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 [52], 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

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:

```python
a = 1
b = '2'

# TypeError
a + b
```

(a) Internal exception

```python
import numpy as np

v = [1, None]
# TypeError
w = np.sort(v)
```

(b) External exception

```python
a = int_or_str()

if isinstance(a, int):
  # TypeError
  raise TypeError
```

(c) User-defined exception

Cython

```
def add(x, y): # TypeError
  cdef int ret = x + y
```

Python

```python
a.add('1', '2')
```

(d) Cython exception

Figure 1: Typical examples of runtime type errors in Python

```python
1 def add(x: int, y: int):
2     return x + y # a TypeError exception
3 add('1', 2) # a type mismatch error
```

A type-error exception occurs at line 2 due to the invalid operation of adding a string and an integer. Moreover, the program also contains a type mismatch error at line 3 because a string value is passed to a function that expects an integer value according to its type annotation. We refer to both cases as type errors, and most static type analysis tools support detecting both of these errors.

To investigate real-world type errors in Python, we analyzed 145 runtime errors collected from the three benchmarks, TypeBugs [42], BugsInPy [58], and ExcePy [62], which consist of actively developed open-source projects supporting Python 3.5 or above. We found that those 145 type errors can be classified into four categories:

- Internal errors (61): Exceptions or mismatches whose error locations are within the current Python project.
- External errors (33): Type errors whose error locations are in external libraries.
- User-defined errors (35): Custom type errors explicitly raised by developers (i.e., `raise TypeError(...)`).
- Cython errors (16): Type errors raised in the Cython language that allows C extensions for Python.

Figure 1 illustrates typical examples of these errors.

Among them, our study focuses on the internal type errors and some of the external errors. We focus on internal errors because they are the most prevalent yet amenable to detection by static analysis tools. External and Cython errors are beyond the scope of static type analysis tools as they occur in library code or source code written in a foreign language. For example, static analysis tools cannot detect the type errors in Figure 1b and Figure 1d because the source code of the `numpy` library is unavailable, and they cannot analyze programs written in the Cython language. However, 7 out of 33 external errors could be identified by leveraging the signature information from the typeshed project [47]. Thus, we included those seven errors in our benchmark. In addition, since user-defined type errors are ignored by existing tools (e.g., Figure 1c), we also excluded them from our study.

In summary, we gathered 68 real-world type errors in Python from 20 open-source projects ranging from 3k to 417k lines.

3 OVERVIEW OF PYINDER

We analyzed the 68 bugs to identify static analysis features that could effectively detect them. We describe these features, which have been integrated into our tool Pyinder, along with examples. We note that all type annotations in this section below are added for explanation purposes. These annotations are not present in the original code; instead, Pyinder automatically infers them from other parts of the programs.

Type Analysis. Before introducing the features, we first describe how type analysis is performed in Pyinder and existing tools. Type analysis consists of two steps: type inference and type checking. The goal of type inference is to determine the types of variables at each point in the program, while type checking interprets the program based on the inferred types to identify potential type errors. For example, consider the following code snippet:

```python
1 x, y = '1', any() # any() returns any type
2 x+1               # x: {str}
3 y+1               # y: {Any}
```

In the type inference phase, Pyinder infers the types of variables `x` and `y` as the `str` and `Any` types, respectively, at line 1. Next, in the type checking phase, Pyinder detects type errors based on the inferred types. For instance, Pyinder discovers that the expression `x+1` at line 2 causes a type error. At line 3, however, Pyinder does not report any issue because the inferred type (`Any`) is imprecise. This design choice aims to avoid too many false alarms and focuses on reporting type errors with sufficient evidence. It is worth noting that all existing type analysis tools [13, 17, 37, 46] follow this practice. Additionally, like existing tools, Pyinder performs type analysis on each method entry rather than assuming a main method.

While the type checking phase is the same for all tools, as they are based on the same Python type system, the design choices for the type inference phase differ significantly between tools. Therefore, in this paper, we focus on enhancing type inference for more effective type error detection. Below, we introduce the key features of Pyinder.

3.1 Preserving Types when Merging

First of all, Pyinder merges types carefully. Specifically, it avoids merging types into the `Any` type when types are combined at control flow merge points, within collection types, and in dictionaries.

Control Flows Merge Points. Preserving type information at control flow merge points was important. Among the 68 type errors, detecting 29 type errors required this feature. Consider the example:

Listing 1: Simplified from keras-39

```python
1 def update(target: Any):
2     if target is None:
3         info = 'one'
4     if target < 0: ... # TypeError: None < int
```

A type error at line 4 occurs because the variable `target` can be `None` when taking the true branch at line 2. Thus, we preserve the `None` type of `target` at the merge point right before line 4 rather than merging it with the `Any` type in the false branch, i.e., inferring the type of `target` as `{Any,None}` rather than `{Any}`. Merging `None` (true-branch) and `Any` (false-branch) into `{Any}` would miss the type error at line 4 because we do not report issues involving `Any`.

Collections. Distinguishing types in collection data types was also important, which was required to detect 20 type errors in our benchmark. Consider the example:

Listing 2: Simplified from salt-56381

```
1 ret = [1, '2']
2 msg = ret[0] + 'msg' # TypeError: int + str
```

A type error occurs at line 2 because a type of `ret[0]` is of the `int` type. To detect the type error, we should infer that the element type of `ret` can be `int`. Therefore, instead of merging the two differing types, `int` and `str`, into `Any`, inferring the type of `ret` as `List[Any]`, we need to distinguish the two cases, inferring the type of `ret` as `List[{int, str}]`.

Dictionaries. Among collection types, special care was required for dictionaries in order to accurately detect bugs. Consider the example below:

```
1 x = {1: 1, '2': '2'} # dict[{int,str}, {int,str}]
2 # x[1] is inferred as {int,str}
3 x[1] + "1" # TypeError: int + str
4 x[1] + 1   # False Alarm: str + int
```

Inferring a type of `x` as `dict[{int,str}`$\longrightarrow$`{int,str}]` successfully detects a type error at line 3. However, it also generates a false alarm at line 4 because `x[1]` is inferred as `{int,str}` instead of `{int}`. Thus, Pyinder abstracts dictionary types in a key-type sensitive way, inferring the type of `x` as `dict[int` $\longrightarrow$ `int, str` $\longrightarrow$ `str]`. With this refinement, the type of `x[1]` at line 4 is inferred as `int` because the key type in this case is `int`, which removes the false alarm. Without this feature, we found that 35% more (false) alarms were generated to detect the same number of bugs.

3.2 Cost-Effective Interprocedural Analysis

Pyinder cost-effectively supports interprocedural analysis. In total, 37 type errors required tracking interprocedural value flows (parameter passing and return flows) to detect them. However, it is well-known that interprocedural analysis is challenging as it blows up the analysis cost. We investigated the benchmarks to identify the right feature for Python type analysis.

Shallow Call Depths. We found that tracking shallow call depth, especially up to 3, is sufficient to detect most of the type errors, which reduced the cost of interprocedural analysis by 78% compared to the conventional analysis supporting unbounded call depths. In order to determine the most effective call depth, we analyzed how many call depths are required to detect the type errors in our benchmark. Figure 2 shows that 31 type errors can be found within call depths of $\leq$3. In contrast, three type errors required call depths beyond 3 but trying to detect them increased the analysis cost by 2.1 times.

Lightweight Type-based Context Sensitivity. We found that context sensitivity is essential but a lightweight approach based on types is sufficient. To detect the type errors in our benchmarks, analyzing called methods separately for their call contexts was important; otherwise, a context-insensitive analysis increased the cost of analysis by 1.4 times. At the same time, because the goal of static analysis is to analyze types of variables, using types as call contexts (rather than using invocation sites or receiver objects as contexts in conventional call-site-sensitive or object-sensitive analyses) was sufficient to achieve the desired precision. Thus, Pyinder applies a context-sensitive analysis that uses argument types as contexts.

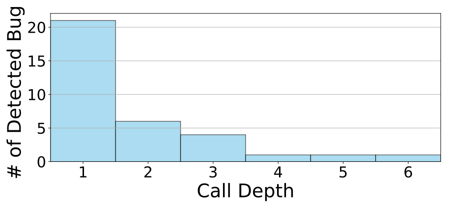

Figure 2: Call depth required to detect type errors.

3.3 Usage-based Inference of Likely Types

Pyinder infers likely types of function parameters by observing how they are used in the body. Consider the following example:

Listing 3: Simplified from pandas-24572

```
1 class Multi:
2     def format(self, adjoin=None) -> tuple: ...
3     def msg(self): -> list: ...
4
5 def write(columns): # columns: Multi, but not annotated
6     form = columns.format(adjoin=False)
7     list_obj = columns.msg() + [" End"]
8     return ["Start "] + form # TypeError: list + tuple
```

At line 5, a parameter `columns` is an instance of the `Multi` class, but its type is not annotated and remains unknown in the program. However, we can infer that its type is `Multi` by observing how it is used at lines 6 and 7 because the two methods, `format` and `msg`, of class `Multi` are invoked on the object that `columns` points to. With this information, we can predict the type of `form` at line 6 as `tuple` from the type annotation of `format` at line 2, which enables us to detect a type error at line 8.

