파이썬, 엄밀하게 분석하고 검증하기

오학주
고려대학교 정보대학 컴퓨터학과

2026.10.01@연세대학교

내용

- Part 1 (연구 분야 소개): **프로그램 분석** 및 **자동 수정** 기술
- Part 2 (최신 연구 소개) **엄밀한 파이썬 분석**을 위한 **중간언어**

Part 1: 프로그램 분석 및 자동 수정 기술 소개

소프트웨어 오류 문제

- 소프트웨어 오류 = 사회 모든 영역에서 발생


Knight Capital Says Trading Glitch Cost It $440 Million

BY NATHANIEL POPPER AUGUST 2, 2012 9:07 AM 356

Runaway Trades Spread Turmoil Across Wall St.

금융


Worldwide airport chaos after computer check-in systems crash

Save 31

Long queues at the Thomas Cook check-in desks at Gatwick Airport on Thursday morning CREDIT: JERRY GANDHI/TWITTER

항공


Tesla in fatal California crash was on Autopilot

31 March 2018 Share

자동차


Up to 270 women may have died after breast cancer screening IT error

Health secretary says computer errors led to 450,000 missed screenings in England since 2009

Have you been affected?

의료

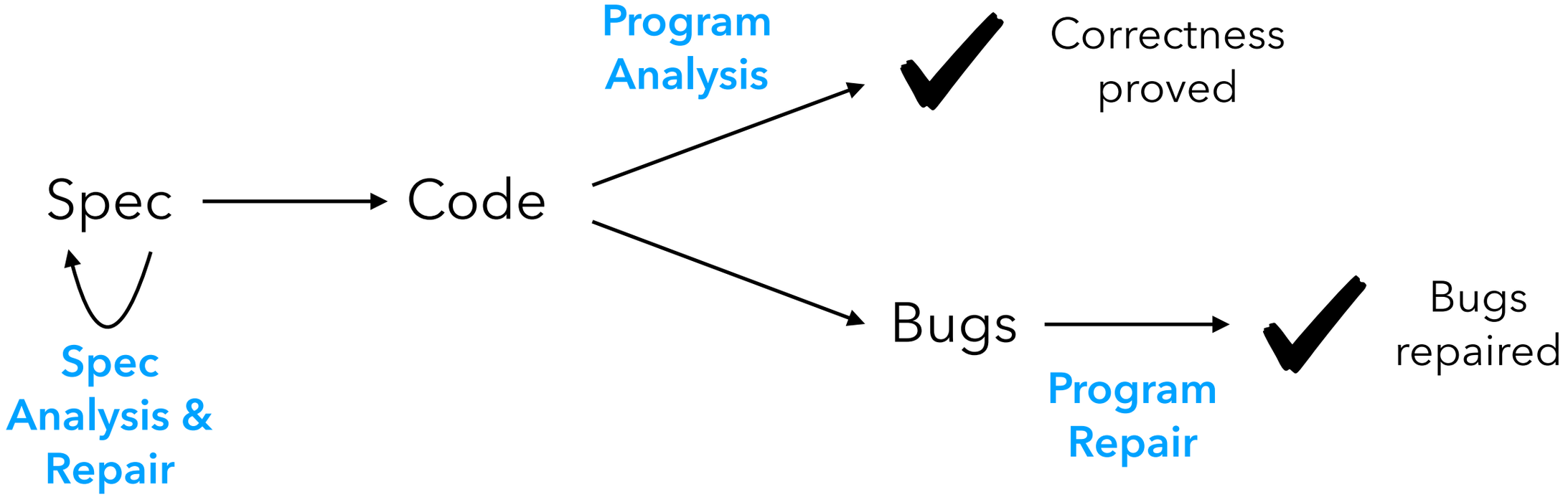
- 사회경제적 비용 = 2.4조 달러 (미국, 2022)



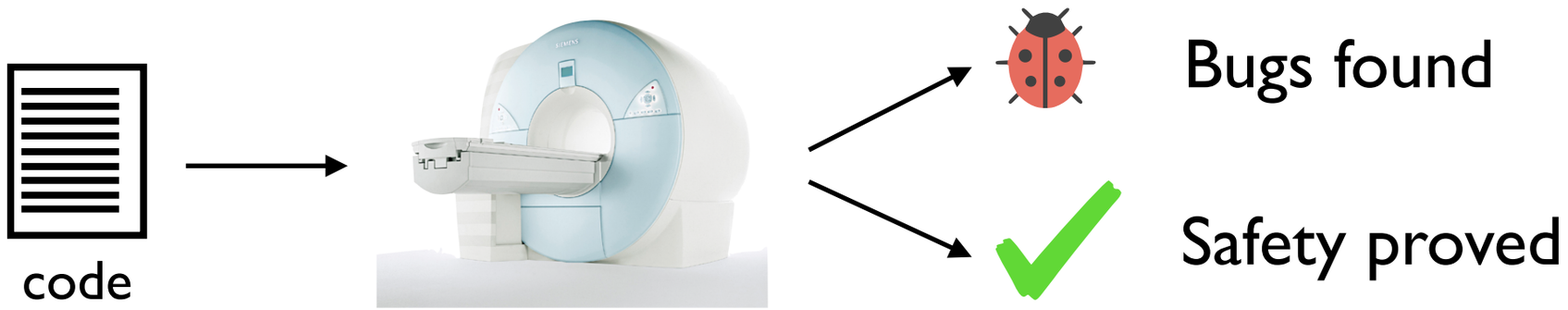
The Cost of Poor Software Quality in the US: A 2022 Report. CISQ

연구 비전

- Q) 어떻게 안전한 소프트웨어를 손쉽게 만들 것인가?
- A) 소프트웨어 **오류 자동 검출 및 자동 수정 기술** 달성



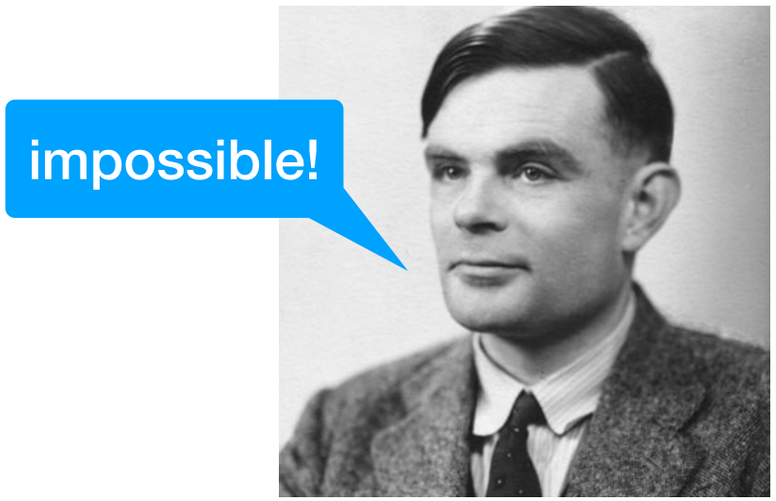
프로그램 분석



- 소프트웨어의 실행 성질을 엄밀히 확인하는 기술
  - 정적 분석: 실행 전 확인 (요약 해석, 모델 체킹 등)
  - 동적 분석: 실행 중 확인 (퍼징, 기호 실행 등)
- 소프트웨어 산업에서 적극적으로 활용되기 시작

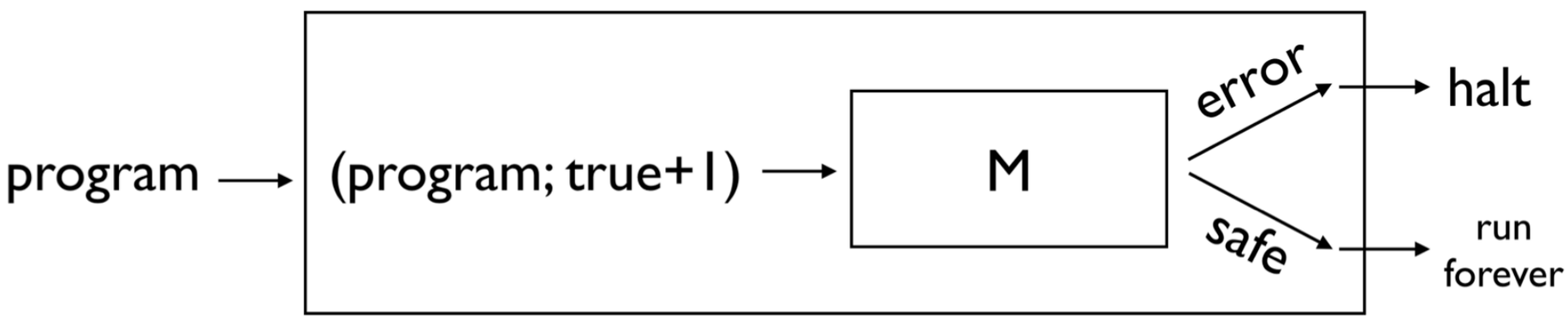
완벽한 프로그램 분석은 불가능

- Halting problem: 주어진 프로그램이 항상 종료하는가?

