Static Type Analysis by Abstract Interpretation of Python Programs

Raphaël Monat [ID]
Sorbonne Université, CNRS, LIP6, Paris, France
raphael.monat@lip6.fr

Abdelraouf Ouadjaout [ID]
Sorbonne Université, CNRS, LIP6, Paris, France
abdelraouf.ouadjaout@lip6.fr

Antoine Miné [ID]
Sorbonne Université, CNRS, LIP6, Paris, France
Institut Universitaire de France, Paris, France
antoine.mine@lip6.fr

──── **Abstract** ────

Python is an increasingly popular dynamic programming language, particularly used in the scientific community and well-known for its powerful and permissive high-level syntax. Our work aims at detecting statically and automatically type errors. As these type errors are exceptions that can be caught later on, we precisely track all exceptions (raised or caught). We designed a static analysis by abstract interpretation able to infer the possible types of variables, taking into account the full control-flow. It handles both typing paradigms used in Python, nominal and structural, supports Python's object model, introspection operators allowing dynamic type testing, dynamic attribute addition, as well as exception handling. We present a flow- and context-sensitive analysis with special domains to support containers (such as lists) and infer type equalities (allowing it to express parametric polymorphism). The analysis is soundly derived by abstract interpretation from a concrete semantics of Python developed by Fromherz et al. Our analysis is designed in a modular way as a set of domains abstracting a concrete collecting semantics. It has been implemented into the MOPSA analysis framework, and leverages external type annotations from the Typeshed project to support the vast standard library. We show that it scales to benchmarks a few thousand lines long, and preliminary results show it is able to analyze a small real-life command-line utility called PathPicker. Compared to previous work, it is sound, while it keeps similar efficiency and precision.

1   Introduction

Sound static analyses for static languages, such as C and Java, are now widespread [5, 25, 11, 10, 33]. They have been particularly successful in the verification of critical embedded software, which use a subset of C and preclude complex control flow and dynamic memory allocation. The sound, efficient, and precise analysis of more dynamic languages and program traits remains a challenge.

Dynamic programming languages, such as JavaScript and Python, have become increasingly popular over the last years. Python currently ranks as the second most used programming language on Github.[1] It is appreciated for its powerful and permissive high-level syntax, e.g., it allows programmers to redefine all operators (addition, field access, etc.) in custom classes, and comes equipped with a vast standard library. Python is a highly dynamic language. It notably features:

1. *Dynamic typing:* variables in Python are neither statically declared nor typed. Any variable can hold a value of any type, and the type of the value held may change during program execution.
2. *Introspection:* programs can inspect the type of variables at run-time to alter their execution. Operators exist to support both nominal and structural types. Firstly, `isinstance(o, cls)` checks whether `o` has been instantiated from class `cls`, or a class inheriting from `cls`. For example, `isinstance("a", str)` returns `True`, while `isinstance(object(), str)` returns `False`. Secondly, `hasattr(o, attr)` checks whether `o` (or its class, or one of its parent class) has an attribute with name equal to the `attr` string. For instance, `hasattr(42, "__add__")` returns `True` because `int`, the class of `42`, has an addition method, named `"__add__"`. This kind of structural typing is so-called "duck typing". Python classes are first-class objects and can be stored into variables.
3. *Self-modification:* the structure of Python objects can be altered at run-time. For instance, it is possible to add attributes to an object or class at run-time, or remove them. It is also possible to create new classes at run-time.
4. *Eval:* it is possible to evaluate arbitrary string expressions as Python code at run-time with `eval` statements.

Due to dynamic typing (point 1), type errors are detected at run-time and cause `TypeError` exceptions, whereas such errors would be caught at compile time in a statically typed language. In this article, we propose a static analysis to infer type information, and use this information to detect all exceptions that can be raised and not caught. Our type analysis is flow-sensitive, to take into account the fact that variable types evolve during program execution and, conversely, run-time type information is used to alter the control-flow of the program, either through introspection or method and operator overloading (points 2 and 3). Moreover, it is context-sensitive as, without any type information on method entry, it is not possible to infer its control flow at all. However, handling `eval` is left as future work (possibly leveraging ideas proposed by [17] on JavaScript).