However, we found that this simple usage-based type inference could be ineffective in practice as it often infers infeasible types. To mitigate this issue, Pyinder selects the most likely type with the highest score. In a nutshell, we assign higher scores when usage patterns and signatures are similar. For example, assume that the following `Index` class also exists in the previous example shown in Listing 3.

```
1 class Index:
2     def format(self, **kwargs) -> list: ...
3     def msg(self) -> tuple: ...
```

Note that `Index` also supports methods `format` and `msg`, and therefore the type of `columns` in Listing 3 is inferred as `{Multi,Index}`, causing a false alarm at line 7 because the `msg` method of `Index` returns a value of the `tuple` type. Thus, Pyinder considers `Multi` to be the most likely type because, when calling the `format` method at line 6, `adjoin` is used as a keyword argument. Note that the `format` method of `Multi` has this keyword argument according to its signature, whereas the `format` method of `Index` assumes a more general parameter. Based on this, we conclude that the type of `columns` at line 6 is more likely `Multi` and only use that type (rather than using multiple types `{Multi,Index}`) when performing type checking at the subsequent lines of method `write`.

We observed that choosing the most likely type instead of using all possible types via simple usage-based inference is 2.7 times more efficient in terms of analysis time and produces 10% fewer alarms, without affecting the type-error detection capability of Pyinder.

3.4 Inference of Intended Member Types

We also found that accurately inferring intended class member types by considering feasible method call sequences in the class is critical to detect bugs while avoiding false alarms.

To do so, Pyinder first considers all possible method call scenarios to detect bugs. For example, consider the following class definition:

Listing 4: Simplified from core-29829

```
1 class InputText():
2     def __init__(self, low: Any): self.low = low
3     def set_low(self, low=None):  self.low = low
4     def set_value(self, v: int):
5         if v < self.low: ... # TypeError: int < None
```

In this example, when the method `set_value` is called after `set_low` (e.g., `set_low()` ⟶ `set_value(1)`), a type of class member variable `self.low` can be `None`, causing a type error at line 5. To detect such errors, which accounted for 43% of the type errors in our benchmark, Pyinder infers class member types by considering all the possible scenarios of calling class methods and computing class invariants in terms of member types (e.g., "The type of `self.low` can be `None` in class `InputText`").

However, we found that type checking based on the types derived from these class invariants generates numerous false alarms because of infeasible method call scenarios that developers do not intend. To mitigate this issue, Pyinder filters out unintended types of class members, reducing false alarms by 60% without compromising type error detection. Consider the following example:

Listing 5: Simplified from pandas-38431

```
1  class Parser():
2      def __init__(self):            self.n = None
3      def setup(self, n: list[str]): self.n = n
4
5      def return_data(self, flag, col):
6          if flag: return self.n[col] + "?" # False Alarm
7          else:    return self.n[col] + "!" # False Alarm
8
9      def set_data(self, flag, col):
10         if flag: self.n[col] = "empty"    # False Alarm
11         else:    self.n[col] = "filled"   # False Alarm
12
13 def get_parser():
14     parser = Parser()
15     parser.setup(["name"])
16     parser.return_data(True, "name")
17     parser.set_data(True, 0)
```

In class `Parser`, methods `return_data` and `set_data` methods are intended to be used only after the `setup` method has been called, as exemplified in the `get_parser` function at lines 13–17. With this intended usage, no type errors occur in this example. However, inferring member types by considering all possible scenarios causes false type-error alarms at lines 6, 7, 10, and 11 because the type of `self.n` is inferred as `{None,list[str]}`, where `None` comes from `__init__` and `list[str]` from `setup`. Although these type errors are possible following call sequences such as `__init__` ⟶ `return_data` or `set_data`, these scenarios are considered infeasible by the developer. Therefore, the developer does not consider the type errors at lines 6, 7, 10 and 11 to be real bugs. To avoid these alarms, we need to identify that scenarios such as `__init__` ⟶ `return_data` or `set_data` are not intended, while `setup` ⟶ `return_data` or `set_data` are intended by the developer in this class definition.

From the benchmark study, we observed that many unintended method-call scenarios share three characteristics: (1) the scenarios do not explicitly appear in the given program, (2) following those scenarios, (false) type-error alarms with the same cause are repeatedly raised in multiple locations, and (3) there exist other method-call scenarios that can eliminate those type-error alarms. When a scenario meets these conditions, Pyinder considers it to be unintended and does not report alarms related to the scenario. For example, Pyinder excludes the call sequence `__init__` ⟶ `return_data` or `set_data` from consideration because (1) the sequence is not explicitly used by the developer in the program, (2) type errors are repeatedly generated at lines 6, 7, 10, and 11 due to the sequence, and (3) there exists an alternative call sequence, e.g., `__init__` ⟶ `setup` ⟶ `return_data` or `set_data`, that can eliminate those alarms. Thus, we filter out the unintended type (`None`) of `self.n` when analyzing methods `return_data` and `set_data`, removing type error alarms at lines 6, 7, 10, and 11.

3.5 Novelty of Pyinder's Features

Comparison with Existing Tools. Our four features described so far are mostly novel compared to those implemented in existing type analysis tools for Python [13, 17, 37, 46]. Existing tools only partially support these features, as detailed in Table 1; in existing tools, the features are enabled without addressing their negative impacts, such as increased false alarms. Supporting a feature in Sections 3.1–3.4 without consideration not only improves bug detection but also negatively affects the analysis in terms of increased false positives or analysis costs. The challenge is therefore how to support such a feature cost-effectively in a way that minimizes these negative side-effects. The main novelty of Pyinder is to provide practical guidelines for designing type analysis tools for Python, aiming at supporting those features while minimizing their side-effects.

Comparison with Existing Static Analyses. Our guidelines for achieving cost-effective type analysis for Python are also novel within the broader context of static analysis. Balancing soundness, precision, and scalability is a central challenge in static analysis, and numerous approaches have been proposed over the past decades. In particular, our features align with approaches that selectively apply expensive analysis techniques (e.g., context-sensitive interprocedural analysis) only when beneficial, while minimizing negative impacts [24, 27, 28, 31, 34–36, 51, 54]. Developing such selective static analysis inevitably requires language and problem-specific features [24, 27, 34, 36], and our work can be seen as a proposal for such features tailored specifically to Python type inference. Note that existing works on selective static analysis have primarily targeted other languages like C and Java, making their features not applicable to Python. Furthermore, existing techniques for Python type inference [5, 18, 21, 50] have focused on uniform approaches rather than selective ones. In Section 6, we discuss existing techniques in more detail.

Table 1: Comparison with existing tools. P1 through P9 indicate detailed points of the features in Section 3.

Feature	Detailed points for supporting the feature		Mypy	Pyre	Pytype	Pyright	Pyinder
Section 3.1	**P1**	Preserving types at control flow merge points	✕	✕	✕	○	○
	P2	Preserving element types in collections	✕	✕	○	○	○
	P3	Considering dictionaries with a key-type-sensitive manner	✕	✕	✕	✕	○
Section 3.2	**P4**	Shallow call depths (up to 3)	✕	✕	✕	✕	○
	P5	Lightweight type-based context sensitivity	✕	✕	○	✕	○
Section 3.3	**P6**	Usage-based type inference	✕	✕	✕	✕	○
	P7	Selecting likely types among usage-based types	✕	✕	✕	✕	○
Section 3.4	**P8**	Considering class invariants in terms of member types	✕	✕	✕	○	○
	P9	Filtering out unintended member types	✕	✕	✕	✕	○

4 PYINDER IN DETAIL

In this section, we formally present Pyinder. The goal is to precisely describe the features informally explained in Section 3.

Program. A program P comprises a sequence of class declarations $C^* \subseteq CDecl$. A class declaration $C \in CDecl$ is a tuple (c, M^*) of class name $c \in ClsType$ and a list M^* of method declarations. A method declaration $M \in MDecl$ is a tuple (m, p, S, ret) of a method name $m \in Mthd$, a parameter $p \in Var$, a body statement $S \in Stmt$, and a return variable $ret \in Var$. We write p_m for the parameter p of the method m. For formalization, we consider the following sets of statements and expressions:

$$S \to S_1; S_2 \mid \text{if } b\ S_1\ S_2 \mid x := E \mid x.y := E \mid x[n] := E \mid x\{y\} := E$$
$$E \to n \mid s \mid [\,] \mid \{\} \mid x \mid x.y \mid x[n] \mid x\{y\} \mid x.m(y) \mid E_1 \oplus E_2 \mid c()$$

where n, s, $[\,]$, and $\{\}$ are integer, string, list, and dictionary constants, respectively. We write $x[n]$ and $x\{y\}$ for item access of list and dictionary, respectively, and $c()$ for an instantiation of class c. We assume that a boolean expression is simply either true or false (because type analysis typically does not track conditional values precisely). A variable can be either a local or `self`. We assume that variables and method names are unique and type annotations are not provided in the program.