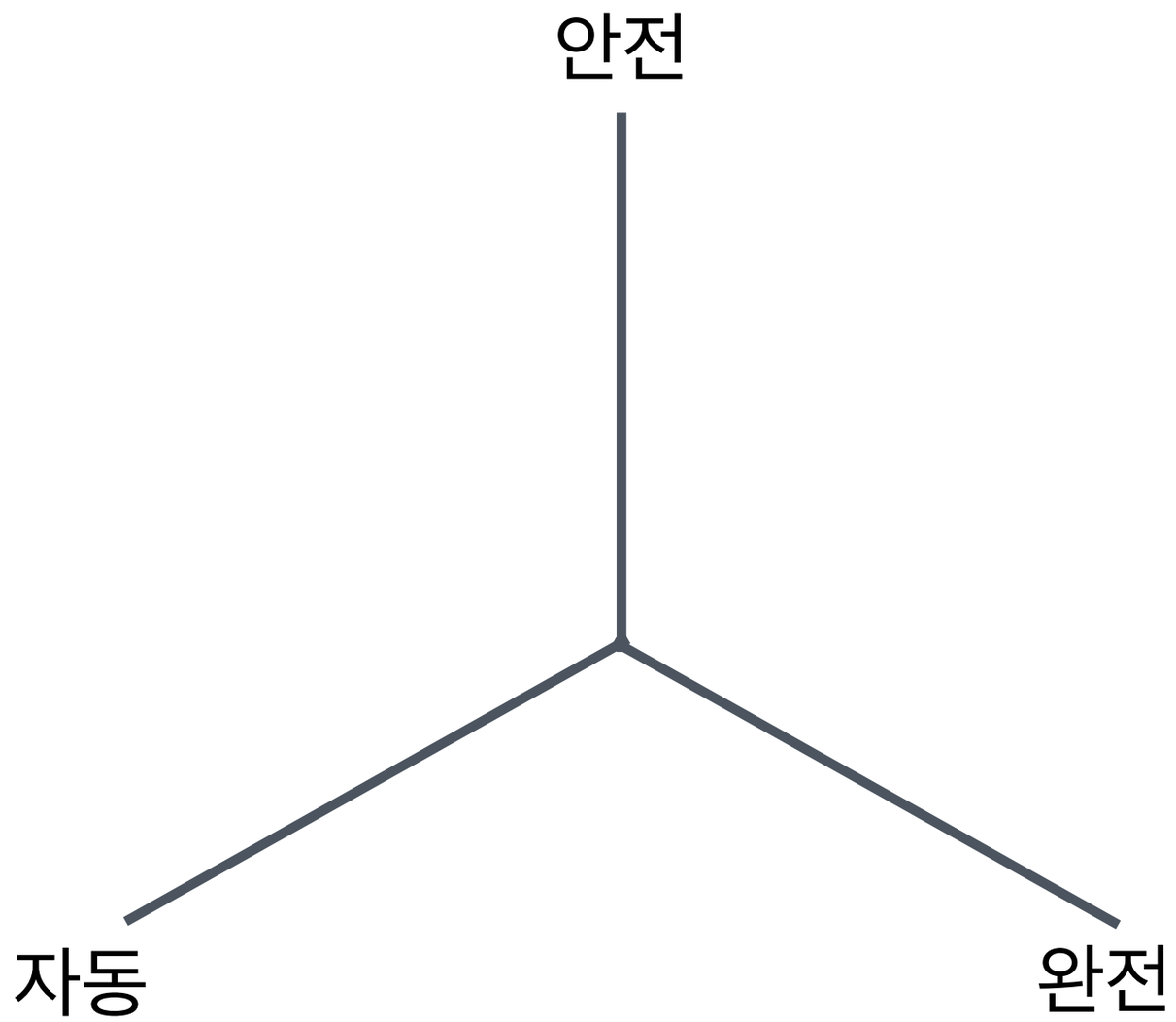


Alan Turing

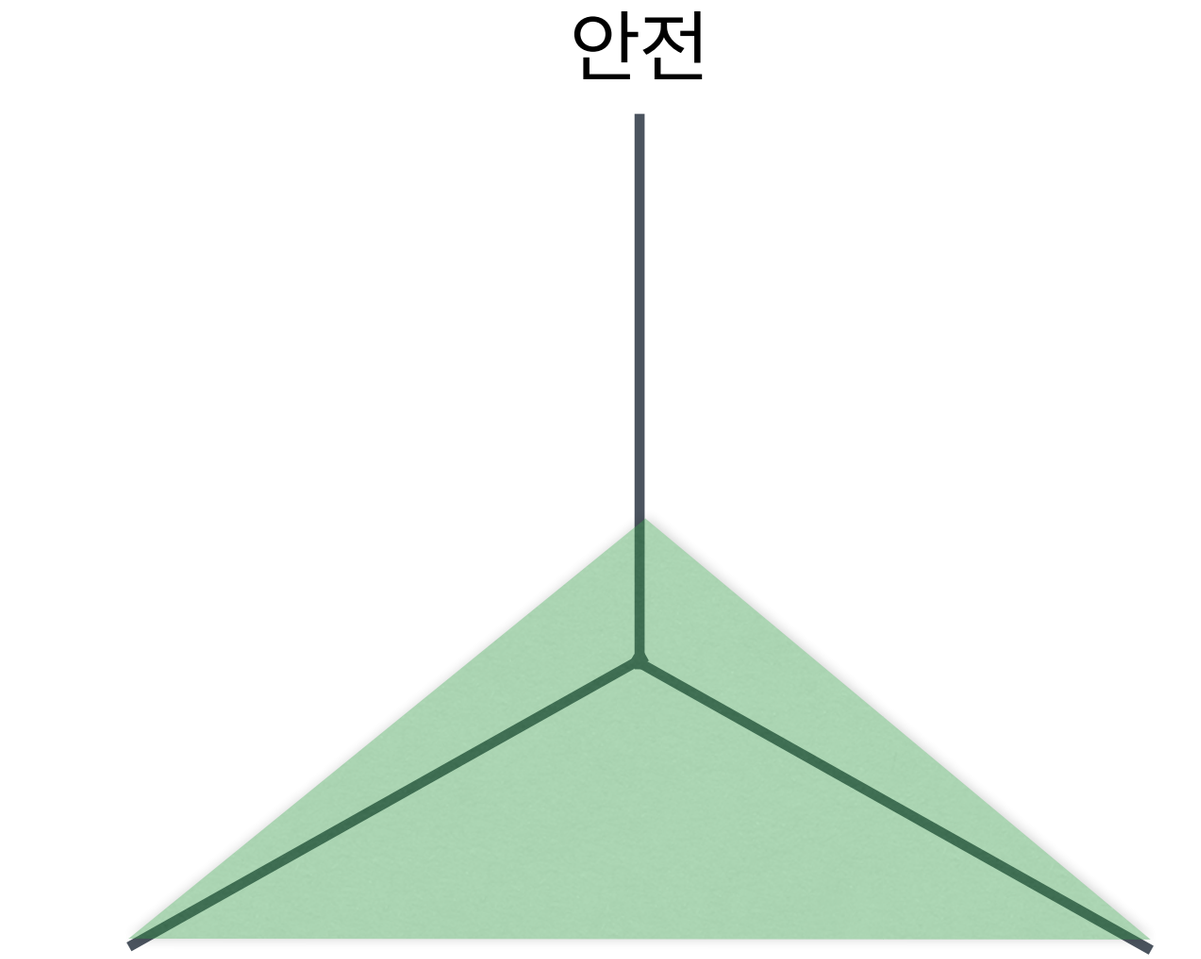
- 완벽한 프로그램 분석기 M이 존재한다면 Halting problem이 풀린다.



프로그램 분석 기법



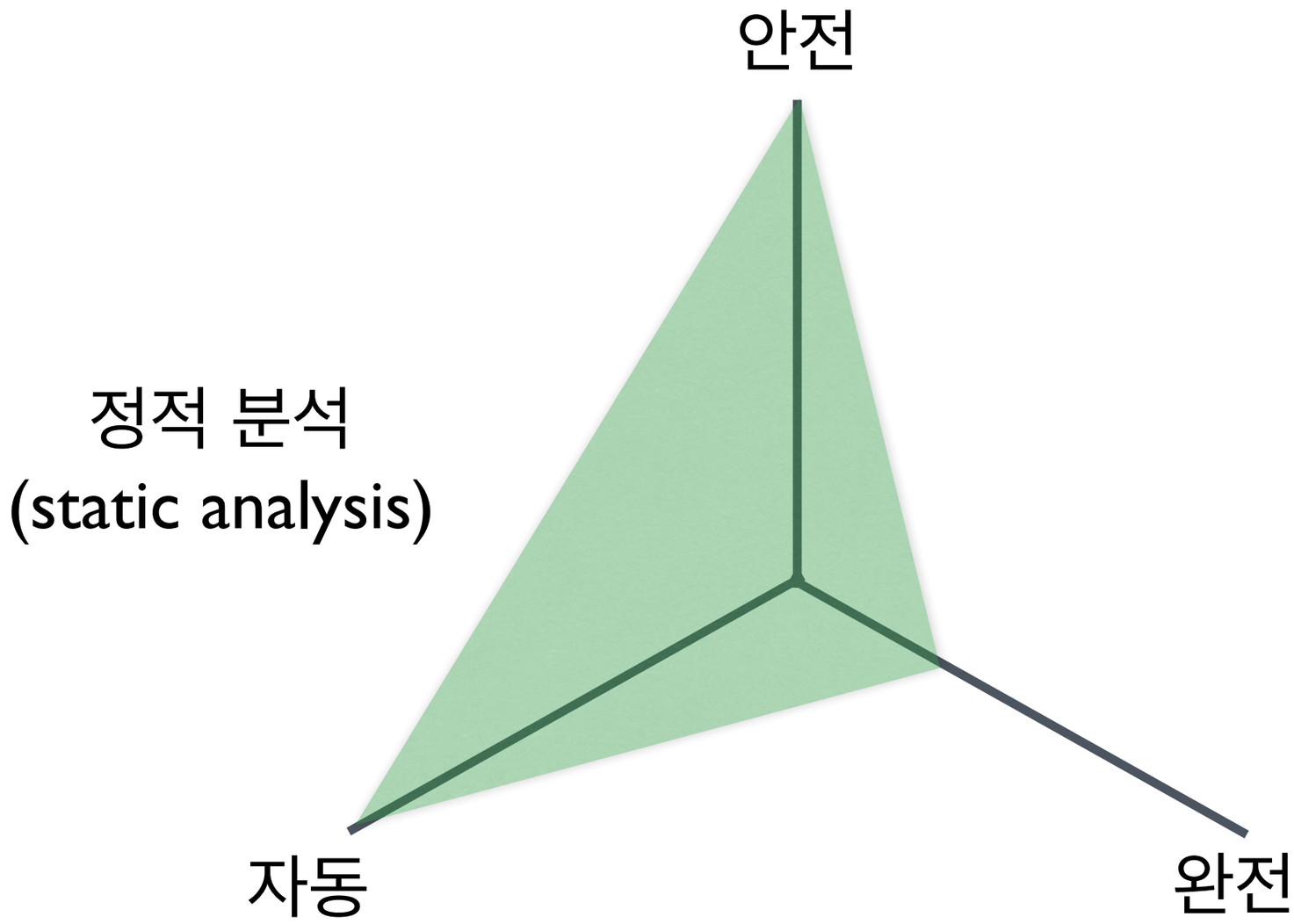
프로그램 분석 기법



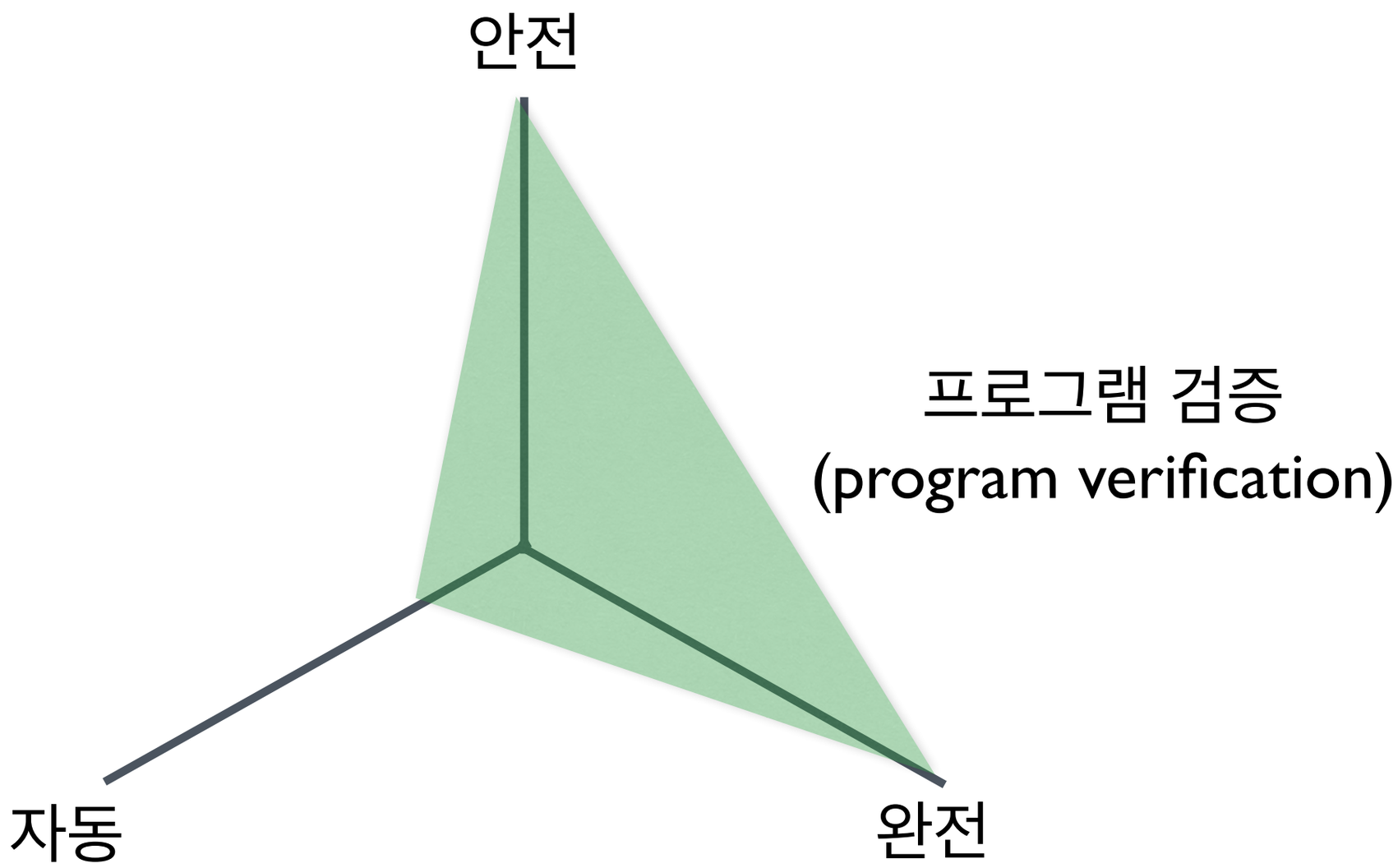
테스팅 / 퍼징 / 기호실행

(testing / fuzzing / symbolic execution)