Motivating Example.   Consider the code from Fig. 1 (where `*` stands for a non-deterministic boolean). It defines a function `fspath`, taken from the standard library, with a parameter `p` as input. If `p` is an object instantiated from a class inheriting from `str` or `bytes`, it is returned. Otherwise, the function searches for an attribute called `fspath`, and calls it as a

---

[1] `https://octoverse.github.com/#top-languages`

```
1   def fspath(p):                   10      else: raise TypeError
2     sb = (str, bytes)              11
3     if isinstance(p, sb):          12   class Path:
4       return p                     13     def fspath(self):
5     if hasattr(p, "fspath"):       14       return 42
6       r = p.fspath()               15
7       if isinstance(r, sb):        16   if *: p = "/dev"
8         return r                   17   else: p = Path()
9       else: raise TypeError        18   r = fspath(p)
```

◼ **Figure 1** Motivating example.

method. If the return type is not an instance of `str` or `bytes`, an exception is raised. Thus, when `fspath` does not raise an error, it takes as input an instance of `str` or `bytes`, or an object having a method `fspath` returning either a string or a bytes-based string. In all cases, the return type of `fspath` is either `str` or `bytes`.

We model correct and erroneous calls to `fspath` in lines 12 to 18. In particular, we define a `Path` class, having a method `fspath` returning an integer, hence, a call to function `fspath` on an instance of `Path` would raise a `TypeError`. Our analysis is able to infer that, at the end of line 16, `p` is a string, and that at line 17, `p` is an instance of the `Path` class, which has a field `fspath` that can be called. It finds that, either `r` is a string, or a `TypeError` is raised.

As it is part of the standard library, the `fspath` function is particularly well-behaved: it does not make many implicit assumptions on the argument type (only that `p.fspath` is callable), but instead uses type information to handle different cases in different program branches. Nevertheless, the context is important to infer whether a specific call to `fspath` can raise a `TypeError` or not. A more typical Python programmer might replace lines 5 to 10 with a call to `return p.fspath()`, leaving implicit the fact that `p` should have a method `fspath` returning `str` or `bytes` chains. This is summarized in one of Python motos: "easier to ask for forgiveness than permission". Our analysis would correctly infer that invalid calls to that modified function would raise an `AttributeError` exception.

Static Analysis of Python. The sound analysis of Python programs is particularly challenging as it is a large and complex language. Python is not defined by a standard (formal or informal), but by its reference implementation, CPython. Fromherz et al. [13] introduced formally a concrete collecting semantics defined by structural induction on programs as a function manipulating reachable program states. We rely on a slight extension of this semantics. We then build a computable static analysis by abstract interpretation [9]. While [13] presented a value-analysis (employing in particular numeric domains), we introduce a more abstract analysis that only reasons on Python types. Similarly to [13], it tracks precisely the control-flow, including the flows of exceptions. We believe (and our preliminary experiments support) that flow-sensitive, context-sensitive type analysis for Python achieves a sweet-spot in term of efficiency versus usefulness. We believe that this level of precision is both necessary and, in practice, sufficient to infer the type-directed control flow (such as type testing or method dispatch) and infer exception propagation. Although our abstract domain tracks types precisely (including nominal types and attributes), it is nevertheless more scalable than a value analysis.

Relationship with Typing.   It is worth comparing our approach with classic typing. Statically typed languages ensure the absence of type-related errors through static type checking, possibly augmented with automatic type inference. However, while static typing rejects untypable programs, our analysis gives a semantics to such programs by propagating type errors as exceptions. This is important in order to support programs that perform run-time type errors and catch them afterwards, which is common-place in Python. Indeed, our goal is not to enforce a stricter, easier to check, way to program in Python,[2] but rather to check as-is programs with no uncaught exceptions. Secondly, static type checking is generally flow-insensitive and context-insensitive, trying to associate a unique type to each variable throughout program executions, while we use a flow- and context-sensitive analysis. Following classic abstract interpreters [5], our analysis is performed by structural induction on the syntax, starting from a main entry point. It is thus unable to analyze functions in isolation. While this kind of modularity is a highlight of typing algorithms, we believe that it is not well suited to Python. Consider, for instance, that a call to a function can alter the value, and so the type, of a global variable, which is difficult to express in a type system. Moreover, even a simple function, such as `def f(a, b):  return a + b`, has an unpredictable effect as the `+` operator can be overloaded to an arbitrary method by the programmer. We view type analysis as an instance of abstract interpretation, one which is slightly more abstract than classic value analysis. This view is not novel: [8] reconstructs Hindley-Milner typing rules as an abstract interpretation of the concrete semantics of the lambda calculus. One benefit of this unified view is the possibility to incorporate, in future work, some amount of value analysis through reduction. For instance, our analysis currently considers, to be sound, that any division can raise a `ZeroDivError`, which could be ruled out by a simple integer analysis. Finally, the correctness proof of our analysis is derived through a soundness theorem linking the concrete and the abstract semantics, in classic abstract interpretation form, and not by subject reduction. Both our analysis and type systems are conservative, but we replace the motto "well-typed programs cannot go wrong" with a guaranteed over-approximation of the possible (correct and incorrect) behaviors of the program.