Type Errors. For simplicity, we focus on type errors that occur within class members. We define type errors as follows:

$$\epsilon \in TypeError = \mathcal{P}(Cause) \times Loc$$
$$o \in Cause = (ClsType \times Var) \times \mathcal{P}(Type)$$

A type error ϵ is a pair of causes ($O \in \mathcal{P}(Cause)$) and error location (Loc), and a cause $o = ((c, x), T)$ means that the type error occurs due to member variable $x \in Var$ of class $c \in ClsType$ and the involved types are $T \in \mathcal{P}(Type)$. Consider the example:

```
1 class A:
2     def f(self):
3         self.x, self.y = 1, 'two'
4         self.x + self.y # TypeError: int + str
```

An error at line 4 is represented by $(\{((\text{A}, \text{x}), \{\texttt{int}\}), ((\text{A}, \text{y}), \{\texttt{str}\})\}, 4)$ which means that the error occurs because variables x and y of class A have types int and str, respectively.

Type Checking. We use a conventional type checking routine:

$$TypeCheck : Pgm \times Summary \to \mathcal{P}(TypeError)$$

which takes a program and a type summary as input, and produces as output a set of type errors. A type summary $\alpha \in Summary : Loc \to \mathcal{P}(State)$ is a map from program locations to states, and denotes the result of static type inference. Given P and α, type checking is done as follows:

$$TypeCheck(P, \alpha) = \bigcup_{l \in lines(P)} Check(l, \alpha(l))$$

where $lines(P)$ denotes the set of all lines of the program P and $Check(l, \alpha(l))$ executes the instruction at line l on the states in $\alpha(l)$ and collects potential type errors.

For Pyinder to be effective, the main challenge is how to efficiently generate useful type summary α. From now on, we explain the type inference algorithm of Pyinder.

4.1 Basic Type Inference Algorithm

Pyinder uses the following abstract domain for static type inference:

$$\begin{array}{lll} d \in TypeTable & = & Mthd \to State \\ s \in State & = & TypeEnv \times MemberEnv \\ \Gamma \in TypeEnv & = & Var \to \mathcal{P}(Type) \\ \Sigma \in MemberEnv & = & ClsType \times Var \to \mathcal{P}(Type) \\ t \in Type & = & \{\texttt{int}, \texttt{str}, \texttt{Any}\} + Collection + ClsType \\ \kappa \in List & = & Type^* \\ \gamma \in Dict & = & Type \to \mathcal{P}(Type) \end{array}$$

The goal of type inference is to compute a type table d which maps methods to states. A state s consists of a type environment Γ and a member environment Σ. A type environment Γ maps local variables of methods to their types, and a member environment Σ stores the types of class member variables. A type is *Int*, *Str*, *Any*, *Collection*, or *ClsType*. As explained in Section 3.1, we distinguish collection types with special care for dictionaries. That is, we distinguish types of list elements without merging and abstract dictionary types in a key-type-sensitive way. We denote the merge operator for the list by $\uplus : List \times List \to List$ and write $list : \mathcal{P}(Type) \to List$ for the function that converts a set of types to a list.

Pyinder performs type inference by computing the least fixpoint

$$d^* = \lim_{n \to \infty} F^n(d_{\text{init}})$$

of a semantic function $F : TypeTable \to TypeTable$, where $d_{\text{init}} = \lambda m.(\emptyset, \emptyset)$ denotes the initial type table that maps all methods to the empty state. The semantic function F defined below updates a given type table d by analyzing each method in the program P:

$$F(d) = \bigsqcup_{(c, M^*) \in P} \bigsqcup_{(m, p, S, \_) \in M^*} [m \mapsto \widehat{[\![S]\!]}_k(\Gamma_{\text{init}}, \Sigma_{\text{init}})]$$

where the initial type and member environments for m are:

$$\Gamma_{\text{init}} = [\text{self} \mapsto \{c\}, p \mapsto \{\texttt{Any}\}] \qquad \Sigma_{\text{init}} = \bigsqcup_{(\_, \Sigma) \in range(d)} \Sigma \quad (1)$$

That is, at the start of the analysis, we assume that the parameter of m has type Any. For member environment, to consider all possible method call scenarios in a class, we collect all member variable types observed so far by merging member environments in the current type table d. We note that the initial type environment Γ_{init} is enhanced by the usage-based type inference in Section 4.2, and the unintended types from the initial member environment Σ_{init} are filtered out by the technique in Section 4.3.

The semantic function $\widehat{[\![S]\!]}_k : \mathit{State} \to \mathit{State}$ for statement S is defined as follows (k: call depth limit in Section 3.2, initially 3):

$$\begin{aligned}
\widehat{[\![S_1;S_2]\!]}_k(s) &= \widehat{[\![S_2]\!]}_k(\widehat{[\![S_1]\!]}_k(s))\\
\widehat{[\![\text{if } b\ S_1\ S_2]\!]}_k(s) &= \widehat{[\![S_1]\!]}_k(s) \sqcup \widehat{[\![S_2]\!]}_k(s)\\
\widehat{[\![x := E]\!]}_k((\Gamma,\Sigma)) &= \Gamma[x \mapsto T], \Sigma'\\
\widehat{[\![x.y := E]\!]}_k((\Gamma,\Sigma)) &= \Gamma, \Sigma'[\forall c \in \Gamma(x), (c,y) \mapsto T \cup \Sigma'((c,y))]\\
\widehat{[\![x\{y\} := E]\!]}_k((\Gamma,\Sigma)) &= \Gamma[x \mapsto \{\text{DSet}(\gamma, \Gamma(y), T) \mid \gamma \in \Gamma(x)\}], \Sigma'\\
\widehat{[\![x[n] := E]\!]}_k((\Gamma,\Sigma)) &= \Gamma[x \mapsto \{\text{LSet}(\kappa, T) \mid \kappa \in \Gamma(x)\}], \Sigma'
\end{aligned}$$

where type set T and member environment Σ' are the results of typing rule $\Gamma, \Sigma \vdash_k E : T, \Sigma'$ shown in Figure 3. Functions DSet and LSet are defined to set a value for each collection type as follows:

$$\text{DSet}(\gamma, T_y, T) = \gamma[\forall t_y \in T_y, t_y \mapsto T] \quad \text{LSet}(\kappa, T) = \kappa \uplus \mathit{list}(T)$$

Note that the join operator $\sqcup$ used at the if statement merges states by taking the union of the types observed in both branches to preserve types without merging them into Any.

In the typing rules, function Call is responsible for performing our interprocedural analysis explained in Section 3.2:

$$\text{Call}(\Sigma, m, T_y, k) = \begin{cases} \Gamma_m(\mathit{ret}), \Sigma_m & \text{if } k > 0 \\ \{\mathit{Any}\}, \Sigma & \text{otherwise} \end{cases} \tag{2}$$

where $(\Gamma_m, \Sigma_m) = \widehat{[\![\text{body}(m)]\!]}_{k-1}(\Gamma[p_m \mapsto T_y], \Sigma)$. The parameter k denotes the call depth limit; if $k \leq 0$, we stop analyzing the method call. Furthermore, note that we perform type-based context sensitivity by setting the parameter p_m to the type T_y given at the call-site instead of assigning all observed types to the parameter (context-insensitive analysis).

Summary Generation. Once a fixed point d^* is computed, we generate a summary α through a procedure $\text{Gen} : \mathit{Pgm} \times \mathit{TypeTable} \to \mathit{Summary}$ defined below:

$$\text{Gen}(P, d^*) = \bigsqcup_{(c,M^*)\in P} \bigsqcup_{(m,\_,S,\_)\in M^*} G_k(S, (\Gamma_{\text{init}}, \Sigma^*)) \tag{3}$$

where Σ^* is the member environment merged from the type table d^* ($\Sigma^* = \bigsqcup_{(\_,\Sigma)\in \mathit{range}(d^*)} \Sigma$). Function $G_k : \mathit{Stmt} \times \mathit{State} \to \mathit{Summary}$ records states for each line:

$$G_k(S,s) = \begin{cases} G_k(S_1, s) \sqcup G_k(S_2, \widehat{[\![S_1]\!]}_k(s)) & \cdots S = S_1; S_2 \\ G_k(S_1, s) \sqcup G_k(S_2, s) & \cdots S = \text{if } b\ S_1\ S_2 \\ [\mathit{line}(S) \mapsto \{s\}] \sqcup G_{k-1}(\text{body}(m), s') & \cdots S \text{ contains } x.m(y) \\ [\mathit{line}(S) \mapsto \{s\}] & \cdots \text{otherwise} \end{cases}$$

If the statement S contains a method call $x.m(y)$, we generate a summary for the method through $G_{k-1}(\text{body}(m), s')$ where s' is a state $(\Gamma[p_m \mapsto T_y], \Sigma)$ given in Eq (2). We also set the initial k as 3 for the summary generation.

