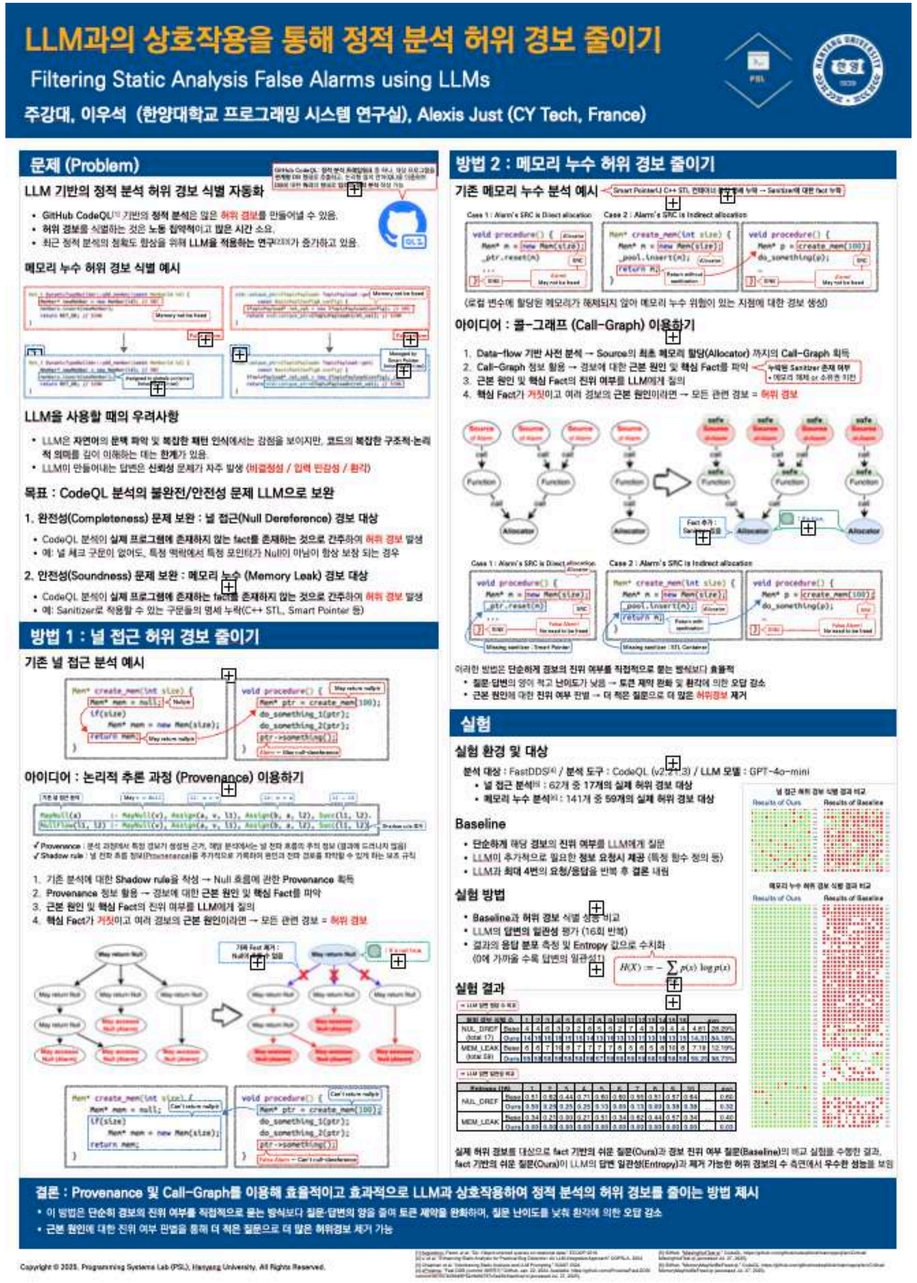
→ **허위경보**를 판별하기 위한 **핵심 Fact**를 **자동으로** 찾아내거나 유도하는 **방식** 찾기

→ 우선 **문제의 범위**를 **축소**하여 정적 분석 전체가 아닌, **오염분석(Taint Analysis)**에 집중하려 함

→ 장기적인 목표는 **LLM**을 사용해 **정적분석** 전반의 신뢰도를 높이는 **일반적인 방법**을 제시하는 것

“일반화에 대한 숙제 존재..!”

Thank you!