Contributions.   Our contributions are as follow:

- We present a sound static type analysis by abstract interpretation that handles most of Python constructions. Compared to classic analyses targetting static languages, we believe the uniqueness and precision lies in the combination of domains, allowing the analyzer to soundly know both nominal and structural types of manipulated objects, as well as the raised exceptions.
- Our analysis is based on several abstract domains that are combined together in a reduced product. A first, non-relational domain tracks for each variable the set of its possible types. A second domain infers type equalities between variables, which can achieve a form of parametric polymorphism. A third domain analyzes containers (such as lists).
- We provide concretization functions for our domains as well as selected transfer functions (for the sake of concision). We use the concretization-only setting of abstract interpretation to define how our analysis is sound with respect to a concrete semantics based on [13]. Compared to previous work on dynamic languages, the formalization of the concretization functions parameterized by the recency abstraction is novel.

---

[2] In practice, we are nevertheless limited to programs that do not use `eval`.

- We have implemented our analysis modularly within the Mopsa framework [20], and showed that it can analyze real-world benchmarks with few false alarms and reasonable efficiency, using the non-relational analysis. Moreover, we have shown that it performs better than existing Python type analyses, and can handle soundly program traits that they cannot, including introspection and exception handling.
- In order for our analysis to scale, our analyzer is able to read stub files containing Pythonic type annotations. We reused the Typeshed project [37], containing annotations of the standard library. We analyze a small (3kLOC) real-life utility called PathPicker [43] using more than 12 modules from the standard library.

Limitations.   The two current limitations to the scalability of our approach are the standard library support, and the interprocedural analysis. The standard library is huge, and while some parts are written in Python others are written in C. We currently leverage the type annotations from Typeshed to support the standard library. We support most, but currently not all these annotations, and we add new Typeshed stub files when needed. The interprocedural analysis is based on inlining, which is costly but precise. It is improved with a cache in Section 6.2, and a more efficient interprocedural analysis is left as future work. The smashing-based abstraction of containers, combined with no information on the length of those containers creates spurious alarms. For example, in order to be sound, each list access raises an invalid access exception `IndexError`. The language support is wide enough to scale to programs a few thousand lines long, but still not complete. The analyzer does not support recursive functions for now, but the implementation would not be technically difficult (as loops and dynamic allocation are supported, respectively using a usual accelerated fixpoint computation – see Fig. 6 in [13] – and a recency abstraction [3]). It however detects recursive calls and stops at that point. Recursion is not used a lot in Python. In particular, there is no tail-call optimization, and the default recursion stack has depth 1000. Similarly, metaclasses are not supported but should not be technically difficult to implement. The `super` class and arbitrary code evaluation using `eval` are not supported. The latter feature is less used than in JavaScript (we never encountered it in our benchmarks), and solutions exist [17]. We could reuse [13] to handle generators. Asynchronous operators are not supported either [32].

Outline.   Section 2 starts by recalling and slightly improving on the formalization of Python semantics from Fromherz et al. [13]. Section 3 then presents a non-relational static type analysis. A relational type equality domain refines the previous analysis in Section 4. Section 5 presents the analysis of containers, with the example of lists. Section 6 discusses our implementation and presents experimental results. Section 7 discusses related works and Section 8 concludes.

