What Types Are Needed for Typing Dynamic Objects? A Python-Based Empirical Study

Ke Sun[1], Sheng Chen[2], Meng Wang[3], and Dan Hao[1(✉)]

[1] Key Lab of HCST (PKU), MOE; SCS, Peking University, Beijing, China
sunke@stu.pku.edu.cn, haodan@pku.edu.cn
[2] The Center for Advanced Computer Studies, UL Lafayette, Lafayette, USA
sheng.chen@louisiana.edu
[3] University of Bristol, Bristol, UK
meng.wang@bristol.ac.uk

Abstract. Dynamic object-oriented languages, such as Python, Ruby, and Javascript are widely used nowadays. A distinguishing feature of dynamic object-oriented languages is that objects, the fundamental runtime data representation, are highly dynamic, meaning that a single constructor may create objects with different types and objects can evolve freely after their construction. While such dynamism facilitates fast prototyping, it brings many challenges to program understanding. Many type systems have been developed to aid programming understanding, and they adopt various types and techniques to represent and track dynamic objects. However, although many types and techniques have been proposed, it is unclear which one suits real dynamic object usages best. Motivated by this situation, we perform an empirical study on 50 mature Python programs with a focus on object dynamism and object type models. We found that (1) object dynamism is highly prevalent in Python programs, (2) class-based types are not precise to handle dynamic behaviors, as they introduce type errors for 52% of the evaluated polymorphic attributes, (3) typestate-based types, although mostly used in static languages, matches the behaviors of dynamic objects faithfully, and (4) some well-designed but still lightweight techniques for object-based types, such as argument type separation and recency abstraction can precisely characterize dynamic object behaviors. Those techniques are suitable for building precise but still concise object-based types.

Keywords: Type System · Empirical Study · Python

1 Introduction

Dynamic object-oriented languages, such as Python, Ruby, and Javascript are commonly used across many domains. They use dynamic typing to increase reusability and flexibility, facilitating fast prototyping (development) not provided by most static languages. In particular, unlike in static object-oriented languages where objects have mostly fixed attributes and their types [25], objects

C.-K. Hur (Ed.): APLAS 2023, LNCS 14405, pp. 24–45, 2023.
https://doi.org/10.1007/978-981-99-8311-7_2

in dynamic languages are highly dynamic. Both the attributes and types of an object may be changed freely over its life cycle.

To illustrate, consider the Python program in Fig. 1, which is adapted from *Rich*, a terminal beautification tool [34]. This program defines two instance objects (shorted as objects) of `Panel`: `panel1` and `panel2`. However, the types of the two objects are not stipulated by class definition and can be set and changed freely. We refer to this phenomenon as *object dynamism*.

Behaviors Causing Object Dynamism. Object dynamism originates from two sources: constructor polymorphism and object evolution [42,54]. With **constructor polymorphism**, objects of different types can be made from the same constructor. Lines 17-18 show such an example. Specifically, although `panel1` and `panel2` are created from the same `__init__`, `panel1` has the type $\tau_1 = \{$`title` : `Text`, `width` : `int`, `height` : `int`$\}$, while `panel2` has the type $\tau_2 = \{$`title` : `str`, `width` : `NoneType`$\}$[1]. In particular, two attributes (`title` and `width`) are shared but with different types, and one attribute (`height`) appears only in τ_1. After the construction phase, objects can dynamically evolve, making the types continue to change, denoted as **object evolution**. Lines 19-20 present such an example, which changes the `width` attribute of `panel2` from `NoneType` to `int` and adds the attribute `height`.

```
1  class Panel:
2      def __init__(self, title, width):
3          self.title = title
4          self.width = width
5          if self.width is not None:
6              self.height = self.width
7      def _title(self):
8          if isinstance(self.title, str):
9              return Text.from_markup(self.title)
10         else:
11             return self.title.copy()
12     def setheight(self, height):
13         self.height = height
14     def measure(self):
15         return self.width * self.height
16
17 panel1 = Panel(Text(), 42)
18 panel2 = Panel("Example Table", None)
19 panel2.width = 5 # modification
20 panel2.setheight(42) # extension
```

Fig. 1. An example Python program

An Empirical Study on Dynamic Objects for Type Systems. As dynamic languages are being used to build more and more important and large software systems, many type systems [10,23,30,33] have been developed to aid program comprehension and early programming error detection. Those type systems come up with special object types for representing dynamic objects. The most widely adopted design choice is to augment the class-based types, which assign a single type to all objects created from the same class, with features such as union types [9], and the ability to reason about type tests [28,48] and local type

[1] We refer the constructed types of `panel1`/`panel2` by τ_1/τ_2 in the rest of this paper.

assignments [3]. Due to their performance superiority and annotating convenience, class-based types have been extensively used in industrial and academic type systems [10,19,22,30,40,53]. However, it is not clear whether they provide dynamic objects with type representations that are precise enough.

On the other hand, many object-based type systems [8,17,45,55] have also been proposed, which assign a unique type to each object, according to its abstract address and evolution processes. Since those techniques provide dynamic objects with more precise representations, it is clear that they produce fewer type errors compared with class-based types. However, the actual improvement has not been investigated on a large scale. Meanwhile, the contributions of individual aspects of object-based types to the improvements are not understood.

To help understand the prevalence and characteristics of dynamic object behaviors, as well as the effectiveness of existing object types, we present an empirical study based on a dynamic analysis of 50 mature Python programs with over 3.76 million LOC. There have been several studies [4,24,42,54] that also consider the dynamic object behaviors. However, their analysis is not focused on types, and thus their implications on type systems are limited.

Some of our significant findings include: (1) Both constructor polymorphism and object evolution are very prevalent. The average proportions of classes exposing them are both higher than 20%, and they often occur at the same time. (2) Class-based types can be a practical choice, especially when paired with the ability to reason about type tests and local assignments. Although false type errors are reported for 52% of the polymorphic attributes when experimenting with them, those type errors are largely due to attribute absence, which is notoriously hard to detect statically [32]. (3) Typestate-based types, as already utilized extensively to represent object evolution in static languages [6,29,46], can be promising to be adopted to dynamic languages, considering the large proportion of attribute-absences-related errors, which turn out to be introduced mainly by object evolution. (4) Object-based types are found to be effective in representing dynamic objects. In particular, the ability to perform strong updates [8,23] is critical to increase the precision.

In summary, this paper makes the following contributions.

1. A large corpus composed of 50 mature Python projects with over 3.76 million LOC and a well-designed dynamic analysis infrastructure capable of analyzing precise and detailed object behaviors.
2. An empirical study of object dynamism in Python with findings and advice for evaluating current type systems and inspiring future type systems.

The artifact of this paper, containing the analysis infrastructure and the experiment scripts, can be accessed via https://github.com/ksun212/Python-objects.

2 Background

In this section, we introduce dynamic object behaviors in detail and review related studies on dynamic behaviors.

2.1 Dynamic Object Behaviors in Python

Constructor Polymorphism. We name the behavior that objects of different types are made from a single constructor as constructor polymorphism. In Python, a constructor is a normal member method that initializes attributes. In our example, different objects are constructed in a functional way, i.e., object types are solely decided by argument types. However, as an imperative language, the program's state can also influence the behaviors of constructors. For example, an attribute may be set when a global variable holds a specific value.

Object Evolution. We name the behavior that an object changes its type after being constructed as object evolution. Based on the action, object evolution can be classified into (1) extension (i.e., adding new attributes), (2) modification (i.e., modifying types of attributes), and (3) deletion (i.e., deleting attributes).