The type inference procedure described so far includes the first two features in Section 3: (1) type preservation when merging and (2) cost-effective interprocedural analysis. From now on, we formalize the remaining features.

4.2 Usage-based Likely Type Inference

We first define the usage-based type inference in Section 3.3. To describe this algorithm, we extend the language for parameters and arguments. At first, we assume that a method declaration and a call expression can have multiple parameters and arguments. In addition, a parameter can have a default value, which is called a default parameter, i.e. `def f(x=1)`, and an argument also has two types: positional arguments $x.m(x)$ and keyword arguments $x.m(y = 1)$. We assume that method names may not be unique in this section.

Our goal is to update Γ_{init} in Eq (1) for unannotated parameter p through usage-based type inference that infers the most likely type of a parameter as follows:

$$\Gamma_{\text{init}}[p \mapsto \{\mathsf{MostLikely}(p, \phi_u, \phi_d)\}]$$

where ϕ_u is a map from a parameter to a set of associated members of the parameter ($p.y$ or $p.m(y)$), and ϕ_d is a map from a class type to a set of defined members in the class as follows:

$$\begin{aligned}
\phi_u \in &\quad \mathit{UsedAttrs} &= &\quad \mathit{Var} \to \mathcal{P}(\mathit{Member})\\
\phi_d \in &\quad \mathit{DefinedMem} &= &\quad \mathit{ClsType} \to \mathcal{P}(\mathit{Member})\\
\mu \in &\quad \mathit{Member} &= &\quad \mathit{Var} + (\mathit{Mthd} \times \mathcal{P}(\mathit{Var}) \times \mathcal{P}(\mathit{Var}))
\end{aligned}$$

where a member $\mu \in \mathit{Member}$ is either a field or a method as a tuple of a method name, positional arguments, and keyword arguments (or parameters without and with default values).

We collect ϕ_u and ϕ_d by syntactically analyzing how parameters and class members are used in the program. In particular, ϕ_u is constructed for each method for parameters, and we design ϕ_d so that it includes class information in both the program and typeshed [47], which is a repository of type hints for the Python standard library. For example, in the case of Listing 3 in Section 3.3, ϕ_u and ϕ_d are defined as follows:

$$\begin{aligned}
\phi_u &= [\text{columns} \mapsto \{(\text{format}, \emptyset, \{\text{adjoin}\}), (\text{msg}, \emptyset, \emptyset)\}]\\
\phi_d &= [\text{Multi} \mapsto \{(\text{format}, \emptyset, \{\text{adjoin}\}), (\text{msg}, \emptyset, \emptyset)\}]
\end{aligned}$$

where we omit the `Index` class.

The function $\mathsf{MostLikely} : \mathit{Var} \times \mathit{UsedAttrs} \times \mathit{DefinedMem} \to \mathit{Type}$ returns the most likely type t of p as follows:

$$\mathsf{MostLikely}(p, \phi_u, \phi_d) = \operatorname*{argmax}_{c \in \mathsf{Pos}(p,\phi_u,\phi_d)} \mathsf{Sim}(\phi_u(p), \phi_d(c))$$

where Pos is a function that returns all possible class types of a parameter p and Sim is a scoring function that returns the similarity between the usage pattern and the class signature, which is introduced in Section 3.3. The function $\mathsf{Pos} : \mathit{Var} \times \mathit{UsedAttrs} \times \mathit{DefinedMem} \to \mathcal{P}(\mathit{Type})$ is defined as follows:

$$\mathsf{Pos}(p, \phi_u, \phi_d) = \{c \mid \phi_u(p) \sqsubseteq \phi_d(c)\}$$

where $\sqsubseteq$ is a relation that returns true if the class members $\phi_d(c)$ can support the usage pattern $\phi_u(p)$. In other words, it checks whether $\phi_u(p)$ can use the members of $\phi_d(c)$. Then, we select the

$$\frac{}{\Gamma, \Sigma \vdash_k n : \{\texttt{int}\}, \Sigma} \quad \frac{}{\Gamma, \Sigma \vdash_k s : \{\texttt{str}\}, \Sigma} \quad \frac{}{\Gamma, \Sigma \vdash_k x : \Gamma(x), \Sigma} \quad \frac{}{\Gamma, \Sigma \vdash_k c() : \{c\}, \Sigma} \quad \frac{}{\Gamma, \Sigma \vdash_k [\,] : \langle\rangle, \Sigma} \quad \frac{}{\Gamma, \Sigma \vdash_k \{\} : [\,], \Sigma}$$

$$\frac{\Gamma(x) = T_x \quad T = \bigcup_{c \in T} \Sigma((c, y))}{\Gamma, \Sigma \vdash_k x.y : T, \Sigma} \quad \frac{\Gamma(x) = T_x \quad T' = \bigcup_{\kappa \in T_x} \textsc{Elem}(\kappa)}{\Gamma, \Sigma \vdash_k x[n] : T', \Sigma} \quad \frac{\Gamma(x) = T_x \quad \Gamma(y) = T_y}{\Gamma, \Sigma \vdash_k x\{y\} : \bigcup_{\gamma \in T_x} \bigcup_{t_y \in T_y} \gamma[t_y], \Sigma}$$

$$\frac{\Gamma, \Sigma \vdash_k E_1 : T_1, \Sigma_1 \quad \Gamma, \Sigma_1 \vdash_k E_2 : T_2, \Sigma_2}{\Gamma, \Sigma \vdash_k E_1 \oplus E_2 : \textsc{op}(T_1, T_2), \Sigma_2} \quad \frac{\Gamma(x) = T_x \quad \Gamma(y) = T_y \quad \bigsqcup_{c \in T_x} \textsc{Call}(\Sigma, m, T_y, k) = T, \Sigma'}{\Gamma, \Sigma \vdash_k x.m(y) : T, \Sigma'} \ \textsc{Exist}(c, m)$$

Figure 3: Typing rule for expressions. The function $\textsc{op}(T_1, T_2)$ computes the result type of operation between two types, $\textsc{Exist}(c, m)$ checks the existence of method m in the class c, and $\textsc{Elem}(\kappa)$ returns the set of element types in the list κ.

most likely class type by scoring the similarity. The scoring function Sim is designed as follows:

$$\mathsf{Sim}(\mu_u, \mu_d) = \sum_{\substack{(m, p_u, k_u) \in \mu_u \\ (m, p_d, k_d) \in \mu_d}} \frac{1}{||\,|p_u| - |p_d|\,|| + 1} * \frac{|k_u \cap k_d| + 1}{|k_u \cup k_d| + 1}$$

where p_u and k_u are a set of positional and keyword arguments of the usage pattern, and p_d and k_d are a set of parameters without and with a default value of the class signature. The function Sim returns a larger value when the number of parameters without a default value and the number of positional arguments are similar ($||\,|p_u| - |p_d|\,||$) and the number of common parameters with a default value and keyword arguments is larger ($\frac{|k_u \cap k_d|+1}{|k_u \cup k_d|+1}$). In case of a tie, MostLikely returns `Any` type.

4.3 Inference of Intended Member Types

Next, we define the procedure to infer the intended types of class members (Section 3.4).

The goal is to obtain a refined member environment with the intended types of class members for making a summary α^* that reflects the intended types of class members and reporting type errors $\epsilon^* = \mathit{TypeCheck}(P, \alpha^*)$. As mentioned in Section 3.4, we identified three steps for identifying unintended types. The first step is to check explicit call flows exist. The second step is to cluster type errors that share the same cause. Finally, we find methods that eliminate type errors in a cluster. Through these steps, we generate Δ which satisfies the three conditions. Before introducing the process, we assume that type errors $\epsilon_{\text{set}} = \mathit{TypeCheck}(P, \textsc{Gen}(P, d^*))$, a set of type errors without filtering out unintended type candidates.

Step 1: Checking Explicit Call Flows. We make $\epsilon_{\text{impl}} \subseteq \epsilon_{\text{set}}$ that contains type errors that are not caused by explicit call flows:

$$\epsilon_{\text{impl}} = \{(O, l) \in \epsilon_{\text{set}} \mid \forall o \in O. \neg\exists(m, m_l) \in \textsc{Flow}.\textsc{Cause}(o, m)\}$$

where m_l is a method that includes a line l and $\textsc{Flow} : \mathcal{P}(\mathit{Mthd} \times \mathit{Mthd})$ is a set of all flow from m_1 to m_2 in the program. We assume that the set of call flows is obtained from the type inference process. The procedure $\textsc{Cause}(((c, x), T), m)$ checks where a method m exists, which can make the member $c.x$ to the types T as follows:

$$\textsc{Cause}(((c, x), T), m) \iff \Sigma^m_{c.x}((c, x)) \cap T \neq \emptyset$$
$$\text{where } (\_, \Sigma^m_{c.x}) = \bigsqcup_{(\Gamma, \_) \in \alpha(l^s_m)} [\![\widehat{\text{body}(m)}]\!](\Gamma, \Sigma^*[(c, x) \mapsto \{\texttt{Any}\}]) \tag{4}$$

where l^s_m is the start line of a method m. The $\Sigma^m_{c.x}$ contains the types of class member $c.x$ after calling the method m. We make $\Sigma^m_{c.x}$ through re-inferring the method m for all possible contexts $\Gamma \in \alpha(l^s_m)$ with removing types of the class member (c, x) from Σ^*. In other words, if m is a method that changes the types of the class member $c.x$, then $\Sigma^m_{c.x}$ reflects the type change information of $c.x$.