프로그램 분석 기법



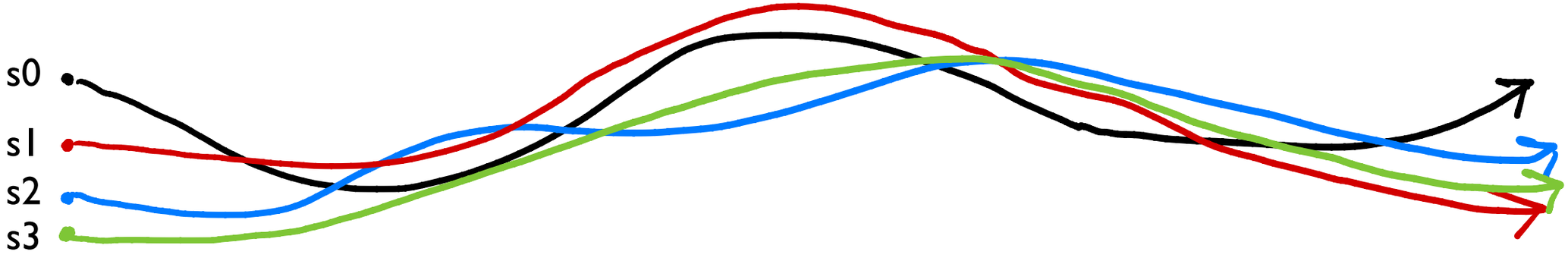
프로그램 분석 기법



테스팅 / 퍼징

- 프로그램의 개별 실행 경로들을 일일이 추적

error zone



error zone

테스팅 / 퍼징

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {
  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

1. Error-triggering test?

2. Probability of the error?
(assume 0 ≤ x,y ≤ 10,000)

기호 실행

- 동일한 실행 경로를 가지는 입력들을 한번에 실행

error zone

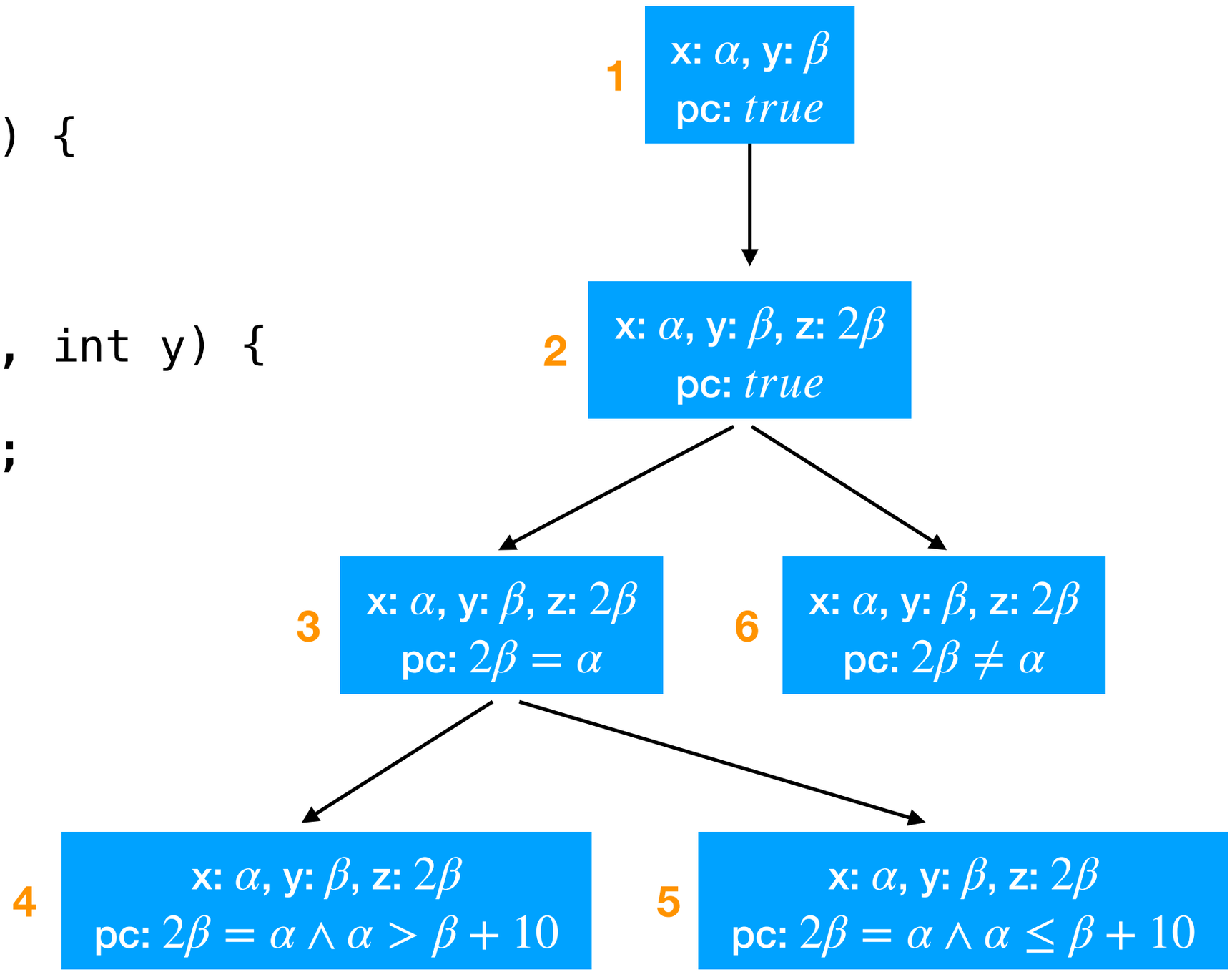


error zone

기호 실행

```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {
1
  z := double (y);
2
  if (z==x) {
   3
    if (x>y+10) {
     4 Error;
    } else { 5 …}
  }
6 }
```



기호 실행 적용 사례

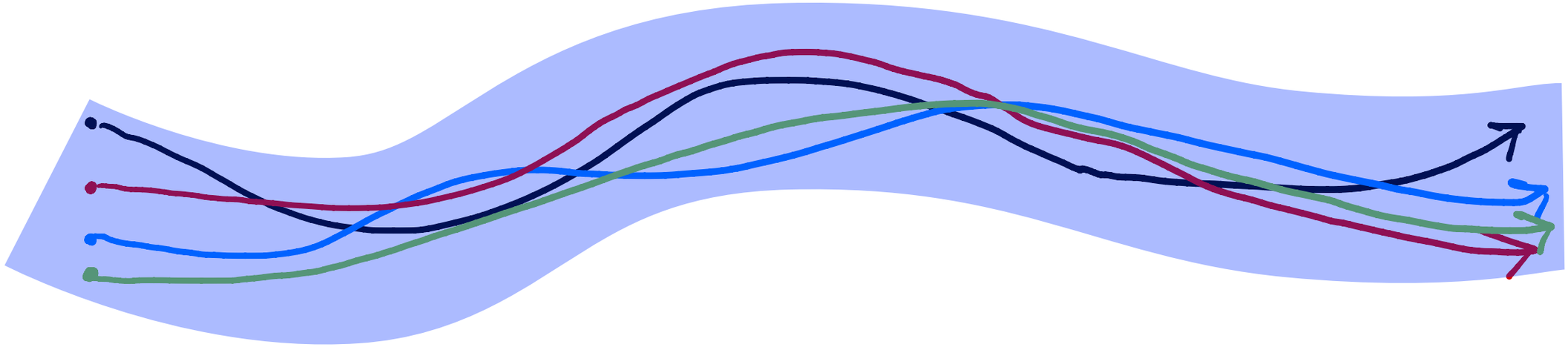
Benchmarks	Versions	Error Types	Bug-Triggering Inputs
vim	8.1*	Non-termination	`K1!1000100100111110(`
	5.7	Abnormal-termination	`H:w>>`"`\` [press 'Enter']
		Segmentation fault	`=ipI\-9~qOqw`
		Non-termination	`v(ipaprq&T$T`
gawk	4.2.1*	Memory-exhaustion	`'+E_Q$h+w$8==++$6E8#'`
	3.0.3	Abnormal-termination	`'f[][][][][y]^/#['`
		Non-termination	`'$g?E2^=-E-2"?^+$=":/?/#["'`
grep	3.1*	Abnormal-termination	`'\(\)\1*?*?|\W*\1W*'`
		Segmentation fault	`'\(\)\1^*@*\?\1*\+*\?'`
	2.2	Segmentation fault	`"_^^*9|^\(\)\'\1*$"`
		Non-termination	`'\({*****\)*\+++\1*\+'`
sed	1.17	Segmentation fault	`'{:};:C;b'`

See "Concolic Testing with Adaptively Changing Search Heuristics. FSE 2019"

정적 분석

- 프로그램 실행을 요약(abstraction)하여 실행

error zone



error zone

정적 분석

$$30 \times 12 + 11 \times 9 = ?$$

- Dynamic analysis (testing): 459
- Static analysis: a variety of answers
  - “integer”
  - “odd integer”
  - “positive integer”
  - “integer between 400 and 500”
  - …

1. Choose abstract value (domain)

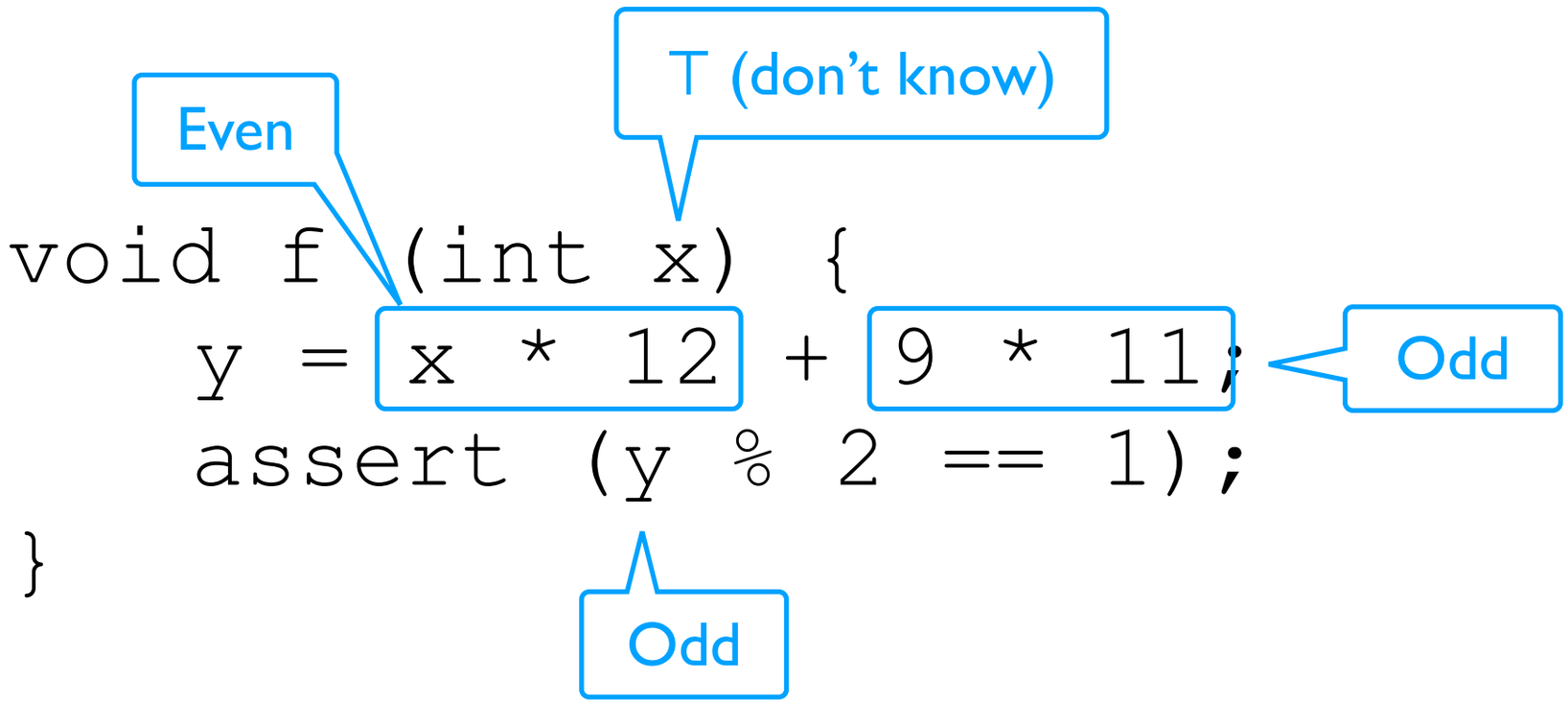
2. “Execute” the program with abstract values

$$e \hat{\times} e \hat{+} o \hat{\times} o = o$$

$$e \hat{\times} e = e \quad e \hat{+} e = e$$
$$e \hat{\times} o = e \quad e \hat{+} o = o$$
$$o \hat{\times} e = e \quad o \hat{+} e = o$$
$$o \hat{\times} o = o \quad o \hat{+} o = e$$