2.2 Existing Studies on Dynamic Behaviors

Although object dynamic behaviors are significant for building type systems and other static analyses for dynamic languages, only a few studies have been done to study their actual prevalence. In particular, Richards et al. [42] and Wei et al. [54] conducted empirical studies on the dynamic behaviors in JavaScript programs. However, the former did not distinguish between changes of attribute types (changes that lead to object evolution) and attribute values (changes do not lead to object evolution), and the latter measured constructor polymorphism via the number of runtime instances instead of object types that matter more to type system design. Only two pieces of work [4,24] investigated the dynamic behavior of Python objects. However, they mainly focus on object evolution, without investigating constructor polymorphism. Meanwhile, neither of the above studies analyzes the effectiveness of existing object types.

Several studies investigated other features of dynamic languages, such as eval expressions [41], callsite polymorphism [5,27], and dynamic variables [14]. These studies are not from type systems' perspective, the focus of this work.

3 Types for Dynamic Objects

In this section, we present a type syntax for class-based types and discuss local type refinements. Then, we review important aspects for object-based types.

3.1 Class-Based Types

Type Syntax. To consolidate the notion of class-based types in this paper, we propose a type syntax, which mostly coincides with the definition of class-based types in existing class-based type systems [10,19,

$$
\begin{aligned}
\tau &::= cls \mid \tau \vee \tau \mid abs \\
CT &::= cls : \{\overline{attr : \tau}\}, CT \mid \emptyset \\
\Gamma &::= x : cls, \Gamma \mid \emptyset \\
x &\in Program\ Variables \\
attr &\in Attribute\ Names \\
cls &\in Class\ Names
\end{aligned}
$$

Fig. 2. Syntax for class-based types.

22,30,40,53]. The only difference is that we model attribute absence using a constant type *abs*, while other systems provide type qualifiers [31] or simply omit it [19,30,53] (Fig. 2).

Under this syntax, the type environment (Γ) maps from variables to class names. The class table (CT) maps from class names to object types, which is a record labeled by attribute names with values ranging over attribute types (τ). τ can be another class name (such as builtin classes `int`, `str`, and user-defined classes `Panel`, `Text`), a special constant type *abs* to signify that the attribute is absent or a union of two attribute types. For the example in Fig. 1, a type system using this syntax gives type environment $\{$`panel1` : `Panel`, `panel2` : `Panel`$\}$, and class table $\{$`Panel` : $\{$`title` : `Text` $\vee$ `str`, `width` : `int` $\vee$ `NoneType`, `height` : `int` $\vee$ `abs`$\}\}$.

Polymorphic Attributes Cause Type Errors. Although class-based types provide a natural way to express object dynamism, they often introduce type errors. To see this, consider type-checking the method `measure`, which results in a type error, since `width` can be `NoneType` while `height` can be *abs*, both invalidating the addition operation. Those type errors are caused by polymorphic attributes, i.e., the attributes holding union types, when not all components of the union type can be used in any access-site of the attribute.

Local Type Refinements. To eliminate suspicious type errors, *local type refinements* [3,28,48] are often used to refine the union types. The core observation is that developers tend to use type tests and local assignments to refine the type of polymorphic attributes, which can be utilized to refine the union types to a smaller range thus eliminating type errors. For example, consider type-checking the method `_title`, in the first branch, the type of `title` is refined to be `str` by the type test `isinstance(self.title,str)`. For an example of local assignments, consider inserting `if self.width is None: self.width=42` into the beginning of the method `measure`, which refines the type of `width` to be only `int`.

3.2 Object-Based Types

Class-based type systems assign all objects belonging to the same class with the same type. We have discussed that this design choice introduces spurious type errors. Although local type refinements can be used to eliminate type errors, they rely on type tests or local assignments, which are unavailable in many cases. To eliminate the type errors, another idea is to assign more precise types to dynamic objects, by exposing more fine-grained object addresses and performing strong updates for object evolution as much as possible [1,8,17,44,45,52,55]. In the following of this paper, we name this kind of typing discipline as object-based types, whose effectiveness will be discussed in our study.

Store Abstraction. Each dynamic object receives a unique address from the heap (store). While types of objects may be identified by their addresses, few type systems support this, since addresses are allocated at runtime while the type systems we are investigating perform static checking. Instead, type systems often use abstractions of store to denote object types. In class-based type systems, each object uses the class name as its address. All objects of the same class share the

same address. To keep sound, the types of all those objects, along their life cycles, must be merged. This is the reason that many attributes in class-based type systems are polymorphic, causing spurious type errors.

Using class names as addresses often leads to imprecision. A prominent approach is extending the class name with the construction location. Construction location can be annotated [16,45] or inferred [19,23,36]. In our example, we can separate the types of `panel1` and `panel2` using this approach, yielding $\Gamma = \{\texttt{panel1} : \mathit{Panel}@17, \texttt{panel2} : \mathit{Panel}@18\}$ and $CT = \{\mathit{Panel}@17 : \{\texttt{title} : \texttt{Text}, \texttt{width} : \texttt{int}, \texttt{height} : \texttt{int}\}, \mathit{Panel}@18 : \{\texttt{title} : \texttt{str}, \texttt{width} : \texttt{int} \vee \texttt{NoneType}, \texttt{height} : \texttt{int} \vee \texttt{abs}\}\}$. Note that $\mathit{Panel}@18$ must cover all the types of `panel2` in its life cycle. Suppose the method `measure` is called on `panel1`, this type system would correctly accept it but still reject the call on `panel2`.

Sometimes, using the construction locations is not precise enough, since many construction-sites can be called many times (e.g., occurring inside a function that is repeatedly called.). To handle this, location polymorphism [16] and k-callsite [36] have been proposed. We evaluate the help of k-callsites in our study.

Flow Sensitivity. As we discussed earlier, to keep sound, the types in the life cycle of an object must be merged. The reason is that the store abstraction (i.e., CT) must over-approximate the store at any time of the program execution. One common approach to relax this constraint is *flow-sensitive store abstraction* [2,16,36,43], which allows each program location to be associated with a different store abstraction. In our example, this yields $CT = \{..., \mathit{Panel}@18 : \{\texttt{title} : \texttt{str}, \texttt{width} : \texttt{NoneType}, \texttt{height} : \texttt{abs}\}\}$, $CT' = \{.., \mathit{Panel}@18 : \{\texttt{title} : \texttt{str}, \texttt{width} : \texttt{int} \vee \texttt{NoneType}, \texttt{height} : \texttt{abs}\}\}$, and $CT'' = \{.., \mathit{Panel}@18 : \{\texttt{title} : \texttt{str}, \texttt{width} : \texttt{int} \vee \texttt{NoneType}, \texttt{height} : \texttt{int} \vee \texttt{abs}\}\}$, where $CT/CT'/CT''$ denote the store abstraction associated with Line 18/19/20. Suppose that an access of `width` is inserted before Line 19, which requires it to have the type `NoneType`, then flow sensitivity allows this access to be accepted since the system knows that `width` can only be `NoneType` before Line 19. In a flow-insensitive system, this access would be incorrectly rejected. However, the method `measure` still cannot be called on `panel2`, even after Line 20. This is because, due to the potential existence of aliases, one attribute must be typed with all the types that are previously assigned to the attribute, a methodology often called "weak updates" [8].