LLM과의 상호작용을 통해 정적 분석 허위 경보 줄이기
Filtering Static Analysis False Alarms using LLMs
주강대, 이우석 (한양대학교 프로그래밍 시스템 연구실), Alexis Just (CY Tech, France)
문제 (Problem)
LLM 기반의 정적 분석 허위 경보 식별 자동화
• GitHub CodeQL[1] 기반의 정적 분석은 많은 허위 경보를 만들어낼 수 있음.
• 허위 경보를 식별하는 것은 노동 집약적이고 많은 시간 소요.
• 최근 정적 분석의 정확도 향상을 위해 LLM을 적용하는 연구[2][3][4]가 증가하고 있음.
메모리 누수 허위 경보 식별 예시
LLM을 사용할 때의 우려사항
• LLM은 자연어의 문맥 파악 및 복잡한 패턴 인식에서는 강점을 보이지만, 코드의 복잡한 구조적·논리적 의미를 깊이 이해하는 데는 한계가 있음.
• LLM이 만들어내는 답변은 신뢰성 문제가 자주 발생 (비결정성 / 입력 민감성 / 환각)
목표 : CodeQL 분석의 불완전/안전성 문제 LLM으로 보완
1. 완전성(Completeness) 문제 보완 : 널 접근(Null Dereference) 경보 대상
• CodeQL 분석의 실제 프로그램에 존재하지 않는 fact를 존재하는 것으로 간주하여 허위 경보 발생
• 예: 널 체크 구문이 없어도, 특정 맥락에서 특정 모인터가 Null이 아님이 항상 보장 되는 경우
2. 안전성(Soundness) 문제 보완 : 메모리 누수 (Memory Leak) 경보 대상
• CodeQL 분석이 실제 프로그램에 존재하는 fact를 존재하지 않는 것으로 간주하여 허위 경보 발생
• 예: Sanitizer로 적용할 수 있는 구문들의 명세 누락(C++ STL, Smart Pointer 등)
방법 1 : 널 접근 허위 경보 줄이기
기존 널 접근 분석 예시
아이디어 : 논리적 추론 과정 (Provenance) 이용하기
1. 기존 분석에 대한 Shadow rule을 작성 → Null 흐름에 관한 Provenance 확득
2. Provenance 정보 활용 → 경보에 대한 근본 원인 및 핵심 Fact를 파악
3. 근본 원인 및 핵심 Fact의 진위 여부를 LLM에게 질의
4. 핵심 Fact가 거짓이고 여러 경보의 근본 원인이라면 → 모든 관련 경보 = 허위 경보
방법 2 : 메모리 누수 허위 경보 줄이기
기존 메모리 누수 분석 예시
(로컬 변수에 할당된 메모리가 해제되지 않아 메모리 누수 위험이 있는 지점에 대한 경보 생성)
아이디어 : 콜-그래프 (Call-Graph) 이용하기
1. Data-flow 기반 사전 분석 → Source의 최초 메모리 할당(Allocator) 까지의 Call-Graph 획득
2. Call-Graph 정보 활용 → 경보에 대한 근본 원인 및 핵심 Fact를 파악
3. 근본 원인 및 핵심 Fact의 진위 여부를 LLM에게 질의
4. 핵심 Fact가 거짓이고 여러 경보의 근본 원인이라면 → 모든 관련 경보 = 허위 경보
• 질문-답변의 양이 적고 난이도가 낮음 → 토큰 제약 완화 및 환각에 의한 오답 감소
• 근본 원인에 대한 진위 여부 판별 → 더 적은 질문으로 더 많은 허위경보 제거
실험
실험 환경 및 대상
분석 대상 : FastDDS[5] / 분석 도구 : CodeQL (v2.[illegible]3) / LLM 모델 : GPT-4o-mini
• 널 접근 분석[6] : 62개 중 17개의 실제 허위 경보 대상
• 메모리 누수 분석[6] : 141개 중 59개의 실제 허위 경보 대상
Baseline
• 단순하게 해당 경보의 진위 여부를 LLM에게 질문
• LLM이 추가적으로 필요한 정보 요청시 제공 (특정 함수 정의 등)
• LLM과 최대 4번의 요청/응답을 반복 후 결론 내림
실험 방법
• Baseline과 허위 경보 식별 성능 비교
• LLM의 답변의 일관성 평가 (16회 반복)
• 결과의 응답 분포 측정 및 Entropy 값으로 수치화
(0에 가까울 수록 답변의 일관성↑)
$$H(X) := -\sum p(x)\log p(x)$$
실험 결과
실제 허위 경보를 대상으로 fact 기반의 쉬운 질문(Oura)과 경보 진위 여부 질문(Baseline)의 비교 실험을 수행한 결과, fact 기반의 쉬운 질문(Oura)이 LLM의 답변 일관성(Entropy)과 제거 가능한 허위 경보의 수 측면에서 우수한 성능을 보임
결론 : Provenance 및 Call-Graph를 이용해 효율적이고 효과적으로 LLM과 상호작용하여 정적 분석의 허위 경보를 줄이는 방법 제시
• 이 방법은 단순히 경보의 진위 여부를 직접적으로 묻는 방식보다 질문-답변의 양을 줄여 토큰 제약을 완화하며, 질문 난이도를 낮춰 환각에 의한 오답 감소
• 근본 원인에 대한 진위 여부 판별을 통해 더 적은 질문으로 더 많은 허위경보 제거 가능
Copyright © 2025, Programming Systems Lab (PSL), Hanyang University. All Rights Reserved.