2 Concrete Semantics of Python

We first discuss the concrete semantics of Python, as our static analysis is stated as a sound abstract interpretation of this semantics. This semantics is the implementation of Python's official interpreter, CPython (we focus on Python 3.7). It is thus not standardized nor formally defined. We rely on the reference manual and use CPython's behavior and source code as an oracle in case of doubt. We use a slight evolution of the semantics proposed in [13], where we have adapted the semantics to get closer to the real Python language: while builtin objects, such as integers, were considered as primitive values, they are now objects allocated on the heap, which allows the creation of classes inheriting from builtin classes. We start by defining the memory state on which a Python program acts, and then describe a few parts of the concrete semantics of Python.

$$\mathbf{Id} \subseteq string$$

$$\mathcal{E} \stackrel{\text{def}}{=} \mathbf{Id} \rightharpoonup \mathbf{Addr} \cup \texttt{LocalErr}$$

$$\mathcal{H} \stackrel{\text{def}}{=} \mathbf{Addr} \rightharpoonup \mathbf{ObjN} \times \mathbf{ObjS}$$

$$\mathcal{F} \stackrel{\text{def}}{=} \{\ cur, ret, brk,$$

$$cont, exn\ a, a \in \mathbf{Addr}\ \}$$

$$\mathbf{Addr} \stackrel{\text{def}}{=} \mathbf{Location} \times \mathbb{N}$$

$$\mathbf{ObjN} \stackrel{\text{def}}{=} \texttt{int}(a \in \mathbf{Addr}, i \in \mathbb{Z}) \cup \texttt{bool}(b)$$

$$\cup\ string(a, s \in string) \cup \texttt{None} \cup \texttt{NotImpl}$$

$$\cup\ \mathbf{List}(a, ls), ls \in \mathbf{Addr}^* \cup \mathbf{Method}(a, f)$$

$$\cup\ \mathbf{Fun}(f) \cup \mathbf{Class}(c) \cup \mathbf{Instance}(a \in \mathbf{Addr})$$

$$\mathbf{ObjS} \stackrel{\text{def}}{=} string \rightharpoonup \mathbf{Addr}$$

Figure 2 Concrete semantic domains.

2.1 Concrete Semantic Domain

The memory state consists in two parts: the environment $\mathcal{E}$ and the heap $\mathcal{H}$, defined in Fig. 2. The environment $\mathcal{E}$ is a partial, finite map from variable identifiers **Id** to addresses **Addr**. The set **Addr** of heap addresses is infinite. To simplify the definition of our semantics and its abstraction, we assume that, up to isomorphism, **Addr** has the form **Location** $\times \mathbb{N}$ where **Location** is the set of program locations (in the following, a line number). We use the notation $@(l, n) \in \mathbf{Addr}$ to designate unambiguously the n-th address allocated at program location l. Due to scoping rules in Python, local variables may be locally undefined. We denote this using an additional value for local variables, called `LocalErr`, as using such undefined variables results in an `UnboundLocalError` exception. The heap $\mathcal{H}$ maps addresses to objects defined by their nominal and structural information. The nominal part gives the class and the value of the object, while the structural one gives for each attribute its allocation address.

Although everything is object in Python, we distinguish builtin objects from the other ones. Primitive objects include integers, strings, booleans, `None`, `NotImpl` and lists (storing the addresses of each of its elements). Other containers such as dictionaries, sets, and tuples are handled similarly. These builtin objects are kept separate from the user-defined classes as their implementation uses low-level fields only accessible by CPython, which are hidden by the semantics. Some builtin objects (integers, strings, and lists only) depend on an address, being the class from which they are instantiated. This allows to handle classes inheriting from builtins: for example, if `A` inherits from the `int` class, `A(10)` is represented as $\texttt{int}(@, 10)$, where @ points to the class of A. This behavior was not expressible in the previous concrete semantics [13]. In the following, the notation $\texttt{int}(10)$ denotes $\texttt{int}(@, 10)$, where @ points to the class of integers (which happen in most cases), and similarly for other builtin objects. Methods bind an instance address and a function. Instances are defined by the address of the class from which they are instantiated. Classes and functions are also objects (we do not detail their inner structure). Concerning the structural part, a finite number of attributes may be added to classes, functions, and instances, so we additionally keep a map from attribute names to addresses for those objects.

Environment and Heap Example.   Consider the code of Fig. 3. We start from an initial, empty state $(\emptyset, \emptyset)$, and show how the state of the environment e and heap h are *extended* after each step in Fig. 4. After the class declaration at lines 1 to 5, the identifier `A` refers to the eponymous class, which is allocated at line 1, and is the first allocated object there, hence it has the address $@(1, 0)$. The instance of `A` created at line 6 has an attribute `val` being

```
1  class A:                          6   x = A()
2    def __init__(self):            7   y = x.val
3      self.update(0)               8   z = x
4    def update(self, x):           9   z.update('a')
5      self.val = x * 2             10  if *: x.atr= 'b'
```

Figure 3 Mutating objects: an example.