Step 2: Clustering Type Errors. Since the type error can have multiple causes, we cluster type errors based on similar causes. We define similar causes as those where the same member variables are involved and share the same types as many as possible. To collect type errors with similar causes, we use the DBSCAN algorithm [12]. Generated by DBSCAN, a cluster is considered a set of repeated type errors that share causes. We denote a cluster as $\theta \in \mathcal{P}(\mathit{TypeError})$ and the set of clusters as $\Theta = \{\theta_1, \cdots, \theta_n\}$, which is the result of clustering type errors ϵ_{impl}.

To cluster type errors, we define a distance function as $(1 - \textsc{ErrSim}(\epsilon_1, \epsilon_2))$, and $\textsc{ErrSim}$ is designed as follows:

$$\textsc{ErrSim}(\epsilon_1, \epsilon_2) = |\textsc{SimT}(O_1, O_2)| * \frac{|\textsc{Mem}(O_1) \cap \textsc{Mem}(O_2)|}{|\textsc{Mem}(O_1) \cup \textsc{Mem}(O_2)|}$$

where $\epsilon_1 = (O_1, \_)$ and $\epsilon_2 = (O_2, \_)$, which means that O_1 and O_2 are the set of causes in the type error ϵ_1 and ϵ_2. Functions $\textsc{SimT}$ and $\textsc{Mem}$ are defined as follows:

$$\textsc{SimT}(O_1, O_2) = \bigcup_{((c,x),T_1) \in O_1} \bigcup_{((c,x),T_2) \in O_2} T_1 \cap T_2$$
$$\textsc{Mem}(O) = \{(c, x) \mid ((c, x), \_) \in O\}$$

where $\textsc{SimT}$ is higher when the common class members share the same types, and $\textsc{Mem}$ is higher when the same member variables are involved. The hyperparameters of DBSCAN are set to a threshold of 0.5 and a minimum number of samples of 4 based on observations from the *ignore comments* (`#type: ignore`) used for intended member types to suppress alarms in the benchmark programs.

Step 3: Finding Methods that Eliminate Type Errors. Before this step, we first collect class members involved in type errors in a cluster θ such as $\sigma_\theta = \bigcup_{\epsilon \in \theta} \bigcup_{(O, \_) \in \epsilon} \textsc{Mem}(O)$. Then, we collect methods $\mathcal{M}^\theta_{\text{elim}} \subseteq \mathit{Mthd}$ that eliminate type errors in a cluster θ with σ_θ as follows:

$$\mathcal{M}^\theta_{\text{elim}} = \{m \in \mathit{Mthd} \mid \forall(\_, l) \in \theta, \textsc{TestFlow}(l, \sigma_\theta, m, m_l)\}$$

where m_l is a method that includes a line l and $\textsc{TestFlow}(l, \sigma_\theta, m, m_l)$ checks the call flow $m \longrightarrow m_l$ can eliminate the type error at line l. The function $\textsc{TestFlow}$ is defined as follows:

$$\begin{aligned} & \textsc{TestFlow}(l, \sigma_\theta, m, m_l) \\ \iff & \mathit{Check}(l, \textsc{FlowSummary}(\sigma_\theta, m, m_l)) = \emptyset \end{aligned}$$

where $\textsc{FlowSummary}(\sigma_\theta, m, m_l)$ is a summary for the method m_l assuming the method m is called before as follows:

$$\textsc{FlowSummary}(\sigma_\theta, m, m_l) = \bigsqcup_{(\Gamma, \_) \in \alpha(l^s_m)} G(\text{body}(m_l), (\Gamma, \Sigma^m_{\sigma_\theta}))$$

where $\Sigma^m_{\sigma_\theta}$ contains types of changed class members $(c, x) \in \sigma_\theta$ after calling the method m as follows:

$$\Sigma^m_{\sigma_\theta} = \left[(c, x) \mapsto \begin{cases} \Sigma'^m_{\sigma_\theta}(c, x) & \text{if } \Sigma'^m_{\sigma_\theta}(c, x) \neq \{\texttt{Any}\} \\ \Sigma^*(c, x) & \text{otherwise} \end{cases} \mid (c, x) \in \sigma_\theta \right]$$

where $\Sigma'^m_{\sigma_\theta}(c, x)$ is a result of re-running the type inference for the method m with removing types of class members $(c, x) \in \sigma_\theta$, which is similar to Eq (4), as follows:

$$(\_, \Sigma'^m_{\sigma_\theta}) = \bigsqcup_{(\Gamma, \_) \in \alpha(l^s_m)} [\![\widehat{\text{body}(m)}]\!](\Gamma, \Sigma^*[(c, x) \mapsto \{\texttt{Any}\} \mid (c, x) \in \sigma_\theta])$$

In other words, $\Sigma^m_{\sigma_\theta}$ considers only the members whose types have changed in the method m while keeping the types of other members as types in Σ^*.

In summary, given a cluster θ, the method $m \in \mathcal{M}^\theta_{\text{elim}}$ eliminates type errors in the cluster θ, which means $\Sigma^m_{\sigma_\theta}$ is a refined member environment that removes unintended types of class members $(c, x) \in \sigma_\theta$.

Applying to Summary. Finally, we generate α^* from the summary α with the refined member environment for each cluster θ as follows:

$$\alpha^* = \bigsqcup_{\theta \in \Theta} \bigsqcup_{m_{\text{err}} \in \mathcal{M}^\theta_{\text{err}}} \bigsqcup_{m \in \mathcal{M}^\theta_{\text{elim}}} \alpha[l \mapsto \alpha'(l) \mid l \in \mathit{lines}(m_{\text{err}})]$$
$$\text{where } \alpha' = \textsc{FlowSummary}(\sigma_\theta, m, m_{\text{err}})$$

where $\mathit{lines}(m)$ denote lines of a method m and $\mathcal{M}^\theta_{\text{err}}$ is a set of methods where type errors in a cluster θ occur, defined as $\{m_l \mid \forall(\_, l) \in \theta\}$.

5 EVALUATION

In this section, we experimentally evaluate Pyinder to answer the following research questions:

(1) **Effectiveness**: How effectively does Pyinder detect type errors in our benchmarks?
(2) **Ablation study**: How does each feature of Pyinder contribute to the final performance?
(3) **Generality**: Can Pyinder detect critical type errors in unseen, recently developed open-source Python projects?

We implemented Pyinder on top of Pyre [13] and compared its performance with four existing tools, Mypy (v1.9.0) [46], Pytype (v2024.04.11) [17], Pyre (v0.9.18) [13], and Pyright (v1.1.339) [37], on Python 3.9.18. All experiments were conducted on a Linux machine (Ubuntu 22.04) with an Intel Zeon CPU and 128GB memory.

5.1 Effectiveness and Ablation Study

Setting. We evaluated Pyinder and existing tools on the 68 type errors from 20 open-source projects (from 4k to 417k lines). All tools were evaluated on the same criteria for type errors as described in Section 2. Since all tools, including Pyinder, produce warnings other than type errors, we only counted type-error alarms within our scope (Section 2). We did not set a time budget for the tools, except for Pytype. While other tools finish within an average of about 10 minutes, Pytype took a significantly longer time. Therefore, we set a time limit of 2 hours in the case of Pytype.

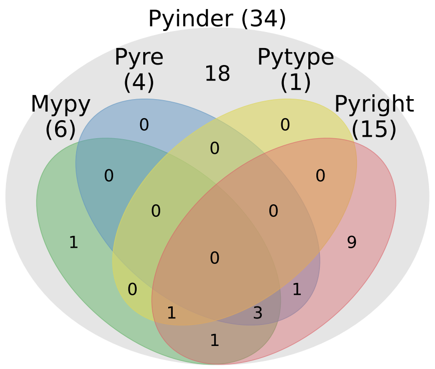

Figure 4: Venn Diagram of the number of detected bugs by each type analysis tool.

Table 2: Comparison with existing type analysis tools on the # of alarms and analysis time (both per 1KLoC).