Strong Updates. A type system that is able to replace the old type for an attribute with a new type when an object evolves is said to be able to perform "strong updates". Strong updates have to be performed on the top of flow-sensitive store abstraction. With strong updates, the type system knows $CT'' = \{..., \mathit{Panel}@18 : \{\texttt{title} : \texttt{Text}, \texttt{width} : \texttt{int}, \texttt{height} : \texttt{int}\}\}$, and subsequently, allows `measure` be called on `panel2` after Line 20.

Due to the alias problem, strong updates can not be performed arbitrarily [8]. It is widely known that strong updates can be applied to linear addresses [2,16,43], i.e., the addresses that refer to only one object. In our example, we have seen that the class name extended with construction locations linearly refer to the two objects. In general, more precise techniques like location polymorphism [16] and k-callsite [36] make more addresses linear. However, those

Table 1. Statistics and Categories of Experiment Subjects

Category	Subjects	LOC
Scientific Computing (SCI)	networkx, pinyin, sklearn, nltk, altair, kornia, stanza, featuretools, dvc, torch, pandas, seaborn, statsmodels, pyod, spacy, snorkel	2.29M
Programming (PRG)	pydantic, typer, bandit, isort, arrow, jedi, black, yapf, mypy	0.46M
Web (WEB)	requests, flask, impacket, routersploit, itsdangerous, pelican, sphinx	0.24M
Terminal (TER)	rich, thefuck, cookiecutter, click, prompt_toolkit	0.14M
Formating (FMT)	jinja, pypdf, markdown, weasyprint	0.12M
Utility (UTL)	pywhat, icecream, pendulum, pre_commit, faker	0.34M
Others (OTH)	newspaper, wordcloud, pyro, pyecharts	0.14M
All (ALL)	50 projects	3.76M

techniques have been witnessed to significantly increase running overhead [36] or incur excessive annotation burden [16]. Another widely adopted solution is recency abstraction [8,23]. Recency abstraction splits one address into two, one for the most recently constructed object and one for all previously constructed objects. Supposing just use the class name as addresses, recency abstraction gives $\Gamma = \{\texttt{panel1} : Panel_s, \texttt{panel2} : Panel_r\}$. For the most recently constructed object, since it is the only object referred to by the address, strong updates can be performed, while all previously constructed objects can only be updated weakly[2]. The assumption of recency abstraction is object evolution usually happens to the most recent object, instead of the previously constructed objects. Our example obeys this assumption since only `panel2` evolves.

4 Experimental Design

This study investigates the following questions around object dynamism in Python.
RQ1. Are dynamic object behaviors prevalent in the wild?
RQ2. How effective are class-based types and object-based types?

4.1 Subjects

In this experiment, we use 50 Python projects from Github. In particular, we select the top 50 popular Python projects on Github whose testing framework is *pytest*, after removing the ones that need to be run on multiprocessing mode (which causes potential races of the log file) or have special requirements (e.g., network or peripheral devices). By requiring *pytest* to be the testing framework, we can run all the subjects under a unified interface, simplifying the experiment setup. Due to space limitations, when presenting and analyzing the results, we

[2] In our example, only `panel1` is not recent. However, in general, there can be many.

divide these 50 subjects into 7 categories and present the results for each category. Table 1 presents these categories, the subjects they contain, and their total LOC. We present the details of these 50 subjects on the artifact.

To learn the dynamic behaviors of these Python projects, we run the test suite of each subject. In order to facilitate the analysis, we prune the test suites until they can be executed within 12 h and produce a trace file of less than 20G. The details of the used tests are also presented on the artifact.

4.2 Tracing and Analysis Infrastructure

Overview of the Infrastructure. Our infrastructure consists of a tracing module and an analysis module. The tracing module is based on CPython 3.9[3]. The tracing module traces the execution of a subject and records the events related to Python objects, such as the start/end of object construction, and assigning/deleting object attributes. The events are recorded with the necessary information to conduct our analysis, such as where the event happens (program location), and which object is related to the event. The analysis module analyzes the events to construct and evaluate class-based types and object-based types.

Constructing Types. We construct class-based types and object-based types from the traces. To construct class-based types for a class c, if one of its objects is observed to be assigned with an attribute a and type c' in the trace, we add a to the attribute set of c, and add c' to the types of $c.a$. If one attribute a is owned by one object of the class, but is not owned by another, we add *abs* to the types of $c.a$. We also add *abs* if the attribute is added/deleted in the evolution phase, since the attribute is absent before/after the extension/deletion. For class-based types, all objects of the same class share the same type. The construction of object-based types is similar, the only difference is that all objects of the same object address (instead of class) share the same type. For object-based types, we simulate flow-sensitive store abstraction by constructing different stores for different locations. On top of flow-sensitive store abstraction, We simulate strong updates by performing strong updates whenever the condition is met (i.e., the object address is linear or obeys recency abstraction).

Note that when constructing types, we construct for all classes observed in the traces. However, when evaluating the types, we focus on the objects whose classes are defined directly in the program, ignoring the objects defined in built-in or third-party libraries, to better reflect the nature of the analyzed programs.

Evaluating Types. We evaluate the effectiveness of class-based types and object-based types against the access-sites. To illustrate, consider the class-based type of `panel1`, namely, {`title : Text` $\vee$ `str`, `width : int` $\vee$ `NoneType`, `height : int` $\vee$ `abs`}. Supposing we observe that the attribute `title` of `panel1` is accessed at Line 11, we evaluate this access-site in two steps. The first step performs local type refinements based on the type tests [28,48] and local assignments [3]. In our example, the polymorphic attribute `title` holds two classes, i.e., `str` and `Text`.

[3] https://github.com/python/cpython/tree/3.9.

However, for the access-site at Line 11, only `Text` is valid, while `str` is ruled out at Line 8. The second step judges if the types after refinement (i.e., `Text`) can be used in the access-site, i.e., satisfy the constraints of the access-site.

The complete constraints in one access-site can not be collected without building a complicated analysis. In our study, we utilize a substantial subset of the complete constraints, named *local constraints*. The major generation rules of local constraints are presented in Fig. 3. In this figure, *obj.a* denotes the access expression, T denotes the set of all types of the attribute a after refinement ($\{$`Text`$\}$ for `title` in our example). *Attr* is the function to extract the attribute set of one object type. In our example, since we have `self.title.copy`, we can generate the constraint $abs \notin \{\texttt{Text}\} \wedge copy \in Attr(\texttt{Text})$, which is true by examining the type set ($\{Text\}$) and the type of `Text`.

Similarly, consider another access-site of `title`, at Line 9. We can refine the type of `title` to `str` this time. However, since `title` is directly passed to another function, we cannot collect any local constraints, and thus we do not evaluate this access-site. So far, we have examined all the access-sites of `title`. Since it satisfies all examined access-sites, it is determined to be safe. For object types to be precise, they should make as many polymorphic attributes safe, since unsafe polymorphic attributes are very likely to be false alarms, due to the fact that the access-sites are collected dynamically without witnessing runtime type errors.

$$
\begin{array}{rlrl}
obj.a & \Longrightarrow abs \notin T & obj.a + e & \Longrightarrow \forall \tau \in T, \_\_\text{add}\_\_ \in Attr(\tau) \\
obj.a() & \Longrightarrow \forall \tau \in T, \_\_\text{call}\_\_ \in Attr(\tau) & obj.a.f & \Longrightarrow \forall \tau \in T, f \in Attr(\tau) \\
obj.a[e] & \Longrightarrow \forall \tau \in T, \_\_\text{getitem}\_\_ \in Attr(\tau) & len(obj.a) & \Longrightarrow \forall \tau \in T, \_\_\text{len}\_\_ \in Attr(\tau)
\end{array}
$$

Fig. 3. Local Constraint Generation Rules

5 Results and Analysis

In this section, we answer the two research questions in two subsections.