정적 분석 장점

- By contrast to testing, static analysis can prove the absence of bugs



정적 분석 단점

- Instead, static analysis may produce false alarms

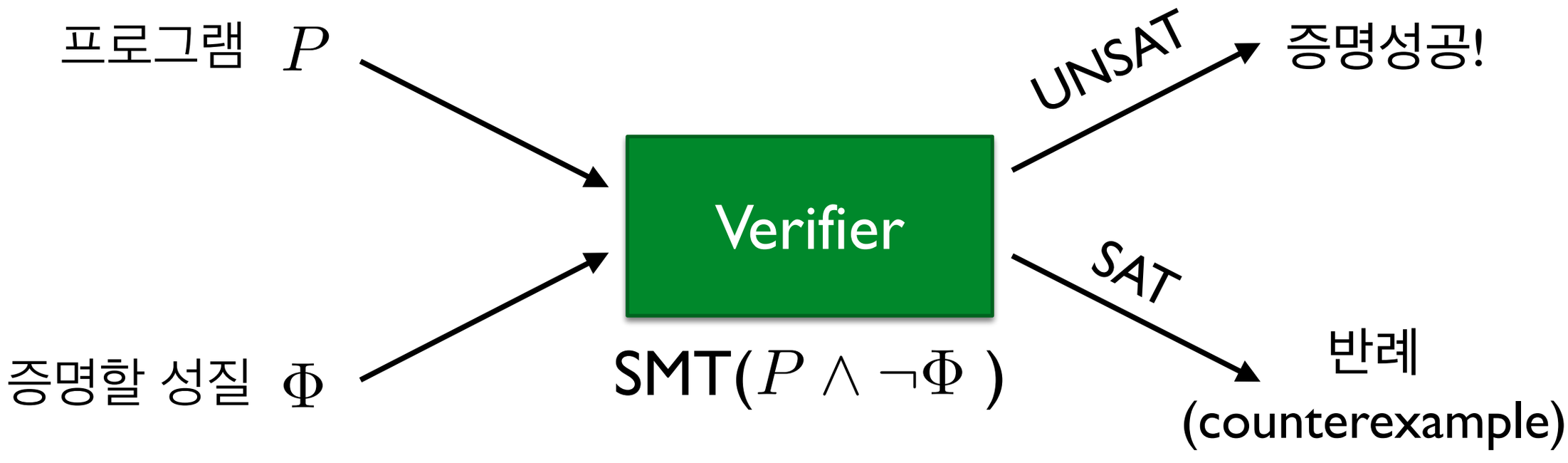
T (don't know)

```
void f (int x) {
  y = x + x;
  assert (y % 2 == 0);
}
```

T (don't know)

false alarm

프로그램 검증

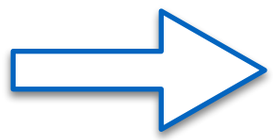


- 프로그램과 증명할 성질을 논리식으로 표현
- 논리식의 satisfiability 여부를 판별

프로그램 검증

```
int f(bool a) {
  x = 0; y = 0;
  if (a) {
    x = 1;
  }
  if (a) {
    y = 1;
  }
  assert (x == y)
}
```

①



Verification Condition:

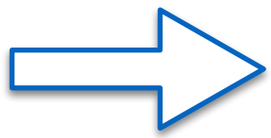
$((a \wedge x) \vee (\neg a \wedge \neg x)) \wedge$
$((a \wedge y) \vee (\neg a \wedge \neg y)) \wedge$
$\neg(x == y)$

② SMT solver: unsatisfiable!

프로그램 검증

```
int f(a, b) {
   x = 0; y = 0;
   if (a) {
      x = 1;
   }
   if (b) {
      y = 1;
   }
   assert (x == y)
}
```

①



Verification Condition:

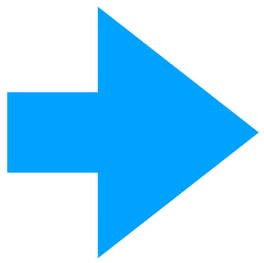
$((a \wedge x) \vee (\neg a \wedge \neg x)) \wedge$
$((b \wedge y) \vee (\neg b \wedge \neg y)) \wedge$
$\neg(x == y)$

②

SMT solver:
satisfiable when $a=1$ and $b=0$

프로그램 검증 장단점

```
bool BubbleSort (int a[]) {
  int[] a := a_0
  for (int i := |a| − 1; i > 0; i := i − 1) {
    for (int j := 0; j < i; j := j + 1) {
      if (a[j] > a[j + 1]) {
        int t := a[j];
        int a[j] := a[j + 1];
        int a[j + 1] := t;
      }
    }
  }
  return a;
}
```

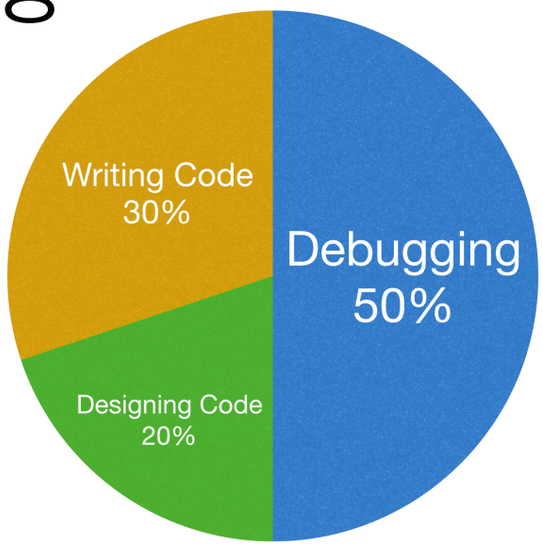


```
@pre : ⊤
@post : sorted(rv, 0, |rv| − 1)
bool BubbleSort (int a[]) {
  int[] a := a_0
  @L1 [ −1 ≤ i < |a|
      ∧ partitioned(a, 0, i, i + 1, |a| − 1)
      ∧ sorted(a, i, |a| − 1) ]
  for (int i := |a| − 1; i > 0; i := i − 1) {
    @L2 [ 1 ≤ i < |a| ∧ 0 ≤ j ≤ i
        ∧ partitioned(a, 0, i, i + 1, |a| − 1)
        ∧ partitioned(a, 0, j − 1, j, j)
        ∧ sorted(a, i, |a| − 1) ]
    for (int j := 0; j < i; j := j + 1) {
      if (a[j] > a[j + 1]) {
        int t := a[j];
        int a[j] := a[j + 1];
        int a[j + 1] := t;
      }
    }
  }
  return a;
}
```

$$\textbf{sorted}(a, l, u) \iff \forall i, j. l \leq i \leq j \leq u \rightarrow a[i] \leq a[j]$$

오류 자동 수정 기술의 필요성

- 개발 생산성은 소프트웨어 산업 경쟁력을 결정하는 주 요인
- 소프트웨어개발에서 가장 생산성이 낮은 단계는 디버깅
  - 개발자들이 가장 부담스러워 하는 단계[1)]
  - 개발자들은 전체 시간의 절반을 디버깅에 사용[2)]
  - 상용 소프트웨어 오류를 수정하는데 평균 200일 소요[3)]



- AI 에이전트가 생성한 코드에 대한 디버깅 부담은 여전히 존재[4, 5)]

1) The art of software testing (3rd edition). 2012
2) Reversible debugging software. University of Cambridge. 2013
3) How long did it take to fix bugs? Mining Software Repositories. 2006
4) Stack overflow developer survey 2025
5) Many SWE-bench-passing PRs would not be merged into main. METR 2026

Linux Kernel 사례

```
in = malloc(1);
out = malloc(1);
... // use in, out
free(out);
free(in);

in = malloc(2);
if (in == NULL) {
  goto err;
}

out = malloc(2);
if (out == NULL) {
  free(in);
  goto err;
}
... // use in, out
err:
  free(in);
  free(out);
  return;
```

메모리 할당

메모리 해제

double-free

메모리 중복 해제
(double-free)

Linux Kernel 사례

USB: fix double frees in error code paths of ipaq driver

```
the error code paths can be enter with buffers to freed buffers.
Serial core would do a kfree() on memory already freed.

Signed-off-by: Oliver Neukum <oneukum@suse.de>
Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>
```

master v4.15-rc1 ... v2.6.24-rc1

Oliver Neukum committed with **gregkh** on 18 Sep 2007 1 par

수동 디버깅의 문제 1:
오류가 제거되었는지 확신하기 어려움

```
in = malloc(1);
out = malloc(1);
... // use in, out
free(out);
free(in);

in = malloc(2);
if (in == NULL) {
  out = NULL;
  goto err;
}

out = malloc(2);
if (out == NULL) {
  free(in);
  in = NULL;
  goto err;
}
... // use in, out
err:
  free(in);
  free(out);
  return;
```

Linux Kernel 사례

수동 디버깅의 문제 2:
오류 수정 과정에서 새로운 오류가 발생

USB: fix double kfree in ipaq in error case

in the error case the ipaq driver leaves a dangling pointer to already freed memory that will be freed again.

Signed-off-by: Oliver Neukum <oneukum@suse.de>
Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>

master v4.15-rc1 ... v2.6.27-rc1

Oliver Neukum committed with gregkh on 30 Jun 2008 1 parent 35

9개월 후에 다시 오류 수정을 시도

memory leak

```
in = malloc(1);
out = malloc(1);
... // use in, out
free(out);
free(in);

in = malloc(2);
if (in == NULL) {
  out = NULL;
  goto err;
}
free(out);
out = malloc(2);
if (out == NULL) {
  free(in);
  in = NULL;
  goto err;
}
... // use in, out
err:
  free(in);
  free(out);
  return;
```

Linux Kernel 사례

수동 디버깅의 문제 3:
오류는 제거했지만 코드 품질이 떨어짐



오류 발견에서 수정까지 총 10개월 소요

```
in = malloc(1);
out = malloc(1);
... // use in, out
free(out);
free(in);
out = NULL;
in = malloc(2);
if (in == NULL) {
  out = NULL;
  goto err;
}
free(out);
out = malloc(2);
if (out == NULL) {
  free(in);
  in = NULL;
  goto err;
}
... // use in, out
err:
  free(in);
  free(out);
  return;
```

오류 자동 수정 및 검증기

```
in = malloc(1);
out = malloc(1);
... // use in, out
free(out);
free(in);

in = malloc(2);
if (in == NULL) {
  goto err;
}

out = malloc(2);
if (out == NULL) {
  free(in);

  goto err;
}
... // use in, out
err:
  free(in); // double-free
  free(out);// double-free
  return;
```

오류 자동 수정 기술

✓개발생산성↑
✓SW품질↑
✓패치 검증

```
in = malloc(1);
out = malloc(1);
... // use in, out
free(out);
free(in);

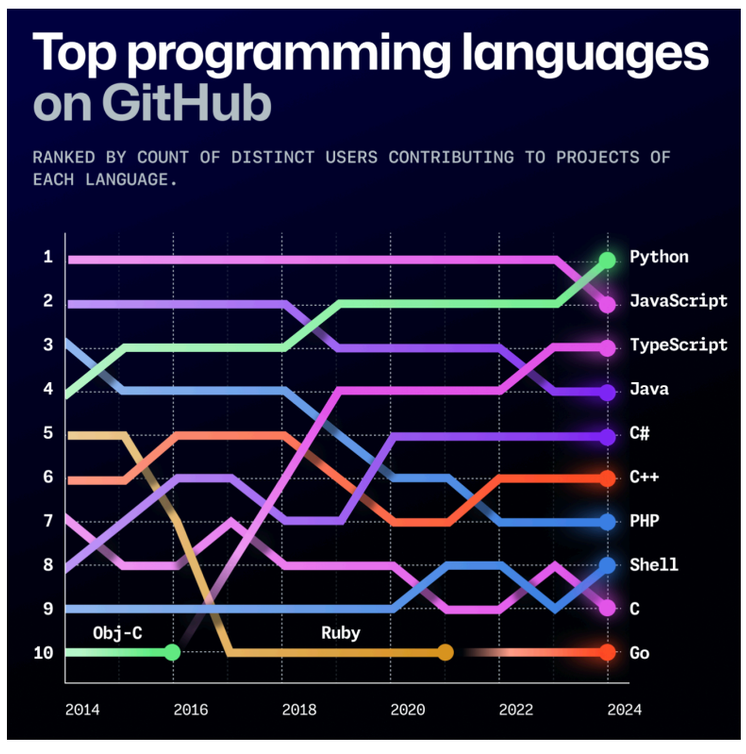
in = malloc(2);
if (in == NULL) {
  goto err;
}
free(out);
out = malloc(2);
if (out == NULL) {
  free(in);

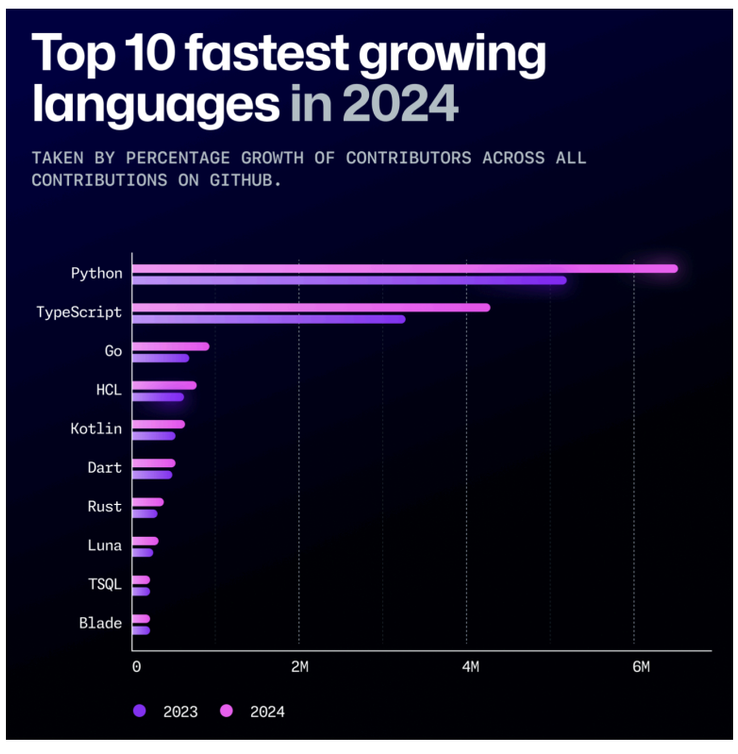
  goto err;
}
... // use in, out
err:
  free(in);
  free(out);
  return;
```