	Mypy	Pyre	Pytype	Pyright	Pyinder
#alarms/KLoC	4.72	8.38	1.74	11.37	4.57
time(s)/KLoC	0.48	0.26	98.28	0.98	5.78

External libraries. When running a tool on a program, we chose not to include the source code of external libraries that the program relies on. This is because including external libraries increased the program size significantly. On average, the programs in our benchmarks used 26 external libraries, which led to an increase in program size by more than 7.4 times. For example, when external libraries were included, the size of program `homeassistant` increased by 55 times. Thus, instead of including library source code, we ran all tools with `typeshed` [47], which provides type hints for the Python standard library as well as some third-party libraries.

Effectiveness of Pyinder. Figure 4 shows that Pyinder outperforms existing tools in detecting type errors. Pyinder uncovered 34 bugs, more than doubling the count collectively detected by Mypy, Pyre, Pytype, and Pyright. Pyright identified 15 bugs, while Mypy and Pyre detected 6 and 4 bugs, respectively. The set of bugs discovered by Pyinder was a superset of those detectable by existing tools while there was no clear overlap between the existing tools.

Table 2 compares Pyinder and existing tools in terms of the number of alarms and analysis time on the 68 benchmarks (averaged over 1K lines of code). The results clearly show the benefit of using Pyinder over the existing tools; Pyinder reports fewer alarms than existing tools, and running Pyinder requires less time than collectively running the existing tools.

Ablation study. To assess the contribution of each feature described in Section 3, we created 9 variants of Pyinder as shown in Table 3. For example, V1 denotes a variant that is identical to Pyinder except that it merges types into `Any` at merge points.

Figure 5 shows the results of the ablation study. While V1 and V2 decreased the number of detected bugs by 26 and 9, respectively, V3 resulted in increased analysis time and the number of total alarms. When we set the call depth limit to 6 (V4), the cost increased by about 2.1 times, decreasing the number of detected bugs due to timeout, while V5 increased the cost by 1.4 times. When using V6, 5

Table 3: V1 through V9 indicate the variants of Pyinder where each corresponding detailed point introduced in Table 1 is excluded from the analysis.

V1	Merging into `Any` at control flow merge points
V2	Merging into `Any` in collections
V3	Merging dictionaries in a key-type-insensitive manner
V4	Deep call depths (up to 6)
V5	Context-insensitive analysis
V6	No usage-based type inference
V7	Usage-based type inference without scoring
V8	Only considering member types in `__init__` method
V9	No filtering out unintended types

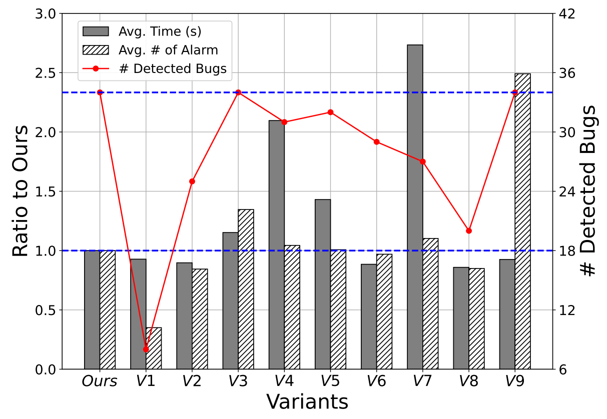

Figure 5: Ablation study results.

```
1 def to_unicode(
2     x: str | bytes
3 ): ...
4
5 def f(self, n, d=None):
6   x = self.r.h.get(n, d)
7
8   # x can be None type
9   return to_unicode(x)
```

(a) Example of a type mismatch bug caused by a commit made five years ago. The type mismatch occurs at line 9 because the variable `x` can be `None` type (Simplified from scrapy-1).

```
1 @property
2 def secret(self) -> str
3 def decrypt(s, h):
4   # TypeError: str+bytes
5   d = s + h
6
7 def f(self):
8   h = ... # bytes
9   decrypt(self.secret, h)
```

(b) Example of incorrect type annotation introduced four years ago. As a result of the incorrect type annotation, a type error alarm occurs at line 6 (Simplified from langchain-1).

Figure 6: The examples of bugs exclusively detected by Pyinder in recently updated projects.

type errors were missed while the analysis cost and the number of alarms were similar to those of Pyinder. Interestingly, usage-based type inference without scoring (V7) increased analysis time by at least 2.5 times, missing type errors due to timeout. V8 represents the strategy of Mypy and Pyre; not considering all possible call scenarios in a class missed 14 type errors. V9 increased the number of alarms by 2.5 times, which is the strategy of Pyright.

5.2 Generality

To check if Pyinder can detect unseen bugs as well, we ran Pyinder on the 9 latest open-source Python projects in Table 4 that were not included in our benchmark in Section 2.

Table 4: Detected bugs in recent open-source Python projects. Status indicates whether the bug has been confirmed or not. (○ : Confirmed/Fixed, × : Won't Fix, △ : Duplicated, - : No response.)

Bug	Status	Mypy	Pyre	Pyright	Pyinder
core-1	-	✗	✗	✓	✓
kivy-1	-	✓	✓	✓	✓
kivy-2	○	✓	✗	✓	✓
kivy-3	-	✓	✓	✓	✓
kivy-4	×	✓	✓	✓	✓
kivy-5	-	✗	✗	✓	✓
langchain-1	○	✗	✗	✗	✓
luigi-1	△	✗	✗	✗	✓
luigi-2	-	✓	✓	✓	✓
pwntools-1	○	✓	✓	✓	✓
pwntools-2	○	✗	✗	✗	✓
pwntools-3	○	✗	✓	✓	✓
pwntools-4	○	✗	✓	✓	✓
telegram*-1	○	✗	✗	✗	✓
scipy-1	○	✓	✓	✓	✓
scipy-2	○	✓	✓	✓	✓
scipy-3	△	✓	✓	✓	✓
scrapy-1	○	✗	✗	✗	✓
tqdm-1	-	✗	✗	✗	✓

*python-telegram-bot

In total, Pyinder successfully found 19 previously-unknown type errors with five times fewer alarms compared to Pyright. For the bugs found, we manually generated test cases based on the alarms and reported them to the developers. The developers confirmed 10 bugs and fixed them. In the case of kivy-4, developers decided not to fix the bug because this bug occurred due to a third-party library.

Surprisingly, Pyinder reported type errors that had persisted for four and five years, which were not detected by any existing tools. Figure 6 shows two bugs exclusively found by Pyinder. In the case of Figure 6a, it is a type mismatch bug caused by a commit made five years ago. Pyinder could infer the type of the variable `x` as `None` and detect the type mismatch at line 9. Figure 6b displays an example of an incorrect type annotation. This annotation was added four years ago to introduce Mypy into the project, but incorrectly written type annotation has persisted until recently. Pyinder could detect the type error at line 5, which was missed by other tools.

5.3 Limitations and Future Directions

Our evaluation also identified limitations of Pyinder in terms of false negatives and positives. The results show that Pyinder could significantly benefit by employing advanced static analysis such as relational analysis to detect more bugs and reduce false alarms.

False Negatives. Pyinder missed 34 out of the 68 bugs. Among them, Pyinder failed to detect 29 bugs due to unknown external libraries and inputs. Figure 7 illustrates representative cases. Both cases raise type errors in the user code, but the type errors are caused by the external libraries (Figure 7a) or inputs (Figure 7b). To find these errors, we require to know the return types of external functions or more information about external inputs, e.g., configuration files or inputs of unseen type in source codes. The remaining 5 type errors require more advanced type inference algorithms than Pyinder. Figure 8 illustrates representative cases. Figure 8a presents

```
1 import ext
2 # ext is an external lib
3 # ext.f returns a string
4 x = e.f()
5 x+1 # TypeError: str+int
```

(a) Example that requires the return type of external function `ext.f()`.

```
1 # conf.json : {'a': 1}
2 f = open("conf.json")
3 conf_dict = json.loads(f)
4 # TypeError: dict + list
5 conf_dict + [1]
```

(b) Example that requires the content of the external input `conf.json`.

Figure 7: Examples of false negatives due to unknown external libraries and inputs.

```
1 def f(x=1, y=1):
2     pass
3
4 a = {'c': 1, 'd': 2}
5
6 # TypeError
7 f(**a) # f(c=1, d=1)
```

(a) Example that requires the value of dictionary unpacking for a method call.

```
1 def f(x: int, y):
2     # y: Optional[int]
3     if x > 0:
4         if y is None:
5             y = 0
6     # TypeError
7     return x+y
```

(b) Example that requires non-trivial type inference for a variable y.

Figure 8: Examples of false negatives required advanced type inference.

```
1 class Symbol:
2     def __radd__(
3         self, x
4     ): return x+1
5 # False Alarm
6 # int.__add__(1, Symbol)
7 1+Symbol()
```

(a) False alarm at line 7 due to the limitation of modeling magic methods.

```
1 def f(x, y):
2     # False Alarm
3     if x > 0: y+1
4     # False Alarm
5     else:     y+'1'
6 f(1, 1)
7 f(-1, '1')
```

(b) False alarm at lines 3 and 5 due to the limitation of type analysis.