5.1 Prevalence of Dynamic Behaviors

In this section, we study the prevalence of dynamic behaviors, as well as several important aspects of them, to help characterize the difficulty of analyzing them.

Constructor Polymorphism. Table 2 presents the statistics of classes that expose constructor polymorphism, where the second column presents the total number of classes in a specific category and in all subjects. The third and fourth columns present statistics on these classes, which we refer to as ratio results and subject-wise median results, respectively. To obtain ratio results (given by column "Ratio"), we divide the total number of classes that expose constructor polymorphism in one category by the total number of classes in that category. To obtain the subject-wise median results (given by column "Median"), we calculate the proportion of classes exposing constructor polymorphism for each subject in

Table 2. Prevalence of constructor polymorphism. Class shows the number of classes. Ratio and Median show the proportion of classes.

Category	Class	Ratio	Median
SCI	1773	0.24	0.28
PRG	360	0.32	0.20
WEB	643	0.12	0.14
TER	214	0.27	0.20
FMT	258	0.30	0.40
UTL	28	0.18	0.20
OTH	266	0.17	0.23
ALL	3542	0.23	0.20

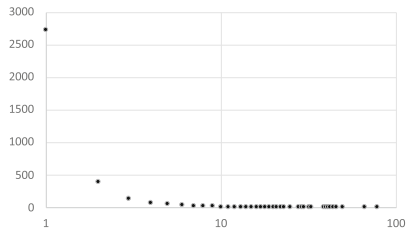

Fig. 4. Degree of constructor polymorphism. The X-axis/Y-axis denotes the number of distinct object types/classes.

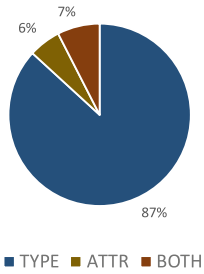

Fig. 5. Overall Relation among Object Types.

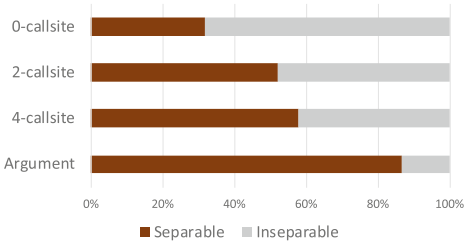

Fig. 6. Separability of Different Construction Contexts.

one category and take the median. Ratio results emphasize the overall proportion, while subject-wise median results emphasize the subject-wise differences. Due to space limitations, we present the results for each category and a summary of all subjects. The results of individual subjects are given on the artifact.

From this figure, the ratio and median proportion of classes that expose constructor polymorphism are both over 20%, indicating that constructor polymorphism is **prevalent**. Besides, we can also notice the differences among categories, e.g., *UTL*, *WEB* and *OTH* have fewer classes that expose constructor polymorphism. Many classes belonging to those categories have relatively simple functionality and do not need constructor polymorphism.

Now we know that constructor polymorphism is prevalent. But how polymorphic are polymorphic constructors, and how difficult it is to analyze them?

How Polymorphic. Figure 4 shows the degree of constructor polymorphism, that is, the number of distinct object types made out of polymorphic constructors. According to Fig. 4, most of the polymorphic constructors have a relative **low degree** (less than 5), indicating that typically only a few object types are made.

Table 3. Prevalence of object evolution. RO/MO presents the ratio/median for objects, while RC/MC presents that for classes.

Category	Object	RO	MO	Class	RC	MC
SCI	3×10^6	0.31	0.21	1773	0.36	0.28
PRG	1×10^6	0.11	0.02	360	0.11	0.15
WEB	2×10^5	0.28	0.40	643	0.31	0.43
TER	5×10^4	0.08	0.07	214	0.18	0.23
FMT	1×10^6	0.46	0.49	258	0.47	0.61
UTL	1×10^5	0.02	0.02	28	0.25	0.20
OTH	3×10^5	0.17	0.21	266	0.42	0.38
ALL	6×10^6	0.27	0.12	3542	0.33	0.28

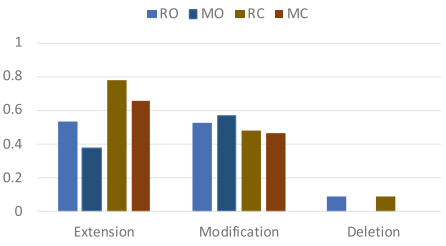

Fig. 7. Actions of Object Evolution

Polymorphic constructor constructs objects of different types. But, how different are these types? To answer this question, we divide the polymorphic constructors into three kinds, according to whether they construct object types with inconsistent attribute types (labeled *TYPE* in Fig. 5, e.g., $\{attr : int\}$ and $\{attr : str\}$), inconsistent attribute sets (*ATTR*, e.g., $\{attr : int\}$ and $\{attr : int, attr2 : int\}$), or both (*BOTH*, e.g., $\{attr : int\}$ and $\{attr : str, attr2 : int\}$). Figure 5 shows the proportion of those three kinds, which shows that most (87%) polymorphic constructors construct object types with consistent attribute sets but inconsistent attribute types. This suggests that polymorphic attribute types, instead of attribute sets, are contributed by constructor polymorphism. Thus, if the main cause of false type errors is attribute sets (we will see that it is), constructor polymorphism should be generally innocent.

Separability. Constructors in Python are just normal functions. To precisely analyze functions, context sensitivity is the prominent technique used in static analysis and type systems [21,26,37,52]. Context sensitivity relies on function call contexts to separate the return types of different function calls. The most widely used function call contexts are k-callsites [21,26,37] and argument types [1,52], namely k-length call stacks and types of arguments of call-sites. Figure 6 shows the proportion of polymorphic constructors that can be separated by argument types or k-callsite contexts. For a polymorphic constructor, if given an argument type/k-callsite of the constructor, only one object type is observed to be constructed under the argument type/k-callsite, we mark the constructor as *separable* by argument types/k-callsites. Otherwise, it is *inseparable*. According to Fig. 6, **argument types** effectively separate more than 80% of polymorphic constructors, implying the high dependency of constructed object types on the argument types. However, k-callsites are not as effective as argument types, although the separability increases with a longer callsite.

Object Evolution. Table 3 shows the prevalence of object evolution. Its second, third, and fourth columns present the total number of objects, the ratio, and the median proportion of objects that expose object evolution. The last three columns of this table present the total number of classes, the ratio, and the median proportion of classes that expose object evolution. Note that if any

object of one class exposes object evolution, we regard the class as exposing object evolution. From the table, a large number of objects/classes (27%/33%) expose object evolution, indicating the **prevalence** of object evolution. Besides, *SCI*, *FMT*, and *WEB* have more objects/classes exposing object evolution and we suspect the reason to be the specific functionalities of these categories. For example, objects of class `DecisionTree` of the subject *sklearn* in the *SCI* category are extended with new attributes after they are trained.

Now we know that object evolution happens frequently. But how do the objects evolve, and how difficult it is to analyze the evolution?

How. Fig. 7 presents the statistics of evolution action. It shows the ratio and median proportion of objects and classes that expose extension, modification, and deletion, among all the **evolving** objects and classes. From this table, extension and modification are dominant. Meanwhile, deletion seldom occurs: although the ratio of deletion is around 10%, the median proportion is zero.