Line	$e \in \mathcal{E}$	$h \in \mathcal{H}$
5	$A \mapsto @(1,0)$	$@(1,0) \mapsto (\textbf{Class } A, \{ \_\_\_init\_\_\_ \mapsto @(2,0), \text{update} \mapsto @(4,0) \})$
		$@(2,0) \mapsto (\textbf{Fun}(\_\_\_init\_\_\_), \emptyset), @(4,0) \mapsto (\textbf{Fun}(\_\_\_update\_\_\_), \emptyset)$
6	$x \mapsto @(6,0)$	$@(6,0) \mapsto (\text{Instance } @(1,0), \text{val} \mapsto @(5,0))$
		$@(5,0) \mapsto (\texttt{int } 0, \emptyset)$
7	$y \mapsto @(5,0)$	
8	$z \mapsto @(6,0)$	
9		$@(6,0) \mapsto (\text{Instance } @(1,0), \textbf{val} \mapsto \textbf{@(5,1)})$
		$@(5,1) \mapsto (string \ 'aa', \emptyset)$
10		Two possible heaps: either the heap from line 9, or:
		$@(6,0) \mapsto (\text{Instance } @(1,0), \text{val} \mapsto @(5,1), \text{atr} \mapsto @(10,0))$
		$@(10,0) \mapsto (string \ 'b', \emptyset)$

Figure 4 Mutating objects example: state evolution.

the integer 0; it is created during the initialization of the class. After line 8, y maps to the integer stored in the field x of the instance of A. As z points to the same instance of A as x, we only need to add a binding between z and the instance's address. At line 9, the instance is updated using a method call, and the field val now maps to the string 'aa'. After line 10, we have two possible heaps, depending on the non-deterministic choice *: either the previous heap, or a heap extended to add atr to the instance of A.

Flow tokens. Following [13], we present our semantic as a function from a set of states in precondition to a set of states in postcondition by induction on the program syntax. To provide a semantics for operations that do not return immediately, such as raise, we use continuations: we label states using *flow tokens* (elements of $\mathcal{F}$), so the states we consider are in $\mathcal{P}(\mathcal{F} \times \mathcal{E} \times \mathcal{H})$. Flow token *cur* represents the current flow on which all instructions that do not disrupt the control flow operate (e.g., assignments, but not raise or return). *ret* collects the set of states given by a return statement, while *brk*, *cont*, *exn* perform similar collections for the break, continue, raise statements, respectively. Each exception will be kept in a separate flow, so *exn* is indexed by the address of the exception object.

2.2 Semantics of Expressions and Statements

We denote by $\mathbb{E}[\![e]\!]$ the semantics of expression e. This semantics has the following signature: $\mathbb{E}[\![e]\!] : \mathcal{F} \times \mathcal{E} \times \mathcal{H} \to \mathcal{F} \times \mathcal{E} \times \mathcal{H} \times (\textbf{Addr} \cup \{\bot\})$, so $\mathbb{E}[\![e]\!]$ returns the address where the object associated with e is stored in the heap (or $\bot$ if an exception is raised and no address is returned). The semantics of statements is written $\mathbb{S}[\![s]\!]$ and has signature: $\mathcal{F} \times \mathcal{E} \times \mathcal{H} \to \mathcal{F} \times \mathcal{E} \times \mathcal{H}$. We do not describe the whole semantics of Python: we cover a few cases that are of interest for the upcoming type analysis; some other cases are described in [13] (the addition operator, conditionals, while loops and generators).

$$\mathbb{E}[\![\,id\,]\!]\,(f,e,h) \stackrel{\text{def}}{=}$$

$\quad$ if $f \neq cur$ then $(f,e,h), \perp$ else

$\quad$ if $id \notin \operatorname{dom} e$ then $\mathbb{S}[\![\,\texttt{raise NameError}\,]\!]\,(f,e,h), \perp$ else

$\quad$ if $e(id) = \texttt{LocalErr}$ then $\mathbb{S}[\![\,\texttt{raise UnboundLocalError}\,]\!]\,(f,e,h), \perp$ else

$\quad (f,e,h), e(id)$

$$\mathbb{S}[\![\,id = e\,]\!]\,(f,e,h) \stackrel{\text{def}}{=}$$

$\quad$ let $(f,e,h), @ = \mathbb{E}[\![\,e\,]\!]\,(f,e,h)$ in

$\quad$ if $f \neq cur$ then (f,e,h) else $(cur, e[id \mapsto @], h)$

■ **Figure 5** Concrete semantics of variable evaluation and assignment.