Figure 9: False positive examples

a bug example that requires a value analysis. It is necessary to unpack the dictionary a and infer the exact value of keys in the dictionary to detect the type error at line 6. Figure 8b displays another example, where a type error occurs at line 7 because the type of variable `y` can be `None`. However, it is difficult to infer the type of `y` as the `None` type because a typical flow analysis cannot preserve the information at line 4. The type of `y` can be inferred as `None` in the true branch at line 4, but due to line 5, the type of `y` is updated to `int` type. Thus, by the end of the branch, the type of `y` is inferred as `Any` (false-branch) or `int` (true-branch), which does not include `None`. As future work, we plan to enhance Pyinder to preserve the type information at line 4 using, for example, a backward analysis.

False Positives. We now discuss the remaining alarms generated by Pyinder in Section 5.1. Over the 68 benchmarks, Pyinder produced a total of 37285 alarms, and we sampled 269 alarms for investigation (with a confidence interval of 90% and an error level of 5%). Among them, we could identify 6 true alarms and 106 false alarms. The remaining 157 alarms could not be definitively classified into true or false without the developer's confirmation because they need a deeper understanding of the programs. We found that the 106 false alarms are caused by the limitation of Pyinder's static analysis. Figure 9 displays two false alarm cases.

One interesting finding is that while 69% ($\frac{73}{106}$) of false alarms were raised due to inadequate modeling, more than half, 37 out of 73, were caused by the limitations of modeling magic methods. Figure 9a presents an example of a false alarm by modeling operators as magic methods. In this example, a type error does not occur at line 7 because the `Symbol` class has the magic method `__radd__`, which is called when the left operand does not support the corresponding operation `__add__`. Unfortunately, the analyzer changes `1+Symbol()` to `int.__add__(1, Symbol())`, which leads to the false alarm at line 8.

Like existing tools, Pyinder supports modeling common magic methods, e.g., `__add__` for the + operator. However, Pyinder currently does not support more complex and tricky cases, such as the `__radd__` method for the + operator. Method `__radd__` is invoked under tricky and complex conditions: when `__add__` raises a `NotImplemented` exception while the left and right operands are of different types, or when the right operand is a subclass of the left operand but has a different implementation of the magic method. We found that manually handling all such behaviors is extremely burdensome; as another example, even a simple expression `a.x` can invoke at least six different magic methods depending on the context, which requires understanding over 700 words of Python documentation. In addition, we sometimes noticed that the documentation is incorrect or unclear, and therefore understanding the behavior of magic methods requires to consult the CPython implementation. For these reasons, existing tools including Pyinder have only implemented the most common cases of magic methods, leaving room for improvement in the magic method modeling. To address this difficulty, as future work we plan to develop an intermediate representation of Python that does not have implicit behaviors and re-implement Pyinder on it.

In other cases, 31% ($\frac{33}{106}$) of false alarms were raised due to spurious relations as seen in Figure 9b. In this case, a relational analysis is required to keep track of the relationship between integer values (first parameter) and types (second parameter), such as "if the first parameter is greater than 0, then the second parameter is an integer type.", in order to avoid false alarms at lines 3 and 5.

5.4 Lessons Learned

We summarize the lessons learned from the evaluation results.

Lesson 1: Current type analysis tools have limitations in detecting real-world type errors. Current type analysis tools for Python, such as Mypy, Pyre, Pytype, and Pyright, have limitations in their ability to fully support the features required to detect real-world type errors. This shortcoming resulted in a low bug detection rate of less than 25%. Furthermore, the inability to fully leverage these crucial features leads to an increase in false alarms or higher costs. This result indicates the need for more advanced approaches to detect type errors more effectively.

Lesson 2: Novel features of Pyinder. Pyinder introduces novel techniques that collect types selectively based on empirical observation, thereby enhancing practicality without sacrificing detection rate. Without this design, the analysis could lead to a cost increase of more than 2.5 times or generate a significant number of false alarms. To address this issue, we proposed a novel design for each feature, such as handling specific merge points or collection types and selectively inferring the types of class members. As a result, Pyinder achieved a reasonable analysis time and the lowest number of alarms among existing tools, while maintaining a high bug detection rate.

Lesson 3: The necessity of handling tricky magic method modeling. We observed that 69% of false alarms were caused by the limitations in handling tricky cases of magic methods even though Pyinder supports the most common cases of magic methods. For example, `__radd__` method for the + operator has tricky conditions and even a simple expression `a.x` can invoke at least six different magic methods depending on the context. In other words, due to the complex implicit behaviors of magic methods in expressions, it is challenging to model all expressions accurately as magic methods. Thus, we plan to develop an intermediate representation of Python without implicit behaviors to address this issue.

Lesson 4: There is room for further advancement in detecting type errors. We discovered that more advanced techniques are required in terms of both false negatives and positives. To reduce false negatives, we need to infer the exact value of variables or use backward analysis to more accurately infer the type of variables. To reduce false positives, keeping track of the relationship between variables is required to avoid spurious alarms. This result indicates that it is necessary to design more sophisticated type inference techniques to detect type errors more effectively.

6 RELATED WORK

Empirical Studies on Python Type Analysis Tools. Empirical studies have been conducted on industrial static type analysis tools for Python such as Pyre [13], Pyright [37], Mypy [46], and Pytype [17]. Khan et al. [32] investigated type-related defects, and concluded that about 15% of bugs can be prevented by simply using a type checker such as Mypy. Xu et al. [59] demonstrated that combining static type checking tools with dynamic instrumentation can improve the performance. Rak-amnouykit et al. [48] compared similarities and differences between Mypy and Pytype. Our work differs from these prior works in that we aim to improve existing tools (vs. [32]), do not rely on dynamic analysis (vs. [59]), and focus on type error detection (vs. [48]).

Static and Dynamic Type Inference. Static analysis has been extensively used to infer types of Python programs [5, 9, 18, 21, 33, 49, 50] based on, for example, constraint-based type inference [21] or abstract interpretation [18, 49]. While these works have focused on designing type inference approaches to broadly support Python features, our work focuses on how to selectively collect types in a more detailed manner to enhance effectiveness. In other dynamic languages, there is also a large amount of work on type inference, e.g., [2, 4, 6, 14, 15, 19, 19, 22, 26, 38, 45]. However, these works have proposed type inference techniques to support their languages such as Ruby [2, 14, 15] or JavaScript [4, 6, 19, 19, 22, 26, 45], which are not suitable for Python programs.

A number of techniques have been proposed to infer types for Python programs dynamically [3, 10, 11, 25, 40, 45, 55, 60]. They have concentrated on dynamic observations to infer types correctly [3, 45]. In contrast, we suggest static features to improve type error detection in Python.

Selective Static Analysis. Our four features in Section 3 can be understood as practical guidelines for selectively applying expensive type analysis techniques. Previous techniques for selective static analysis have been primarily developed for other languages, such as C and Java [24, 27, 28, 31, 34–36, 51, 54], and they have focused on preserving the precision while reducing the cost [54] or vice versa [36]. In designing a selective analysis, designing a right set of features is a key [24, 27, 34, 36], but none of the existing works proposed such features for Python type analysis.

Learning-Based Type Inference. Recently, there have been many works on type inference using machine learning techniques [1, 7, 23, 29, 39, 43, 44, 56, 57, 61]. They have proposed a probabilistic type inference technique from variable names [61], or various models for type inference, such as graph [1], sequence [23, 44], and hierarchical models [39]. Peng et al. [43] and Wei et al. [56] combine static analysis and learning-based type inference. Our work is largely orthogonal to this line of work, and can be combined with them.

7 CONCLUSION

Detecting type errors early is pivotal for enhancing the reliability of Python programs. However, statically detecting type errors poses a challenge with current type analysis tools because manual type annotations are uncommon in real-world Python code. This paper presents empirical observations revealing four key features for practical static type error detection in Python. Experimental results using 68 bugs demonstrate that our approach can detect 34 (50%) bugs, more than double the number detectable by existing tools with fewer alarms. Finally, we reported 19 bugs in the latest real-world programs and analyzed the remaining weaknesses of Pyinder. We hope that our efforts in this work pave the way for improving the practicality of static type analysis tools for Python.

DATA AVAILABILITY

For open science, we make our tool and all experimental data publicly available via the GitHub repository[1].

ACKNOWLEDGMENT

This work was supported by Samsung Electronics Co., Ltd (IO201216-08209-01), Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair, 50%), ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (IITP-2024-2020-0-01819, 10%), the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No. 2021R1A5A1021944, 20%), and the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.RS-2024-00440780, Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains, 10%).

[1]https://github.com/kupl/PyinderArtifact

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 91–105. https://doi.org/10.1145/3385412.3385997

[2] Jong-hoon An, Avik Chaudhuri, and Jeffrey S Foster. 2009. Static typing for Ruby on Rails. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 590–594.