Furthermore, we analyze the pattern of evolution and find that most of the evolution processes are monotonic. Monotonicity is a property that has been used extensively in previous studies on object evolution [7,11,47]. Different from the types described in Sect. 3, types based on monotonicity allow object evolution to be soundly analyzed without the need for store abstraction [39]. In this study, following previous studies, we define monotonic evolution as the evolution in which attributes are only added but not deleted, and when the type of one attribute is changed, it only changes from a type to its subtype (we only consider nominal subtype). We calculate the ratio of evolving objects that evolve monotonically and find the ratio very high (85%). We believe that although monotonicity has not been widely spread around the techniques for dynamic languages, it is promising to propose systems utilizing it.

Function. The function where one evolution action happens significantly influences the difficulty of analyzing it. As shown in Fig. 8, we divide the functions where evolution actions happen into three kinds: *Local* means the evolution action happens in the same function as the construction-site. *Method* means the evolution action happens in one of the member methods of the evolving objects. Those two kinds generally allow modular reasoning to be performed [30,51,55] and are easier to analyze; *Others* denotes other functions. From the figure, we can see that most functions are member methods of the object. There are also some (23%) functions belonging to *Local*. Those findings indicate modular techniques for analyzing object evolution should be able to cover most cases.

Condition. Figure 9 shows the conditions under which object evolution happens. More precisely, this figure shows the distribution of evolution locations (evolution-sites), based on the **intra**procedural preconditions. The intraprocedural precondition of one evolution-site is the condition that must be satisfied to reach the evolution-site from the function entry. While in actual systems, **inter**procedural preconditions (i.e., the condition to reach the function callsite) must also be considered, collecting them requires a complicated infrastructure. Thus, we use intraprocedural preconditions to speculate the difficulty of analyzing the conditions. We split the intraprocedural preconditions into four major

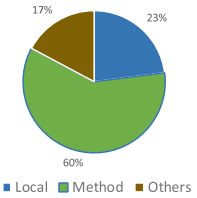

Fig. 8. Functions of Object Evolution

Table 4. Overall Dynamism

Category	Ratio		Median	
	Static	Hybrid	Static	Hybrid
SCI	0.54	0.14	0.59	0.11
PRG	0.60	0.03	0.67	0.02
WEB	0.61	0.05	0.50	0.05
TER	0.64	0.09	0.69	0.08
FMT	0.46	0.23	0.53	0.27
UTL	0.64	0.07	0.60	0.00
OTH	0.52	0.11	0.54	0.14
ALL	0.56	0.11	0.56	0.06

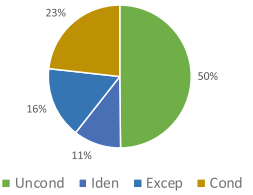

Fig. 9. Condition of Evolution

Table 5. Results of the Evaluation of Class-based Types

Category	Attributes		Access-site Evaluation							Absences	
	ALL	POL	EVA	UNI	TES	LOC	RN	RA	RB	ABS	CABS
SCI	18127	3802	1127	0.12	0.36	0.50	0.54	0.87	0.93	2623	239
PRG	2143	360	100	0.50	0.62	0.72	0.76	0.77	0.82	23	2
WEB	3842	957	217	0.13	0.26	0.41	0.59	0.74	0.95	572	58
TER	1126	187	36	0.36	0.78	0.83	0.92	0.86	0.94	12	0
FMT	1832	746	245	0.15	0.19	0.26	0.28	0.92	0.95	460	39
UTL	146	21	2	0.00	0.00	0.50	0.50	0.50	0.50	6	1
OTH	2145	231	31	0.42	0.55	0.55	0.65	0.87	0.97	114	6
ALL	29361	6304	1758	0.16	0.35	0.48	0.53	0.86	0.93	3810	345

kinds: (1) *Uncond*, where the precondition is simply $True$; (2) *Iden*, where the precondition is not $True$, but all branches conduct evolution identically[4]; (3) *Excep*, where the precondition is just to exclude the exceptional execution path (e.g., `if cond then raise exception else evolve`); (4) *Cond*, where the precondition does not belong to the previous three cases. From the figure, we can see most (77%) of the evolution-sites fall into *Uncound*, *Iden*, or *Excep*. Meanwhile, the proportion of *Cond* is still non-negligible (23%). This kind of evolution can be precisely analyzed only by path-sensitive type systems. However, most existing type systems for dynamic objects are not path-sensitive; instead, they merge the different types of one object in different branches. Although there do exist path-sensitive systems based on dependent and intersection types [16], or abstract interpretation [36], those systems suffer from performance issues, and complex type annotations [50]. To this end, we argue that more advanced techniques should proposed, maybe by making better use of the potential correspondence between conditional evolution and conditional accesses.

Overall Dynamism. Table 4 shows the overall dynamism of evaluated projects. The **second** and **fourth** columns show the ratio and median proportion of classes that do not expose any dynamic behaviors (i.e., static classes). The **third** and **fifth** columns show the metrics of classes that expose both kinds of dynamic

[4] In such cases, there is no need to precisely distinguish the branches.

behaviors (i.e., hybrid classes). From the table, the proportions of static classes in all classes and within a project are both 56%. Since static objects are ideal for performing program optimization [15,49], we believe that their high proportion encourages more optimization for them. Also, the infrastructure of this paper is a good start for identifying static objects/classes.

On the other hand, the ratio of classes that expose both behaviors is non-negligible (11%). This implies that two behaviors are sometimes utilized simultaneously because they may serve different purposes. Thus, we believe it is promising to develop unified techniques to handle both dynamic behaviors.

5.2 Effectiveness of the Types

In this section, we analyze the effectiveness of class-based types and object-based types. We start with the analysis of class-based types.

Class-Based Types. As discussed, polymorphic attributes are a good indicator of the effectiveness of class-based types. Thus, we first analyze polymorphic attributes, followed by an evaluation of the effectiveness of class-based types.

Polymorphic Attributes. Recall that an attribute is **polymorphic** if it holds a union type. In other words, it is assigned with multiple classes or *abs*. The second and third columns of Table 5 present the number of all attributes and polymorphic attributes. We observe that the proportion of polymorphic attributes is high (6304/29361 = 21.4%), indicating their prevalence in dynamic languages.

To understand how types held by polymorphic attributes are related, we classify polymorphic attributes into six kinds in Fig. 10. These six kinds include: (a) *ABS*, where each attribute (e.g., `height`) holds a single class or *abs*, (b) *OPT*, where each attribute (e.g., `width`) holds a single class or `NoneType`, after removing *abs*, (c) *NOM*, where each attribute holds multiple classes that, after removing `NoneType` and *abs*, have nominal relation (i.e., the nominal join is not `Object`), (d) *NUM*, where each attribute holds multiple classes that, after removing `NoneType` and *abs*, are all numeric (i.e., builtin numeric classes, `int` and `float`, and user-defined numeric classes, e.g., `numpy.float32`), (e) *STRU*, where each attribute holds multiple classes that, after removing `NoneType` and *abs*, have structural relation (i.e., the structural join is not `Object`), and (f) *OTHE*: the polymorphic attributes not belonging to previous kinds. When a polymorphic attribute belongs to more than one kind, we classify it into the earlier appeared kind because it is more specific. For example, the polymorphic attribute holding `int` and `float` belongs to both *NUM* and *STRU*. We classify it into *NUM* since all attributes belonging to *NUM* belong to *STRU*, but not vice versa.