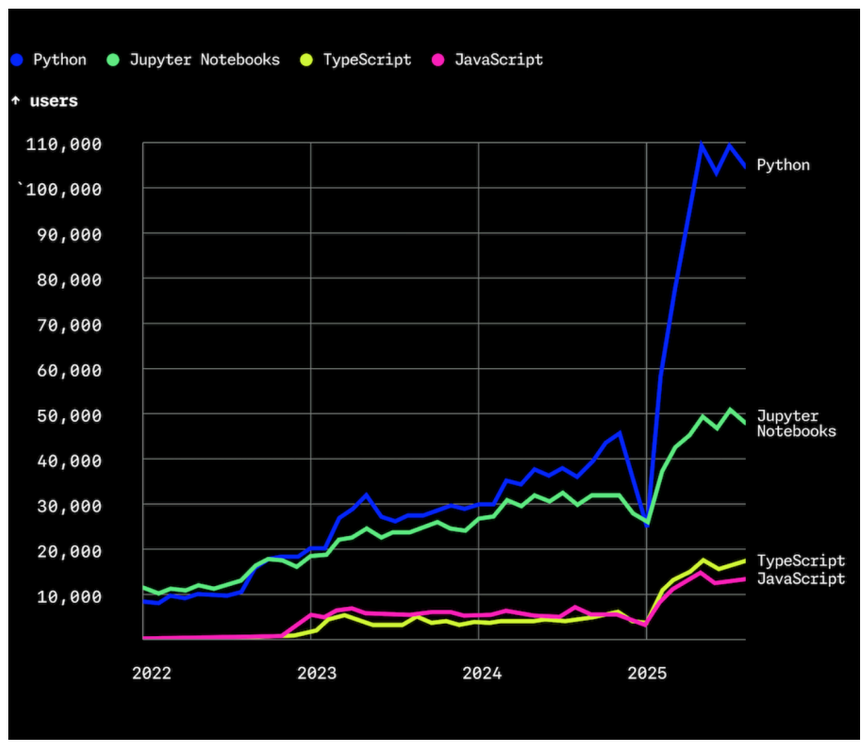
Part 2: 파이썬 프로그램 분석을 위한 중간언어

Python = 현대 SW 생태계의 표준 언어

- 가장 널리 사용되며 가장 빠르게 성장하고 있는 프로그래밍 언어
- AI 시대의 표준 프로그래밍 언어: AI가 가장 많이, 가장 잘 쓰는 언어

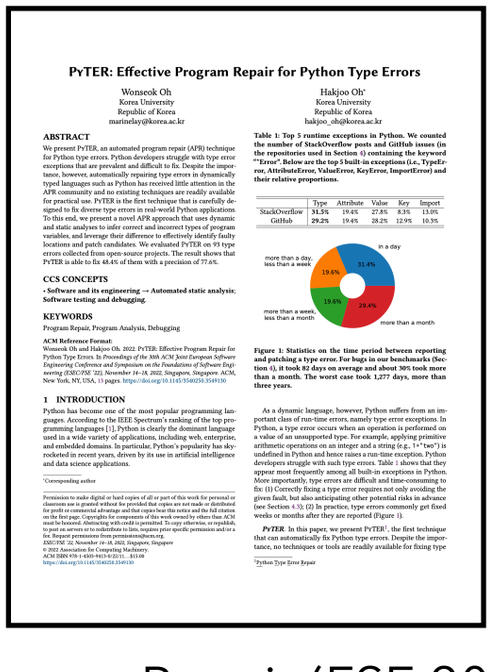






GitHub Octoverse report 2024

파이썬 프로그램 분석 및 검증



PyTER: Effective Program Repair for Python Type Errors

Wonseok Oh
Korea University
Republic of Korea
marinelay@korea.ac.kr

Hakjoo Oh*
Korea University
Republic of Korea
hakjoo_oh@korea.ac.kr

ABSTRACT

We present PyTER, an automated program repair (APR) technique for Python type errors. Python developers struggle with type error exceptions that are prevalent and difficult to fix. Despite the importance, however, automatically repairing type errors in dynamically typed languages such as Python has received little attention in the APR community and no existing techniques are readily available for practical use. PyTER is the first technique that is carefully designed to fix diverse type errors in real-world Python applications. To this end, we present a novel APR approach that uses dynamic and static analyses to infer correct and incorrect types of program variables, and leverage their difference to effectively identify faulty locations and patch candidates. We evaluated PyTER on 93 type errors collected from open-source projects. The result shows that PyTER is able to fix 48.4% of them with a precision of 77.6%.

CCS CONCEPTS

• Software and its engineering → Automated static analysis; Software testing and debugging.

KEYWORDS

Program Repair, Program Analysis, Debugging

ACM Reference Format:
Wonseok Oh and Hakjoo Oh. 2022. PyTER: Effective Program Repair for Python Type Errors. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549130

1 INTRODUCTION

Python has become one of the most popular programming languages. According to the IEEE Spectrum's ranking of the top programming languages [1], Python is clearly the dominant language used in a wide variety of applications, including web, enterprise, and embedded domains. In particular, Python's popularity has skyrocketed in recent years, driven by its use in artificial intelligence and data science applications.

*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549130

Table 1: Top 5 runtime exceptions in Python. We counted the number of StackOverflow posts and GitHub issues (in the repositories used in Section 4) containing the keyword "Error". Below are the top 5 built-in exceptions (i.e., TypeError, AttributeError, ValueError, KeyError, ImportError) and their relative proportions.

	Type	Attribute	Value	Key	Import
StackOverflow	31.5%	19.4%	27.8%	8.3%	13.0%
GitHub	29.2%	19.4%	28.2%	12.9%	10.3%

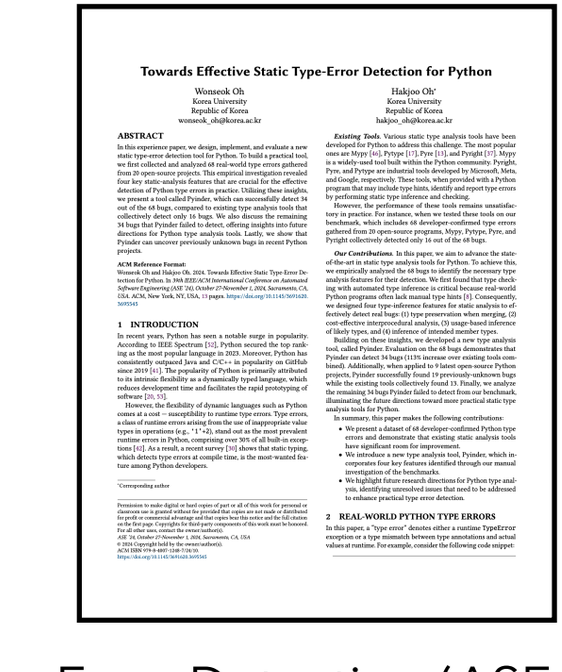
Figure 1: Statistics on the time period between reporting and patching a type error. For bugs in our benchmarks (Section 4), it took 82 days on average and about 30% took more than a month. The worst case took 1,277 days, more than three years.

As a dynamic language, however, Python suffers from an important class of run-time errors, namely type error exceptions. In Python, a type error occurs when an operation is performed on a value of an unsupported type. For example, applying primitive arithmetic operations on an integer and a string (e.g., 1+"two") is undefined in Python and hence raises a run-time exception. Python developers struggle with such type errors. Table 1 shows that they appear most frequently among all built-in exceptions in Python. More importantly, type errors are difficult and time-consuming to fix: (1) Correctly fixing a type error requires not only avoiding the given fault, but also anticipating other potential risks in advance (see Section 4.3); (2) In practice, type errors commonly get fixed weeks or months after they are reported (Figure 1).

PyTER. In this paper, we present PyTER[1], the first technique that can automatically fix Python type errors. Despite the importance, no techniques or tools are readily available for fixing type

[1]Python Type Error Repair

Program Repair (FSE 2022)



Towards Effective Static Type-Error Detection for Python

Wonseok Oh
Korea University
Republic of Korea
wonseok_oh@korea.ac.kr

Hakjoo Oh*
Korea University
Republic of Korea
hakjoo_oh@korea.ac.kr

ABSTRACT

In this experience paper, we design, implement, and evaluate a new static type-error detection tool for Python. To build a practical tool, we first collected and analyzed 68 real-world type errors gathered from 20 open-source projects. This empirical investigation revealed four key static-analysis features that are crucial for the effective detection of Python type errors in practice. Utilizing these insights, we present a tool called Pyinder, which can successfully detect 34 out of the 68 bugs, compared to existing type analysis tools that collectively detect only 16 bugs. We also discuss the remaining 34 bugs that Pyinder failed to detect, offering insights into future directions for Python type analysis tools. Lastly, we show that Pyinder can uncover previously unknown bugs in recent Python projects.

ACM Reference Format:
Wonseok Oh and Hakjoo Oh. 2024. Towards Effective Static Type-Error Detection for Python. In 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695545

1 INTRODUCTION

In recent years, Python has seen a notable surge in popularity. According to IEEE Spectrum [32], Python secured the top ranking as the most popular language in 2023. Moreover, Python has consistently outpaced Java and C/C++ in popularity on GitHub since 2019 [41]. The popularity of Python is primarily attributed to its intrinsic flexibility as a dynamically typed language, which reduces development time and facilitates the rapid prototyping of software [20, 53].

However, the flexibility of dynamic languages such as Python comes at a cost — susceptibility to runtime type errors. Type errors, a class of runtime errors arising from the use of inappropriate value types in operations (e.g., "1"+2), stand out as the most prevalent runtime errors in Python, comprising over 30% of all built-in exceptions [42]. As a result, a recent survey [30] shows that static typing, which detects type errors at compile time, is the most-wanted feature among Python developers.

*Corresponding author

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ASE '24, October 27-November 1, 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695545

Existing Tools. Various static type analysis tools have been developed for Python to address this challenge. The most popular ones are Mypy [46], Pytype [17], Pyre [13], and Pyright [37]. Mypy is a widely-used tool built within the Python community. Pyright, Pyre, and Pytype are industrial tools developed by Microsoft, Meta, and Google, respectively. These tools, when provided with a Python program that may include type hints, identify and report type errors by performing static type inference and checking.

However, the performance of these tools remains unsatisfactory in practice. For instance, when we tested these tools on our benchmark, which includes 68 developer-confirmed type errors gathered from 20 open-source programs, Mypy, Pytype, Pyre, and Pyright collectively detected only 16 out of the 68 bugs.