Variable Evaluation and Assignment.   We define the semantics of variable evaluation and assignment in Fig. 5. To evaluate a variable identifier given an input state, we first check that the flow token is cur: otherwise, the state is returned with $\perp$ instead of an address. Then, two errors may happen: if the variable is not defined in the program at all, a `NameError` is raised. If the variable is not defined in the current scope but defined somewhere else in the program, an `UnboundLocalError` is raised. Otherwise, the variable is evaluated into an address, and both the state and the address are returned.

To assign e to id, Python first starts by evaluating e: if other flows are created – for example if an exception is raised during the evaluation – they are just returned, and the assignment takes place only in the current flow. As most of the time, we want to update only the current flow, we introduce the following notation "letcur $(f,e,h), @ = e_1$ in e_2" which unfolds into "let $(f,e,h), @ = e_1$ in if $f \neq cur$ then $(f,e,h), @$ else e_2".

Semantics of Attribute Accesses.   We show how to access attribute s of expression e in Fig. 6. This is a formalization of one of Python's complex behaviors, shown here to illustrate the complexity of the language. $e.s$ dispatches the attribute access to a method being either `__getattribute__` or `__getattr__`. The first method call usually ends up being `object.__getattribute__`. It can be overloaded by any class, which is supported in our implementation, but not very common, so we describe only the semantics of `object.__getattribute__`.

To evaluate `object.__getattribute__`(e, s), we start by evaluating both e and s, which return object addresses: respectively $@_e$ and $@_s$. If s is not a string, a type error is raised. Then, we search for s in the parents class[3] of the allocated object $h(@_e)$ using the function *mrosearch* (which takes as input a class and a string, and returns a parent class of the input, or $\perp$ if the search is unsuccessful). The function *type* returns the class from which the object given in argument has been instantiated. If no class is found, we search for the attribute in the object only, using the low-level *has_field* and *get_field* operators. *get_field* is a low-level version of attribute access: it searches for the field locally, in the object structure, but it will not recursively search in the object's class (nor its parents). It takes as input an object and

---

[3] Actually, we search for s in the MRO of $h(@_e)$. The MRO of a class `c` is a list of classes from which `c` inherits, starting from its closest parents, to its most ancient one (usually, `object`). Even if multiple inheritances induce a direct acyclic graph, the inheritance relationship is linearized (using the algorithm described in [4]).

$$\mathbb{E}[\![\,\texttt{object.\_\_getattribute\_\_}(e,s)\,]\!]\,(f,e,h)\stackrel{\mathrm{def}}{=}$$

$\quad$letcur $(f,e,h),@_e = \mathbb{E}[\![\,e\,]\!]\,(f,e,h)$ in

$\quad$letcur $(f,e,h),@_s = \mathbb{E}[\![\,s\,]\!]\,(f,e,h)$ in

$\quad$if $isinstance(h(@_s),\mathrm{str})$ then

$\qquad$let $string\ s = h(@_s)$ in

$\qquad$let $c = mrosearch(type(h(@_e)),s)$ in

$\qquad$if $c \neq \bot$ then

$\qquad\quad$let $cs = h(get\_field(c,s))$ in

$\qquad\quad$if $has\_field(cs,"\_\_get\_\_") \wedge has\_field(cs,"\_\_set\_\_")$ then

$\qquad\qquad\mathbb{E}[\![\,get\_field(cs,"\_\_get\_\_")(cs,h(@_e),type(h(@_e)))\,]\!]\,(f,e,h)$

$\qquad\quad$else if $has\_field(h(@_e),s)$ then $\mathbb{E}[\![\,get\_field(h(@_e),s)\,]\!]\,(f,e,h)$

$\qquad\quad$else $\mathbb{E}[\![\,get\_field(c,s)\,]\!]\,(f,e,h)$

$\qquad$else if $has\_field(h(@_e),s)$ then $\mathbb{E}[\![\,get\_field(h(@_e),s)\,]\!]\,(f,e,h)$