From Fig. 10, we observe that a large proportion (53%+21%) of polymorphic attributes are *ABS* and *OPT*, meaning that most attributes are polymorphic because of *abs* or `NoneType`. Nevertheless, a significant proportion (26%) of polymorphic attributes are actually assigned with multiple classes even after removing *abs* and `NoneType`. Luckily, we find that most of those attributes do not belong to *OTHE*, indicating that a supertype (in the sense of nominal, numeric, or structural) is likely to be the intended type of each such attribute. Those

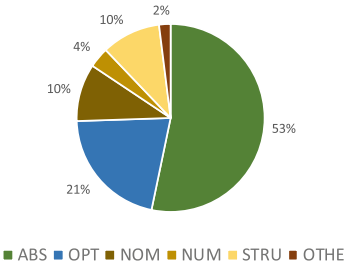

Fig. 10. Classification

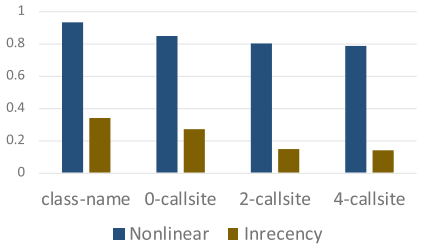

Fig. 11. Object Addresses

attributes are likely to be used without precisely distinguishing their actual types.

Evaluation. We will next evaluate the safety of accessing polymorphic attributes, as specified in Sect. 4.2. In this study, we evaluate only the polymorphic attributes for which at least one access-site exposes local constraints since constraints are necessary for the evaluation. Column *EVA* of Table 5 gives the ratio of evaluated attributes, i.e., 27% (1758/6304). The reason that many polymorphic attributes are not evaluated is twofold. First, there are 27% (1711/6304) attributes that we observe no access-site. The other attributes have access-sites observed, but those access-sites expose constraints that can not be collected by our local constraint generation rules. For example, the attributes may be passed into another function, put into a global container, or directly returned.

Columns *UNI* through *LOC* of Table 5 show the ratio of evaluated attributes that are determined to be safe. According to Sect. 4.2, an attribute is safe if it satisfies the local constraints of all evaluated access-sites. Also, local type refinements (i.e., type tests and local assignments) can be used to refine the types of the evaluated attribute and make the accesses safe. To analyze the effectiveness of local type refinements, we show the ratio of safe attributes with and without local type refinements. **First**, Column *UNI* shows the ratio of safe attributes without local type refinements. In this case, all access-sites of an attribute have to be safe for all its classes. Overall, *UNI* attributes are about 16%. The *UNI* is much higher in some categories, such as *PRG* and *OTH*, indicating that though polymorphic, attributes may be used uniformly without distinguishing their types. **Second**, the attribute may be type-tested against how it will be used, as illustrated in Sect. 3.1. The ratio of attributes that are safe due to such tests or the previous reason is shown in Column *TES*. **Third**, accessing polymorphic attributes may be safe thanks to local assignments [3] before the access, as illustrated also in Sect. 3.1. The ratio of attributes whose accesses are safe due to local assignments or previous reasons is shown in Column *LOC*. The *TES* and *LOC* results for all subjects are 35% and 48%, respectively, and are much higher in some categories (e.g., *PRG*, *TER*), meaning that local type refinements can significantly increase the effectiveness of class-based types.

Threats to Validity. There are three threats to the validity. **First**, since we only evaluate 27% of all polymorphic attributes, it is possible that the findings cannot be generalized to all polymorphic attributes. We do believe that the results are generalizable, however, since the difficulty in collecting constraints is due to the surrounding contexts which do not affect typing in general. To test this assumption, we sampled 300 polymorphic attributes from the 73% unevaluated attributes and conducted a manual analysis of them. We provided the necessary annotations to calculate *LOC* and *RB* for those attributes. The results are very close to the ones in Table 5, with $LOC = 50\%$, and $RB = 97\%$. **Second**, since we do not consider interprocedural constraints, it is possible that the types are actually unsafe to use in the access-site, but we report them to be safe. To this end, we manually investigate 100 safe attributes from the attributes belonging to *LOC*, and analyze if they are actually safe. Among the 100 attributes, we find no unsafe attributes. Thus, we believe that local constraints are effective in determining the safety of polymorphic attributes. **Third**, our interpretation of type tests is not complete. We only consider built-in type tests such as `isinstance` and `hasattr` and their boolean combinations, and ignore user-defined type tests and value tests. It is possible that the attributes considered unsafe by our approach are actually safe if we consider more complete type tests. To this end, we additionally classify all attributes "mentioned" in the type tests as safe. In this setting, *LOC* reaches 51%, only 3% higher than the original *LOC* results. Thus, we believe that our interpretation of type tests covers most cases.
Attribute Absences. For the 52% of attributes whose accesses are deemed unsafe, we manually investigate them and find the main reason is that attributes may hold *abs* or `NoneType` but are used without type tests or local assignments. Combined with our observation that a large proportion of attributes are *ABS* or *OPT*, we conduct an additional experiment to evaluate the connection between those two types and type safety. Specifically, for each attribute deemed as unsafe, we discard `NoneType`, *abs*, and both of them and rerun the experiment. For example, when evaluating `width`/`height` against their access-sites in `measure`, we remove `NoneType` and *abs* from their types and evaluate `int` only. We show the results of removing `NoneType`, *abs*, and both in columns *RN*, *RA*, and *RB*, respectively. According to the results, removing `NoneType` helps increase the proportion (48% to 53%), while removing absences helps significantly (48% to 86%), implying that attribute absences are the main cause of the type errors.

Since attribute absences are the main cause of the type errors, we conduct a specialized analysis of their sources, as shown in Columns *ABS* and *CABS*. *ABS* shows the number of polymorphic attributes holding *abs*, while *CABS* shows the same number when we only consider just-constructed objects. It can be observed from the results that construction contributes a little ($345/3810 = 9\%$) to attribute absences, which implies that evolution is the main source of absences.

Object-Based Types. As discussed earlier, object addresses play an important role in object-based types. In this section, we first investigate several object addresses and then the effectiveness of object-based types.

Table 6. Results of the Evaluation of Object-based Types

CAT	Flow-insensi				Flow-sensi				Strong Updates (wo/w Recency)			
	CLS	L0	L2	L4	CLS	L0	L2	L4	CLS	L0	L2	L4
SCI	0.50	0.55	0.55	0.56	0.51	0.55	0.56	0.56	0.52/0.65	0.58/0.72	0.61/0.94	0.62/0.94
PRG	0.72	0.78	0.85	0.85	0.74	0.79	0.86	0.86	0.74/0.78	0.79/0.84	0.87/0.91	0.87/0.91
WEB	0.41	0.44	0.46	0.46	0.41	0.45	0.46	0.46	0.48/0.83	0.55/0.91	0.57/0.93	0.57/0.93
TER	0.83	0.86	0.92	0.92	0.83	0.86	0.92	0.92	0.86/0.89	0.89/0.92	0.94/0.97	0.94/0.97
FMT	0.26	0.27	0.27	0.27	0.26	0.27	0.27	0.27	0.26/0.35	0.27/0.37	0.27/0.37	0.28/0.37
UTL	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	1.00/1.00	1.00/1.00	1.00/1.00	1.00/1.00
OTH	0.55	0.58	0.58	0.58	0.55	0.58	0.58	0.58	0.65/0.77	0.77/0.97	0.77/0.97	0.77/0.97
ALL	0.48	0.51	0.53	0.53	0.49	0.52	0.53	0.53	0.50/0.65	0.55/0.71	0.58/0.86	0.59/0.86

Object Addresses. Recall that an address is nonlinear if it refers to multiple objects. Nonlinear addresses prevent strong updates. There are two solutions to this problem: more precise addresses or recency abstraction. To measure the effectiveness of more precise addresses, we compare four kinds of addresses, including class names and class names extended with 0/2/4-callsite of construction-sites (the 0-callsite case is simply construction location and so on for the rest). In Fig. 11, *Nonlinear* results measure the proportion of evolving classes that have at least two objects referred to by a single address. To measure the effectiveness of recency abstraction, we compare object addresses with and without recency abstraction. For an address with recency abstraction, strong updates cannot be performed when it refers to *inrecent*, evolving objects. *Inrecency* measures the proportion of evolving classes that have at least one address witnessing such a problem. From the figure, we can observe that the class name can easily be nonlinear, as 93% of the class names are nonlinear. More precise addresses help insignificantly. However, recency abstraction helps significantly. Even with the most imprecise object address (class name), only 34% of evolving classes belong to *Inrecency*. With 2-callsite only 15% of the evolving classes belong to *Inrecency*. In other words, with 2-callsite, most (85%) classes support strong updates.