Our Contributions. In this paper, we aim to advance the state-of-the-art in static type analysis tools for Python. To achieve this, we empirically analyzed the 68 bugs to identify the necessary type analysis features for their detection. We first found that type checking with automated type inference is critical because real-world Python programs often lack manual type hints [8]. Consequently, we designed four type-inference features for static analysis to effectively detect real bugs: (1) type preservation when merging, (2) cost-effective interprocedural analysis, (3) usage-based inference of likely types, and (4) inference of intended member types.

Building on these insights, we developed a new type analysis tool, called Pyinder. Evaluation on the 68 bugs demonstrates that Pyinder can detect 34 bugs (113% increase over existing tools combined). Additionally, when applied to 9 latest open-source Python projects, Pyinder successfully found 19 previously-unknown bugs while the existing tools collectively found 13. Finally, we analyze the remaining 34 bugs Pyinder failed to detect from our benchmark, illuminating the future directions toward more practical static type analysis tools for Python.

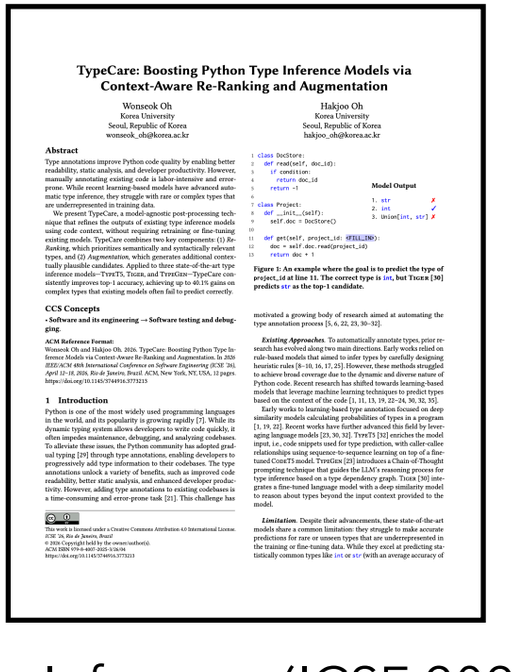
In summary, this paper makes the following contributions:

- We present a dataset of 68 developer-confirmed Python type errors and demonstrate that existing static analysis tools have significant room for improvement.
- We introduce a new type analysis tool, Pyinder, which incorporates four key features identified through our manual investigation of the benchmarks.
- We highlight future research directions for Python type analysis, identifying unresolved issues that need to be addressed to enhance practical type error detection.

2 REAL-WORLD PYTHON TYPE ERRORS

In this paper, a "type error" denotes either a runtime TypeError exception or a type mismatch between type annotations and actual values at runtime. For example, consider the following code snippet:

Type Error Detection (ASE 2024)



TypeCare: Boosting Python Type Inference Models via Context-Aware Re-Ranking and Augmentation

Wonseok Oh
Korea University
Seoul, Republic of Korea
wonseok_oh@korea.ac.kr

Hakjoo Oh
Korea University
Seoul, Republic of Korea
hakjoo_oh@korea.ac.kr

```
class DocStore:
  def read(self, doc_id):
    if condition:
      return doc_id
    return -1

class Project:
  def __init__(self):
    self.doc = DocStore()

def get(self, project_id: <FILL_IN>):
  doc = self.doc.read(project_id)
  return doc + 1
```

Model Output
1. str ✗
2. int ✓
3. Union[int, str] ✗

Figure 1: An example where the goal is to predict the type of project_id at line 11. The correct type is int, but TIGER [30] predicts str as the top-1 candidate.

Abstract

Type annotations improve Python code quality by enabling better readability, static analysis, and developer productivity. However, manually annotating existing code is labor-intensive and error-prone. While recent learning-based models have advanced automatic type inference, they struggle with rare or complex types that are underrepresented in training data.

We present TypeCare, a model-agnostic post-processing technique that refines the outputs of existing type inference models using code context, without requiring retraining or fine-tuning existing models. TypeCare combines two key components: (1) *Re-Ranking*, which prioritizes semantically and syntactically relevant types, and (2) *Augmentation*, which generates additional contextually plausible candidates. Applied to three state-of-the-art type inference models—TypeT5, Tiger, and TypeGen—TypeCare consistently improves top-1 accuracy, achieving up to 40.1% gains on complex types that existing models often fail to predict correctly.

CCS Concepts

• Software and its engineering → Software testing and debugging.

ACM Reference Format:
Wonseok Oh and Hakjoo Oh. 2026. TypeCare: Boosting Python Type Inference Models via Context-Aware Re-Ranking and Augmentation. In 2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773213

1 Introduction

Python is one of the most widely used programming languages in the world, and its popularity is growing rapidly [7]. While its dynamic typing system allows developers to write code quickly, it often impedes maintenance, debugging, and analyzing codebases. To alleviate these issues, the Python community has adopted gradual typing [29] through type annotations, enabling developers to progressively add type information to their codebases. The type annotations unlock a variety of benefits, such as improved code readability, better static analysis, and enhanced developer productivity. However, adding type annotations to existing codebases is a time-consuming and error-prone task [21]. This challenge has motivated a growing body of research aimed at automating the type annotation process [5, 6, 22, 23, 30–32].

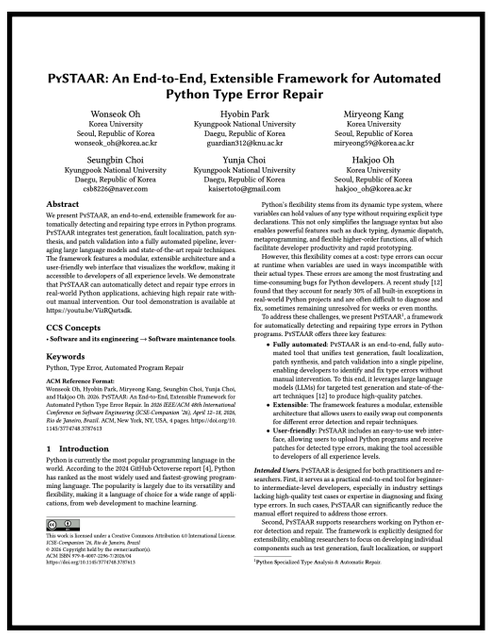
Existing Approaches. To automatically annotate types, prior research has evolved along two main directions. Early works relied on rule-based models that aimed to infer types by carefully designing heuristic rules [8–10, 16, 17, 25]. However, these methods struggled to achieve broad coverage due to the dynamic and diverse nature of Python code. Recent research has shifted towards learning-based models that leverage machine learning techniques to predict types based on the context of the code [1, 11, 13, 19, 22–24, 30, 32, 35].

Early works to learning-based type annotation focused on deep similarity models calculating probabilities of types in a program [1, 19, 22]. Recent works have further advanced this field by leveraging language models [23, 30, 32]. TypeT5 [32] enriches the model input, i.e., code snippets used for type prediction, with caller-callee relationships using sequence-to-sequence learning on top of a fine-tuned CodeT5 model. TypeGen [23] introduces a Chain-of-Thought prompting technique that guides the LLM's reasoning process for type inference based on a type dependency graph. Tiger [30] integrates a fine-tuned language model with a deep similarity model to reason about types beyond the input context provided to the model.

Limitation. Despite their advancements, these state-of-the-art models share a common limitation: they struggle to make accurate predictions for rare or unseen types that are underrepresented in the training or fine-tuning data. While they excel at predicting statistically common types like int or str (with an average accuracy of

This work is licensed under a Creative Commons Attribution 4.0 International License.
ICSE '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773213

Type Inference (ICSE 2026)



PySTAAR: An End-to-End, Extensible Framework for Automated Python Type Error Repair

Wonseok Oh
Korea University
Seoul, Republic of Korea
wonseok_oh@korea.ac.kr

Hyobin Park
Kyungpook National University
Daegu, Republic of Korea
guardian312@knu.ac.kr

Miryeong Kang
Korea University
Seoul, Republic of Korea
miryeong59@korea.ac.kr

Seungbin Choi
Kyungpook National University
Daegu, Republic of Korea
csb8226@naver.com

Yunja Choi
Kyungpook National University
Daegu, Republic of Korea
kaisertoto@gmail.com

Hakjoo Oh
Korea University
Seoul, Republic of Korea
hakjoo_oh@korea.ac.kr

Abstract

We present PySTAAR, an end-to-end, extensible framework for automatically detecting and repairing type errors in Python programs. PySTAAR integrates test generation, fault localization, patch synthesis, and patch validation into a fully automated pipeline, leveraging large language models and state-of-the-art repair techniques. The framework features a modular, extensible architecture and a user-friendly web interface that visualizes the workflow, making it accessible to developers of all experience levels. We demonstrate that PySTAAR can automatically detect and repair type errors in real-world Python applications, achieving high repair rate without manual intervention. Our tool demonstration is available at https://youtu.be/YiiRQutxalk.

CCS Concepts

• **Software and its engineering → Software maintenance tools**.

Keywords

Python, Type Error, Automated Program Repair

ACM Reference Format:
Wonseok Oh, Hyobin Park, Miryeong Kang, Seungbin Choi, Yunja Choi, and Hakjoo Oh. 2026. PySTAAR: An End-to-End, Extensible Framework for Automated Python Type Error Repair. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-Companion '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3774748.3787613

1 Introduction

Python is currently the most popular programming language in the world. According to the 2024 GitHub Octoverse report [4], Python has ranked as the most widely used and fastest-growing programming language. The popularity is largely due to its versatility and flexibility, making it a language of choice for a wide range of applications, from web development to machine learning.

This work is licensed under a Creative Commons Attribution 4.0 International License.
ICSE-Companion '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2296-7/2026/04
https://doi.org/10.1145/3774748.3787613

Python's flexibility stems from its dynamic type system, where variables can hold values of any type without requiring explicit type declarations. This not only simplifies the language syntax but also enables powerful features such as duck typing, dynamic dispatch, metaprogramming, and flexible higher-order functions, all of which facilitate developer productivity and rapid prototyping.

However, this flexibility comes at a cost: type errors can occur at runtime when variables are used in ways incompatible with their actual types. These errors are among the most frustrating and time-consuming bugs for Python developers. A recent study [12] found that they account for nearly 30% of all built-in exceptions in real-world Python projects and are often difficult to diagnose and fix, sometimes remaining unresolved for weeks or even months.

To address these challenges, we present PySTAAR[1], a framework for automatically detecting and repairing type errors in Python programs. PySTAAR offers three key features:

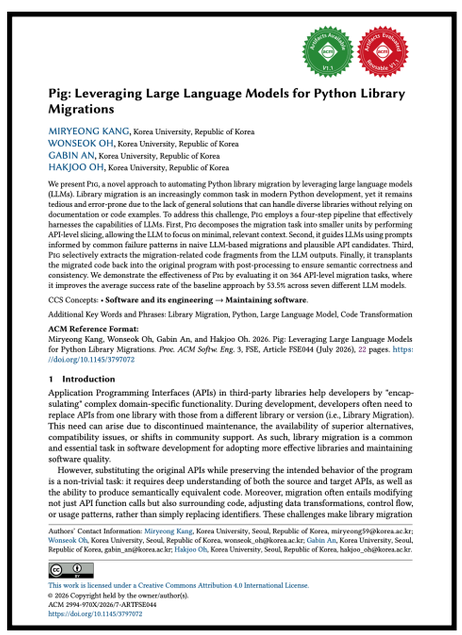
- **Fully automated**: PySTAAR is an end-to-end, fully automated tool that unifies test generation, fault localization, patch synthesis, and patch validation into a single pipeline, enabling developers to identify and fix type errors without manual intervention. To this end, it leverages large language models (LLMs) for targeted test generation and state-of-the-art techniques [12] to produce high-quality patches.
- **Extensible**: The framework features a modular, extensible architecture that allows users to easily swap out components for different error detection and repair techniques.
- **User-friendly**: PySTAAR includes an easy-to-use web interface, allowing users to upload Python programs and receive patches for detected type errors, making the tool accessible to developers of all experience levels.

Intended Users. PySTAAR is designed for both practitioners and researchers. First, it serves as a practical end-to-end tool for beginner- to intermediate-level developers, especially in industry settings lacking high-quality test cases or expertise in diagnosing and fixing type errors. In such cases, PySTAAR can significantly reduce the manual effort required to address those errors.

Second, PySTAAR supports researchers working on Python error detection and repair. The framework is explicitly designed for extensibility, enabling researchers to focus on developing individual components such as test generation, fault localization, or support

[1]Python Specialized Type Analysis & Automatic Repair.