[3] Jong-hoon An, Avik Chaudhuri, Jeffrey S Foster, and Michael Hicks. 2011. Dynamic inference of static types for Ruby. *ACM SIGPLAN Notices* 46, 1 (2011), 459–472.

[4] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards type inference for JavaScript. In *European conference on Object-oriented programming*. Springer, 428–452.

[5] Brett Cannon. 2005. *Localized type inference of atomic types in python*. Ph. D. Dissertation. Citeseer.

[6] Wontae Choi, Satish Chandra, George Necula, and Koushik Sen. 2015. SJS: A type system for JavaScript with fixed object layout. In *International Static Analysis Symposium*. Springer, 181–198.

[7] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2018. RefiNym: Using Names to Refine Types. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 107–117. https://doi.org/10.1145/3236024.3236042

[8] Luca Di Grazia and Michael Pradel. 2022. The evolution of type annotations in python: an empirical study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, Singapore, Singapore,) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/3540250.3549114

[9] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. 2018. Ariadne: analysis for machine learning programs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Philadelphia, PA, USA) *(MAPL 2018)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3211346.3211349

[10] dropbox. 2017. Pyannotate: Auto-generate PEP-484 annotations. https://github.com/dropbox/pyannotate

[11] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: a dynamic analysis framework for Python. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, Singapore, Singapore,) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 760–771. https://doi.org/10.1145/3540250.3549126

[12] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining* (Portland, Oregon) *(KDD'96)*. AAAI Press, 226–231.

[13] facebook. 2017. Pyre-check: Performant type-checking for python. https://github.com/facebook/pyre-check

[14] Michael Furr, Jong-hoon An, and Jeffrey S Foster. 2009. Profile-guided static typing for dynamic scripting languages. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 283–300.

[15] Michael Furr, Jong-hoon An, Jeffrey S Foster, and Michael Hicks. 2009. Static type inference for Ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*. 1859–1866.

[16] Zheng Gao, Christian Bird, and Earl T. Barr. 2017. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 758–769. https://doi.org/10.1109/ICSE.2017.75

[17] google. 2015. Pytype: A static type analyzer for Python code. https://github.com/google/pytype

[18] Michael Gorbovitski, Yanhong A Liu, Scott D Stoller, Tom Rothamel, and Tuncay K Tekle. 2010. Alias analysis for optimization of dynamic languages. In *Proceedings of the 6th Symposium on Dynamic Languages*. 27–42.

[19] Brian Hackett and Shu-yu Guo. 2012. Fast and precise hybrid type inference for JavaScript. *ACM SIGPLAN Notices* 47, 6 (2012), 239–250.

[20] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. 2014. An Empirical Study on the Impact of Static Typing on Software Maintainability. *Empirical Softw. Engg.* 19, 5 (oct 2014), 1335–1382. https://doi.org/10.1007/s10664-013-9289-1

[21] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-Based Type Inference for Python 3. In *International Conference on Computer Aided Verification*. https://api.semanticscholar.org/CorpusID:51873753

[22] Phillip Heidegger and Peter Thiemann. 2010. Recency types for analyzing scripting languages. In *European conference on Object-oriented programming*. Springer, 200–224.

[23] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 152–162. https://doi.org/10.1145/3236024.3236051

[24] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-learning-guided selectively unsound static analysis. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 519–529. https://doi.org/10.1109/ICSE.2017.54

[25] Instagram. 2017. https://github.com/Instagram/MonkeyType

[26] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *International Static Analysis Symposium*. Springer, 238–255.

[27] Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 179 (nov 2020), 30 pages. https://doi.org/10.1145/3428247

[28] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 100 (oct 2017), 28 pages. https://doi.org/10.1145/3133924

[29] Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. 2021. Learning Type Annotation: Is Big Data Enough?. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1483–1486. https://doi.org/10.1145/3468264.3473135

[30] Jetbrains. 2020. https://www.jetbrains.com/lp/python-developers-survey-2020/.

[31] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. *SIGPLAN Not.* 48, 6 (jun 2013), 423–434. https://doi.org/10.1145/2499370.2462191

[32] Faizan Khan, Boqi Chen, Daniel Varro, and Shane McIntosh. 2022. An Empirical Study of Type-Related Defects in Python Projects. *IEEE Transactions on Software Engineering* 48, 8 (2022), 3145–3158. https://doi.org/10.1109/TSE.2021.3082068

[33] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.15

[34] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 141 (oct 2018), 29 pages. https://doi.org/10.1145/3276511

[35] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/3236024.3236041

[36] Jingbo Lu, Dongjie He, and Jingling Xue. 2021. Eagle: CFL-Reachability-Based Precision-Preserving Acceleration of Object-Sensitive Pointer Analysis with Partial Context Sensitivity. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 46 (jul 2021), 46 pages. https://doi.org/10.1145/3450492

[37] microsoft. 2019. Pyright: Static Type Checker for Python. https://github.com/microsoft/pyright

[38] Nevena Milojković, Mohammad Ghafari, and Oscar Nierstrasz. 2017. Exploiting Type Hints in Method Argument Names to Improve Lightweight Type Inference. In *Proceedings of the 25th International Conference on Program Comprehension* (Buenos Aires, Argentina) *(ICPC '17)*. IEEE Press, 77–87. https://doi.org/10.1109/ICPC.2017.33

[39] Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2241–2252. https://doi.org/10.1145/3510003.3510124

[40] Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. 2019. Dynamic Type Inference for Gradual Hindley–Milner Typing. *Proc. ACM Program. Lang.* 3, POPL, Article 18 (jan 2019), 29 pages. https://doi.org/10.1145/3290331

[41] Octoverse. 2023. https://octoverse.github.com/.

[42] 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* (, Singapore, Singapore,) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 922–934. https://doi.org/10.1145/3540250.3549130

[43] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2019–2030. https:

//doi.org/10.1145/3510003.3510038

[44] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-Based Validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/3368089.3409715

[45] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 314–324.

[46] python. 2012. Mypy: Static Typing for Python. https://github.com/python/mypy

[47] python. 2015. typeshed: Collection of library stubs for Python, with static types. https://github.com/python/typeshed

[48] Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. 2020. Python 3 Types in the Wild: A Tale of Two Type Systems. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages* (Virtual, USA) *(DLS 2020)*. Association for Computing Machinery, New York, NY, USA, 57–70. https://doi.org/10.1145/3426422.3426981

[49] Armin Rigo and Samuele Pedroni. 2006. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 944–953.

[50] Michael Salib. 2004. *Starkiller: A static type inferencer and compiler for Python.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[51] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. *SIGPLAN Not.* 46, 1 (jan 2011), 17–30. https://doi.org/10.1145/1925844.1926390

[52] IEEE Spectrum. 2023. https://spectrum.ieee.org/the-top-programming-languages-2023.

[53] Andreas Stuchlik and Stefan Hanenberg. 2011. Static vs. Dynamic Type Systems: An Empirical Study about the Relationship between Type Casts and Development Time. In *Proceedings of the 7th Symposium on Dynamic Languages* (Portland, Oregon, USA) *(DLS '11)*. Association for Computing Machinery, New York, NY, USA, 97–106. https://doi.org/10.1145/2047849.2047861

[54] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 147 (oct 2021), 27 pages. https://doi.org/10.1145/3485524

[55] Nikos Vasilakis, Grigoris Ntousakis, Veit Heller, and Martin C. Rinard. 2021. Efficient Module-Level Dynamic Analysis for Dynamic Languages with Module Recontextualization. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1202–1213. https://doi.org/10.1145/3468264.3468574

[56] Jiayi Wei, Greg Durrett, and Isil Dillig. 2023. TypeT5: Seq2seq Type Inference using Static Analysis. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=4TyNEhI2GdN

[57] Jiayi Wei, Maruth Goyal, Greg Durrett, and Işıl Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. *ArXiv* abs/2005.02161 (2020). https://api.semanticscholar.org/CorpusID:211027382

[58] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1556–1560. https://doi.org/10.1145/3368089.3417943

[59] Wenjie Xu, Lin Chen, Chenghao Su, Yimeng Guo, Yanhui Li, Yuming Zhou, and Baowen Xu. 2023. How Well Static Type Checkers Work with Gradual Typing? A Case Study on Python. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. 242–253. https://doi.org/10.1109/ICPC58990.2023.00039

[60] Zhaogui Xu, Peng Liu, Xiangyu Zhang, and Baowen Xu. 2016. Python Predictive Analysis for Bug Detection. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 121–132. https://doi.org/10.1145/2950290.2950357

[61] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python Probabilistic Type Inference with Natural Language Support. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 607–618. https://doi.org/10.1145/2950290.2950343

[62] Xin Zhang, Rongjie Yan, Jiwei Yan, Baoquan Cui, Jun Yan, and Jian Zhang. 2022. ExcePy: A Python Benchmark for Bugs with Python Built-in Types. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 856–866. https://doi.org/10.1109/SANER53432.2022.00104