Evaluation. We now extend the evaluation of class-based types to object-based types. The results are shown in Table 6. Columns under "Flow-insensi" show the proportion of safe polymorphic attributes when typed under flow-insensitive store abstraction with the four kinds of object addresses. *CLS* shows the proportion when class names are used as object addresses. This column is the same as the *LOC* column of Table 5. *L0*, *L2*, and *L4* show the proportions when 0/2/4 callsite of the construction-sites are used to extend the class names. Note that to keep the comparison with class-based types straightforward, we enable local refinements and use the same polymorphic attributes as in the evaluation of class-based types. It is possible that one attribute (e.g., `title`) is not polymorphic anymore when typed under more precise addresses. Even so, we still include it. It can be observed that using more precise object addresses increases the pre-

cision. However, the improvement is not significant. Columns under "Flow-sensi" show the same metrics but with flow-sensitive store abstraction. It can be shown that flow-sensitivity alone cannot improve the precision much. Flow-sensitivity alone (i.e., without strong updates) is effective only if one object can do something before taking some evolution actions, but not after. We can observe that such conditions should be rare since flow-sensitivity alone is not effective. This observation also aligns with our previous finding that object evolution is mostly monotonic, which means that objects gain new abilities as the evolution goes on, but never lose old abilities.

Columns under "Strong Updates" show the same metrics, but strong updates are performed for linear addresses/addresses that obey recency abstraction. Overall, we can find that the ability to perform strong updates significantly improves precision. This finding conforms to our previous finding that most errors are caused by attribute absences, which are themselves caused by object evolution. Strong updates make it possible to distinguish the object type before and after the evolution, and thus eliminate attribute absences and increase precision. Meanwhile, it can be observed that only performing strong updates for linear addresses is not sufficient, and using recency abstraction helps significantly, especially when used together with *L2* or more precise addresses.

Note that our evaluation of object-based types only reveals the upper bound of the precision. The precision of object-based types is also influenced by other factors such as the analysis of function calls/control flows (which determines whether the effects of different function calls/control flows are precisely separated). As the results in Sect. 5.1 suggest, the analysis of them is not a trivial task. However, since we want to focus on the factors that are specific to object types, while those factors influence the typing of the whole program, we do not conduct a detailed analysis of them and assume them to be precisely analyzed[5]. In other words, our aim is not to conduct a systematic evaluation of object-based types, but to derive observations on some important and representative factors.

Discussion. Now, we summarize the observations gained from our analysis and make suggestions on real-world type systems.

Class-Based Types. As can be observed from our experiment, class-based types can handle many polymorphic attributes. The effectiveness of class-based types is contributed significantly by local type refinement techniques, especially the ability to interpret type tests (a feature typically referred to as occurrence typing [13,28,48]). Moreover, since we find that our relatively simple "occurrence typing" covers most cases, we believe that the technique for occurrence typing needs not be very complicated to fulfill practical uses.

On the other hand, many polymorphic attributes cannot be handled by class-based types yet, especially when they hold *abs*. To make this insight more concrete, we check the polymorphic attributes with Pyright [35], a widely-used class-based type checker for Python, using class-based types similar to ours. More specifically, we randomly sample 100 polymorphic attributes from the 52% of the polymorphic attributes thought as unsafe in our study. We provide neces-

[5] As a dynamic analysis, we can naturally simulate the precision analysis of them.

sary type annotations for those polymorphic attributes and their related code and check the code with Pyright. We found that type errors are reported for 95 of the attributes. The reason that errors are not reported for some attributes is due to the unsound aspects of Pyright. For example, Pyright does not raise any error for the attribute whose corresponding class overrides the `getattr` method.

Object-Based Types. It is obvious from the results that object-based types are much more precise than class-based types. However, we want to emphasize that although our results are in favor of object-based types to a large extent, we do not mean that class-based types are useless since most of the spurious type errors related to class-based types are just caused by attribute absences, which are normally not expected to be excluded statically[6]. What's more, type checking/inference of class-based types is faster, and annotating class-based types is much easier than object-based types [38]. Thus, we suggest using these two kinds of types accordingly. In the scenarios where errors such as type mismatches are emphasized, and attribute absences matter less, we recommend class-based types. Meanwhile, in the scenarios where more rigorous verification is expected [12,18], we believe that object-based types are more suitable. In particular, in dependent type systems [16,50], object-based types with strong updates should be preferred, since they can help dependent type systems prove stronger properties.

Typestate-Based Types. At last, we discuss typestates [6,46]. By modeling evolution processes as finite state machines, typestates allow fine-grained representation of classes whose instances evolve. Typically, users must provide typestate annotations to use such types. However, recent studies [12,20] have proposed an inference algorithm for typestate annotations, when only attribute absences are concerned and evolution happens only inside member methods. Since we have found that attribute absences are the main cause of type errors and evolution does happen mainly inside member methods, we believe that it is promising to utilize typestate-based types. Future work in this direction should carefully differentiate among three states of an attribute, that is, absent, uninitialized (holding `None`), and initialized. What's more, adopting typestate-based types also requires some kind of strong update mechanism and can benefit from the monotonicity, which some findings in our study should help.

6 Conclusion and Future Work

In this paper, we conduct a systematic evaluation of object dynamism and object types. Our results reveal the prevalence of dynamic object behaviors. We also evaluate the widely used types for handling object dynamism and draw important implications for them. Although our study is set on Python, we expect the main findings to be transferable to other dynamic languages, since they share the same core semantics. For future work, we plan to build a type system for dynamic object-oriented languages based on the insights gained in this study.

[6] Even some static languages such as Java do not exclude them.