Type Error Detection & Repair (ICSE 2026 Demo)



Pig: Leveraging Large Language Models for Python Library Migrations

MIRYEONG KANG, Korea University, Republic of Korea
WONSEOK OH, Korea University, Republic of Korea
GABIN AN, Korea University, Republic of Korea
HAKJOO OH, Korea University, Republic of Korea

We present Pig, a novel approach to automating Python library migration by leveraging large language models (LLMs). Library migration is an increasingly common task in modern Python development, yet it remains tedious and error-prone due to the lack of general solutions that can handle diverse libraries without relying on documentation or code examples. To address this challenge, Pig employs a four-step pipeline that effectively harnesses the capabilities of LLMs. First, Pig decomposes the migration task into smaller units by performing API-level slicing, allowing the LLM to focus on minimal, relevant context. Second, it guides LLMs using prompts informed by common failure patterns in naive LLM-based migrations and plausible API candidates. Third, Pig selectively extracts the migration-related code fragments from the LLM outputs. Finally, it transplants the migrated code back into the original program with post-processing to ensure semantic correctness and consistency. We demonstrate the effectiveness of Pig by evaluating it on 364 API-level migration tasks, where it improves the average success rate of the baseline approach by 53.5% across seven different LLM models.

CCS Concepts: • **Software and its engineering → Maintaining software**.

Additional Key Words and Phrases: Library Migration, Python, Large Language Model, Code Transformation

ACM Reference Format:
Miryeong Kang, Wonseok Oh, Gabin An, and Hakjoo Oh. 2026. Pig: Leveraging Large Language Models for Python Library Migrations. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE044 (July 2026), 22 pages. https://doi.org/10.1145/3797072

1 Introduction

Application Programming Interfaces (APIs) in third-party libraries help developers by "encapsulating" complex domain-specific functionality. During development, developers often need to replace APIs from one library with those from a different library or version (i.e., Library Migration). This need can arise due to discontinued maintenance, the availability of superior alternatives, compatibility issues, or shifts in community support. As such, library migration is a common and essential task in software development for adopting more effective libraries and maintaining software quality.

However, substituting the original APIs while preserving the intended behavior of the program is a non-trivial task: it requires deep understanding of both the source and target APIs, as well as the ability to produce semantically equivalent code. Moreover, migration often entails modifying not just API function calls but also surrounding code, adjusting data transformations, control flow, or usage patterns, rather than simply replacing identifiers. These challenges make library migration

Authors' Contact Information: Miryeong Kang, Korea University, Seoul, Republic of Korea, miryeong59@korea.ac.kr; Wonseok Oh, Korea University, Seoul, Republic of Korea, wonseok_oh@korea.ac.kr; Gabin An, Korea University, Seoul, Republic of Korea, gabin_an@korea.ac.kr; Hakjoo Oh, Korea University, Seoul, Republic of Korea, hakjoo_oh@korea.ac.kr.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/7-ARTFSE044
https://doi.org/10.1145/3797072

Library Migration (FSE 2026)

...

Need for Python IR

Challenge

- Developing program analysis for Python is challenging.

  (1) **Complex**: Diverse and complex language features, e.g., operator overloading, metaclasses, generators/coroutines.

  (2) **Implicit**: Even seemingly simple operations can trigger intricate dynamic behaviors.

  (3) **Informal**: No formal spec. Tool developers must rely on incomplete docs or examine the CPython implementation.

- Consequence: Building a sound and rigorous program analysis for Python remains difficult.

Example

- Consider a simple expression: **`a + b`**.

(1) Let `A` and `B` denote the types of `a` and `b`, respectively. If `B` is a subclass of `A` and `__radd__` is defined in class `B`, then Python calls `B.__radd__(b, a)`.
(2) If the above conditions do not hold, or if the call returns `NotImplemented`, and `__add__` is defined in `A`, then Python calls `A.__add__(a, b)`.
(3) If `A.__add__` is undefined or returns `NotImplemented`, `__radd__` is defined in `B`, and `__radd__` was not tried in step (1), then `B.__radd__(b, a)` is called (even if `B` is not a subclass of `A`).
(4) If `__radd__` is undefined or returns `NotImplemented`, a `TypeError` is raised.

- Call expression **`f(x,...)`** behaves differently depending on whether **`f`** is a function, a callable object, or neither:

(1) If `f` is a function, the body of the function is executed.
(2) If `f` is not a function and it is not a callable object (the type of `f` does not define `__call__`), a `TypeError` is raised.
(3) If `f` is not a function but defines `__call__`, and `__call__` is a function, then `__call__(f, x, ...)` is evaluated by executing the body of the function.
(4) If `f` is not a function, defines `__call__`, and `__call__` is not a function, then `__call__(f, x, ...)` is evaluated recursively by going back to step 2.

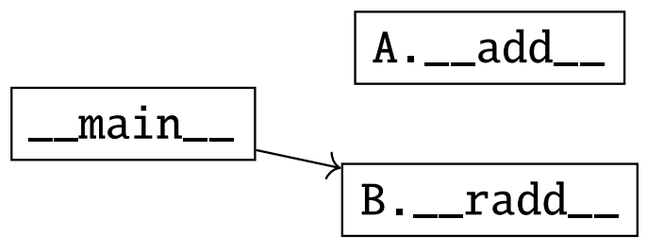
Even analyzing `a+b` involves about a dozen semantic cases

Limitation of Existing Analysis Tools

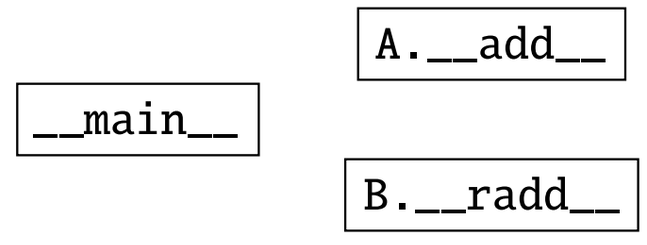
- Most, if not all, existing program analysis tools built on top of Python AST fail to correctly analyze Python programs.

- **Example 1**: Static call graph analyzer (**abstract interpreter)**

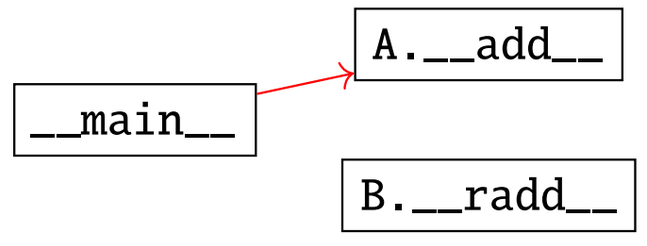
```
1  class A:
2      def __add__(self, other):
3          return self
4
5  class B(A):
6      def __radd__(self, other):
7          return self
8
9  a = A()
10 b = B()
11 a + b # B.__radd__(b,a) is called
```



Ground-truth



PyCG (ICSE 2021*)



Pyright (Microsoft)

* Salis et al. PyCG: Practical Call Graph Generation in Python. ICSE 2021

Limitation of Existing Analysis Tools

- **Example 2**: Building a **deductive verifier** on top of Python AST is notoriously difficult

```
1  a, b = 1, 2
2  b, a = a, b
3  assert a == 1
```



buggy code

Nagini (CAV 2018*)

Verification Succeeded

Google

VeriGuard: Enhancing LLM Agent Safety via Verified Code Generation

Lesly Miculicich[1], **Mihir Parmar**[1], **Hamid Palangi**[1], **Krishnamurthy Dj Dvijotham**[2], **Mirko Montanari**[3], **Tomas Pfister**[1*] **and Long T. Le**[1*]

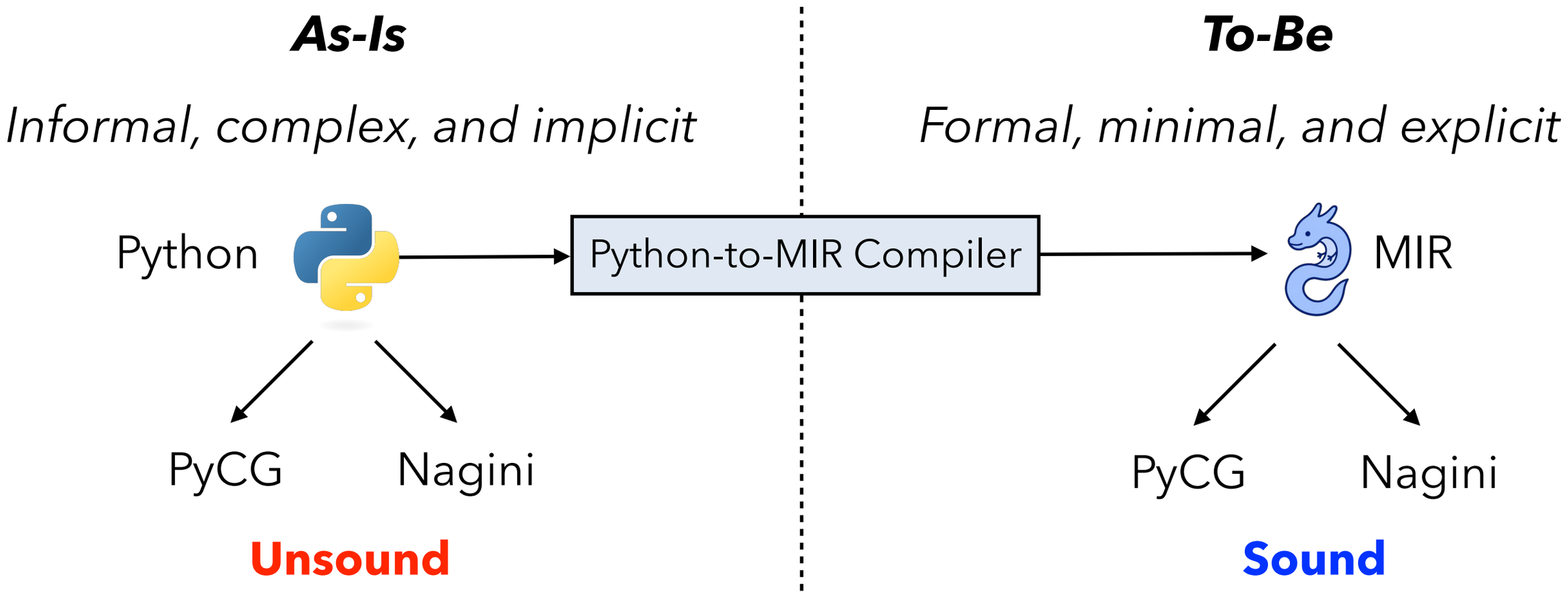
[1]Google Cloud AI Research, [2]Google DeepMind, [3]Google Cloud AI

The deployment of autonomous AI agents in sensitive domains, such as healthcare, introduces critical

* Marc

Goal: Easy and Rigorous Python Analysis

- Our Solution: MIR, a formal, minimal, and explicit IR.
  - **Formal**: Precise, mathematical reasoning is now possible
  - **Minimal**: Frees developers from huge implementation burden
  - **Explicit**: Eliminates subtleties that complicate static reasoning