References

1. Agesen, O.: The Cartesian product algorithm. In: Tokoro, M., Pareschi, R. (eds.) ECOOP 1995. LNCS, vol. 952, pp. 2–26. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-49538-X_2
2. Ahmed, A., Fluet, M., Morrisett, G.: L^3: a linear language with locations. Fundamenta Informaticae **77**(4), 397–449 (2007)
3. Aiken, A., Foster, J.S., Kodumal, J., Terauchi, T.: Checking and inferring local non-aliasing. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pp. 129–140 (2003)
4. Åkerblom, B., Stendahl, J., Tumlin, M., Wrigstad, T.: Tracing dynamic features in python programs. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 292–295 (2014)
5. Åkerblom, B., Wrigstad, T.: Measuring polymorphism in python programs. In: Proceedings of the 11th Symposium on Dynamic Languages, pp. 114–128 (2015)
6. Aldrich, J., Sunshine, J., Saini, D., Sparks, Z.: Typestate-oriented programming. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, pp. 1015–1022 (2009)
7. Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for JavaScript. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 428–452. Springer, Heidelberg (2005). https://doi.org/10.1007/11531142_19
8. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 221–239. Springer, Heidelberg (2006). https://doi.org/10.1007/11823230_15
9. Barbanera, F., Dezaniciancaglini, M., Deliguoro, U.: Intersection and union types: syntax and semantics. Inf. Comput. **119**(2), 202–230 (1995)
10. Bierman, G., Abadi, M., Torgersen, M.: Understanding TypeScript. In: Jones, R. (ed.) ECOOP 2014. LNCS, vol. 8586, pp. 257–281. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44202-9_11
11. Blaudeau, C., Liu, F.: A conceptual framework for safe object initialization: a principled and mechanized soundness proof of the celsius model. Proc. ACM Program. Lang. **6**(OOPSLA2), 729–757 (2022)
12. Bravetti, M., et al.: Behavioural types for memory and method safety in a core object-oriented language. In: Oliveira, B.C.S. (ed.) APLAS 2020. LNCS, vol. 12470, pp. 105–124. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64437-6_6
13. Castagna, G., Laurent, M., Nguyen, K., Lutze, M.: On type-cases, union elimination, and occurrence typing. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022)
14. Chen, Z., Li, Y., Chen, B., Ma, W., Chen, L., Xu, B.: An empirical study on dynamic typing related practices in python systems. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 83–93 (2020)
15. Choi, W., Chandra, S., Necula, G., Sen, K.: SJS: a type system for JavaScript with fixed object layout. In: Blazy, S., Jensen, T. (eds.) SAS 2015. LNCS, vol. 9291, pp. 181–198. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48288-9_11
16. Chugh, R., Herman, D., Jhala, R.: Dependent types for JavaScript. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, pp. 587–606 (2012)
17. Eifrig, J., Smith, S., Trifonov, V.: Sound polymorphic type inference for objects. In: Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 169–184 (1995)

18. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, pp. 302–312 (2003)
19. Furr, M., An, J.h., Foster, J.S., Hicks, M.: Static type inference for ruby. In: Proceedings of the 2009 ACM Symposium on Applied Computing, pp. 1859–1866 (2009)
20. Golovanov, I., Jakobsen, M.S., Kettunen, M.K.: Typestate inference for mungo: Algorithm and implementation. Online Material (2020)
21. Google: Pytype, a static type analyzer for python code. Online Material (2023)
22. Hassan, M., Urban, C., Eilers, M., Müller, P.: MaxSMT-based type inference for Python 3. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018, Part II. LNCS, vol. 10982, pp. 12–19. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_2
23. Heidegger, P., Thiemann, P.: Recency types for analyzing scripting languages. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 200–224. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14107-2_10
24. Holkner, A., Harland, J.: Evaluating the dynamic behaviour of python applications. In: Proceedings of the Thirty-Second Australasian Conference on Computer Science, vol. 91, pp. 19–28 (2009)
25. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. (TOPLAS) **23**(3), 396–450 (2001)
26. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 238–255. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03237-0_17
27. Kaleba, S., Larose, O., Jones, R., Marr, S.: Who you gonna call: analyzing the run-time call-site behavior of ruby applications. In: Proceedings of the 18th ACM SIGPLAN International Symposium on Dynamic Languages, pp. 15–28 (2022)
28. Kent, A.M., Kempe, D., Tobin-Hochstadt, S.: Occurrence typing modulo theories. ACM SIGPLAN Not. **51**(6), 296–309 (2016)
29. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with Mungo and StMungo. In: Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, pp. 146–159 (2016)
30. Lehtosalo, J.: Optional static typing for python. Online Material (2023)
31. Lerner, B.S., Politz, J.G., Guha, A., Krishnamurthi, S.: TeJaS: retrofitting type systems for JavaScript. ACM SIGPLAN Not. **49**(2), 1–16 (2013)
32. Madhavan, R., Komondoor, R.: Null dereference verification via over-approximated weakest pre-conditions analysis. ACM Sigplan Not. **46**(10), 1033–1052 (2011)
33. Maia, E., Moreira, N., Reis, R.: A static type inference for python. Proc. DYLA **5**(1), 1 (2012)
34. McGugan, W.: Rich, a python library for rich text and beautiful formatting in the terminal. Online Material (2023)
35. Microsoft: Pyright, a static type checker for python. Online Material (2023)
36. Monat, R., Ouadjaout, A., Miné, A.: Static type analysis by abstract interpretation of python programs. In: 34th European Conference on Object-Oriented Programming (ECOOP 2020). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2020)
37. Oxhøj, N., Palsberg, J., Schwartzbach, M.I.: Making type inference practical. In: Madsen, O.L. (ed.) ECOOP 1992. LNCS, vol. 615, pp. 329–349. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0053045

38. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
39. Pilkiewicz, A., Pottier, F.: The essence of monotonic state. In: Proceedings of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation, pp. 73–86 (2011)
40. Rastogi, A., Swamy, N., Fournet, C., Bierman, G., Vekris, P.: Safe & efficient gradual typing for typescript. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 167–180 (2015)
41. Richards, G., Hammer, C., Burg, B., Vitek, J.: The eval that men do. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 52–78. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22655-7_4
42. Richards, G., Lebresne, S., Burg, B., Vitek, J.: An analysis of the dynamic behavior of JavaScript programs. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1–12 (2010)
43. Rondon, P.M., Kawaguchi, M., Jhala, R.: Low-level liquid types. ACM Sigplan Not. **45**(1), 131–144 (2010)
44. Salib, M.: Starkiller: a static type inferencer and compiler for Python. Ph.D. thesis, Massachusetts Institute of Technology (2004)
45. Smith, F., Walker, D., Morrisett, G.: Alias types. In: Smolka, G. (ed.) ESOP 2000. LNCS, vol. 1782, pp. 366–381. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46425-5_24
46. Strom, R.E., Yemini, S.: Typestate: a programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng. **SE-12**(1), 157–171 (1986). https://doi.org/10.1109/TSE.1986.6312929
47. Summers, A.J., Müller, P.: Freedom before commitment: a lightweight type system for object initialisation. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, pp. 1013–1032 (2011)
48. Tobin-Hochstadt, S., Felleisen, M.: Logical types for untyped languages. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, pp. 117–128 (2010)
49. Van Rossum, G., Drake, F.L., Jr.: The Python Language Reference. Python Software Foundation, Wilmington (2014)
50. Vekris, P., Cosman, B., Jhala, R.: Refinement types for typescript. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 310–325 (2016)
51. Vitousek, M.M., Kent, A.M., Siek, J.G., Baker, J.: Design and evaluation of gradual typing for python. In: Proceedings of the 10th ACM Symposium on Dynamic languages, pp. 45–56 (2014)
52. Wang, T., Smith, S.F.: Precise constraint-based type inference for Java. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 99–117. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45337-7_6
53. Wang, Y.: PySonar2: an advanced semantic indexer for python. Online Material (2019)
54. Wei, S., Xhakaj, F., Ryder, B.G.: Empirical study of the dynamic behavior of JavaScript objects. Softw. Pract. Exper. **46**(7), 867–889 (2016)
55. Zhao, T.: Polymorphic type inference for scripting languages with object extensions. In: Proceedings of the 7th Symposium on Dynamic Languages, pp. 37–50 (2011)