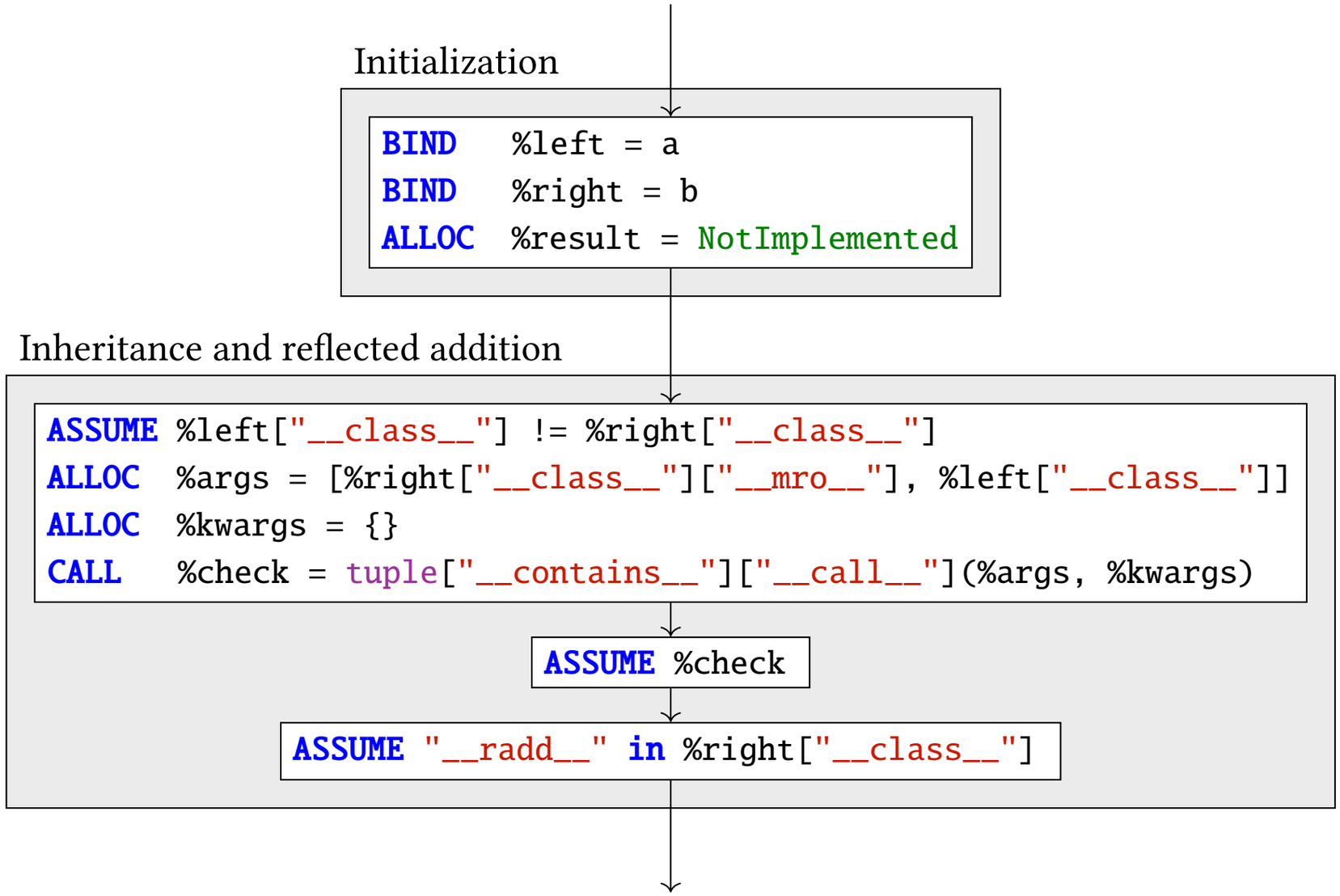


MIR: Syntax

$$
\begin{array}{rcll}
I & \rightarrow & \textbf{skip} & \text{(skip)} \\
 & | & \textbf{assume}(E) & \text{(prune / assume)} \\
 & | & \textbf{alloc}(x, E) & \text{(allocation)} \\
 & | & \textbf{bind}(L_1, L_2) & \text{(binding / aliasing)} \\
 & | & \textbf{env}(x) & \text{(environment)} \\
 & | & \textbf{del}(L) & \text{(delete reference)} \\
 & | & \textbf{call}(x, E, L_1, L_2) & \text{(function call)} \\
L & \rightarrow & x & \text{(variable)} \\
 & | & L[E] & \text{(attribute access)}
\end{array}
$$

Example

- How **`a + b`** is translated into MIR (excerpted):



MIR: Operational Semantics

$$\textsc{I-Skip}\ \frac{}{\ell_{Env}, \sigma \vdash \textbf{skip} \Longrightarrow \sigma} \qquad \textsc{I-Assume}\ \frac{\ell_{Env}, \sigma \vdash E \Downarrow \textbf{true}}{\ell_{Env}, \sigma \vdash \textbf{assume}(E) \Longrightarrow \sigma}$$

$$\textsc{I-Alloc}\ \frac{\ell' \notin Dom(\sigma) \quad \ell_{Env}, \sigma \vdash E \Downarrow \alpha}{\ell_{Env}, \sigma \vdash \textbf{alloc}(x, E) \Longrightarrow \text{EnvUpdate}([\ \ell' \mapsto \alpha\ ]\sigma, \ell_{Env}, x, \ell')}$$

$$\textsc{I-BindVar}\ \frac{\ell_{Env}, \sigma \vdash L \hookrightarrow \ell}{\ell_{Env}, \sigma \vdash \textbf{bind}(x, L) \Longrightarrow \text{EnvUpdate}(\sigma, \ell_{Env}, x, \ell)}$$

$$\textsc{I-BindAttr}\ \frac{\ell_{Env}, \sigma \vdash L_1 \hookrightarrow \ell_1 \quad \ell_{Env}, \sigma \vdash E \Downarrow \pi \quad \ell_{Env}, \sigma \vdash L_2 \hookrightarrow \ell_2}{\ell_{Env}, \sigma \vdash \textbf{bind}(L_1[E], L_2) \Longrightarrow \text{ConUpdate}(\sigma, \ell_1, \pi, \ell_2)}$$

$$\textsc{I-EnvRef}\ \frac{}{\ell_{Env}, \sigma \vdash \textbf{env}(x) \Longrightarrow \text{EnvUpdate}(\sigma, \ell_{Env}, x, \ell_{Env})}$$

$$\textsc{I-DelVar}\ \frac{\sigma(\ell_{Env}) = \rho \quad x \in Dom(\rho)}{\ell_{Env}, \sigma \vdash \textbf{del}(x) \Longrightarrow [\ \ell_{Env} \mapsto \rho \setminus \{x\}\ ]\sigma}$$

$$\textsc{I-DelList}\ \frac{\ell_{Env}, \sigma \vdash L \hookrightarrow \ell \quad \sigma(\ell) = l \quad \ell_{Env}, \sigma \vdash E \Downarrow n \quad n \leq |l|}{\ell_{Env}, \sigma \vdash \textbf{del}(L[E]) \Longrightarrow [\ \ell \mapsto \langle l_1, \ldots, l_{n-1}, l_{n+1}, \ldots, l_{|l|} \rangle\ ]\sigma}$$

$$\textsc{I-DelMap}\ \frac{\ell_{Env}, \sigma \vdash L \hookrightarrow \ell \quad \sigma(\ell) = \alpha \quad \alpha \in \text{Dict} \cup \text{Obj} \cup \text{Env} \quad \ell_{Env}, \sigma \vdash E \Downarrow s \quad s \in Dom(\alpha)}{\ell_{Env}, \sigma \vdash \textbf{del}(L[E]) \Longrightarrow [\ \ell \mapsto \alpha \setminus \{s\}\ ]\sigma}$$

$$\textsc{I-Call}\ \frac{\begin{array}{c}\ell'_{Env} \notin Dom(\sigma) \quad \ell_{Env}, \sigma \vdash E \Downarrow \langle \lambda(a, k, p).(G, z), \ell''_{Env} \rangle \\ \ell_{Env}, \sigma \vdash L_1 \hookrightarrow \ell_1 \quad \ell_{Env}, \sigma \vdash L_2 \hookrightarrow \ell_2 \qquad \ell'_{Env}, [\ \ell'_{Env} \mapsto [\ a \mapsto \ell_1,\ k \mapsto \ell_2,\ p \mapsto \ell''_{Env}\ ]\ ]\sigma \vdash G \rightsquigarrow \sigma'\end{array}}{\ell_{Env}, \sigma \vdash \textbf{call}(x, E, L_1, L_2) \Longrightarrow \text{EnvUpdate}(\sigma', \ell_{Env}, x, \sigma'(\ell'_{Env})(z))}$$

Language Coverage

Feature category	Representative constructs	Mir	λ_π [39]	Ranson et al. [42]
Literals	`1`, `"s"`, `None`	●†	●	▲
Operations	`x+y`, `x==y`, `x is y`	●	●	●
Containers	`[1, 2]`,`{"a": 3}`, `d[k]`	●	●	✗
Variables and binding	`x`, `x=e`, `del x`	●	●	●
Control flow	`if`, `for`, `break`, `continue`	●	●	●
Functions and calls	`def`, `lambda`, `@dec`, `*args`, `**kwargs`	●	●	▲
Iteration and generators	`for`, `yield`, `__next__`	●	●	✗
Objects and classes	`class C(A)`, `obj.x`	●†	●	✗
Exceptions	`raise`, `try`, `except`, `finally`	●	●	✗
Comprehensions	`[x for x in y]`,`{x: y for x, y in z}`	●	●	✗
Imports and modules	`import`, `from m import x`	●†	▲	▲
Scoping declarations	`global`, `nonlocal`	●	●	✗
Context managers	`with`	●	●	✗
Reflective/dynamic features	`getattr`, `setattr`, `locals`	●	✗	✗
Asynchronous constructs	`async def`, `await`	✗	✗	✗

Instance Analyses

- **MirCG**: A static call-graph analyzer for Python
  - Re-implementation of PyCG atop MIR
- **MirCheck**: A deductive verifier for Python
  - Re-implementation of Nagini atop MIR
- Soundness formally proved in Lean

Theorem (MirCG soundness).

Let $C_*(P)$ be the least solution of the abstract constraints. For every finite MIR execution prefix τ from the initial state of P,

$$\mathrm{Calls}(\tau) \subseteq C_*(P).$$

Theorem (MirCheck soundness).

If the generated VCs are valid, every terminating MIR execution from a state satisfying S with valid permissions admits a final ghost mask for which T and permission validity hold:

$$\models \mathrm{VC}_K(P, S, T) \implies \models_{\mathrm{par}} \{\mathrm{Valid}_K \wedge S\}\ P\ \{\mathrm{Valid}_K \wedge T\}.$$

Evaluation

- Research questions
  - **Fidelity**: Does MIR faithfully capture Python's semantics?
    - Differential testing against CPython
  - **Utility**: Is MIR effective for building program analyses?
    - **MirCG**: Re-implementation of **PyCG** atop MIR
    - **MirCheck**: Re-implementation of **Nagini** atop MIR

Fidelity

Benchmark	Total	Applicable	Compiled	Executed	Correct
Mir benchmark	266	266	266	266	266
MBPP	974	619	619	619	619
HumanEval	163	116	116	116	116
Total	1,403	1,001	1,001	1,001	1,001

- **MIR benchmarks**: 266 Python programs covering various features—especially corner cases
- **MBPP and HumanEval**: 735 programs after filtering out the programs that use built-in functions that our MIR interpreter does not currently support

Utility: MirCG

- Soundness and completeness

		MirCG			PyCG [45]		
Benchmark	Total	Sound	Complete	Time (s)	Sound	Complete	Time (s)
MirCG benchmark	100	**87**	77	5.2	15	77	4.9
Micro-benchmark	112	**107**	106	6.7	46	106	5.9
Total	212	**194 (91.5%)**	183 (86.3%)	11.9	61 (28.8%)	183 (86.3%)	10.8

- Analysis cost

		MirCG		PyCG [45]
Project	LoC	Compile (s)	Analyze (s)	Analyze (s)
asciinema	1,553	4.5	**1.3**	0.3
autojump	3,287	6.3	**3.0**	0.2
fabric	2,888	3.2	**1.0**	0.3
face_classification	1,895	3.8	**2.6**	0.3
Sublist3r	1,672	4.2	**2.3**	0.2
Average	2,259	4.4	**2.0**	0.3

Utility: MirCheck

- Soundness and completeness

		MirCheck				Nagini [7]			
Spec. Type	Total	Accept	Reject	Inconc.	Time (s)	Accept	Reject	Inconc.	Time (s)
Correct	25	25	0	0	4.4	16	**5**	4	4.5
Incorrect	25	0	25	0	4.4	**4**	17	4	4.5

- Verification cost

				MirCheck		Nagini [7]	
ID	Description	LoC	Spec.	Verified	Time (s)	Verified	Time (s)
005	Insert delimiter between elements	10	13	✓	5.3	✓	4.9
023	Return string length	2	1	✓	4.3	✓	4.3
027	Swap character case	20	17	✓	7.7	✓	6.7
041	Count car collisions	2	2	✓	4.4	✓	4.4
046	Compute Fib4 number	26	9	✓	4.9	✓	5.0
051	Remove vowels	9	18	✓	9.6	✓	7.9
119	Match concatenated parentheses	26	43	✓	10.0	✓	13.2
127	Check prime intersection length	21	6	✓	4.6	✓	4.7
134	Check final standalone letter	5	4	✓	4.7	✓	4.6
139	Compute special factorial	19	13	✓	4.9	✓	4.9

Summary

- 연구 분야 소개: 프로그램 분석 및 오류 자동 수정 기술
- 최신 연구 소개: 엄밀한 파이썬 프로그램 분석 및 검증이 이제 가능

감사합니다!