$\qquad\quad$else $\mathbb{S}[\![\,\texttt{raise AttributeError}\,]\!]\,(f,e,h),\bot$

$\quad$else $\mathbb{S}[\![\,\texttt{raise TypeError}\,]\!]\,(f,e,h),\bot$

■ Figure 6 Concrete semantics of attribute access.

a string and returns the address of the field defined by its arguments. Similarly, *has_field* checks for the presence of a field at the object structure only (contrary to `hasattr`). If a class c is found, let us denote by cs the object corresponding to the access of s in c. If cs is a data descriptor (meaning it has fields `__get__` and `__set__`), the result is the evaluation of cs's `__get__` method. Otherwise, we return the attribute at e's level (if it exists), or at c's level otherwise.

Semantics of Exceptions.   Finally, we showcase the use of multiple flows by defining the semantics of exceptions in Fig. 7. To raise an expression e, we evaluate e and check that it is an instance of the `BaseException` class. In that case, we change the flow token to *exn*, parameterized by the evaluation of e, and return the environment and heap. If not, we raise a type error. Now, let us consider the exception-catching mechanism. It behaves as follow: `tbody` is evaluated; if no exception is raised the evaluation is finished. Otherwise, if an exception (stored at address $@_{raised}$) is raised during this evaluation, we evaluate `exn`: if it is an exception (i.e, if it inherits from `BaseException`), and if the raised exception is an instance of `exn`, we evaluate `texc`, where the variable `v` is now bound to the raised exception (until the end of the evaluation of `texc`). Otherwise, we let the exception escape this control block.

3 A Non-relational Static Type Analysis

We present a non-relational type abstraction of Python semantics. The abstraction can infer the nominal and structural types of Python values in a flow-sensitive way, i.e., the types of a variable can vary from one program location to another. It is context-sensitive, supports object mutability and aliasing. For concision, we limit here the presentation to atomic types, such as numbers, strings and instances of user-defined classes. This abstraction will be completed with a type relation analysis in Section 4, and extended to containers in Section 5.

$\mathbb{S}[\![\,\mathtt{raise}\ e\,]\!]\,(f, e, h) =$

  $\text{letcur}\ (f, e, h, @) =\ \mathbb{E}[\![\,e()\,]\!]\,(f, e, h)\ \text{in}$

  $\text{if}\ \mathit{isinstance}(h(@), \mathtt{BaseException})\ \text{then}\ (\mathit{exn}\ @, e, h)$

  $\text{else}\ \ \mathbb{S}[\![\,\mathtt{raise\ TypeError}\,]\!]\,(f, e, h)$

$\mathbb{S}[\![\,\mathtt{try}:\ \mathtt{tbody\ except\ exn\ as\ v}:\ \mathtt{texc}\,]\!]\,(f, e, h) =$

  $\text{let}\ (f, e, h) =\ \mathbb{S}[\![\,\mathit{tbody}\,]\!]\,(f, e, h)\ \text{in}$

  $\text{if}\ f \neq \mathit{exn}\ \_\ \text{then}\ (f, e, h)$

  $\text{else let}\ \mathit{exn}\ @_{\mathit{raised}} = f\ \text{in}$

    $\text{letcur}\ (f, e, h), @_{\mathit{exn}} =\ \mathbb{E}[\![\,\mathit{exn}\,]\!]\,(\mathit{cur}, e, h)\ \text{in}$

    $\text{if}\ \neg \mathit{issubclass}(h(@_{\mathit{exn}}), \mathtt{BaseException})\ \text{then}$

      $\mathbb{S}[\![\,\mathtt{raise\ TypeError}\,]\!]\,(\mathit{cur}, e, h)$

    $\text{else if}\ \mathit{isinstance}(h(@_{\mathit{raised}}), h(@_{\mathit{exn}}))\ \text{then}$

      $\text{let}\ (f, e, h) =\ \mathbb{S}[\![\,\mathtt{texc}\,]\!]\,(\mathit{cur}, e[\mathtt{v} \mapsto @_{\mathit{raised}}], h)\ \text{in}\ (f, e \setminus \{\,\mathtt{v}\,\}, h)$

    $\text{else}\ (f, e, h)$

Figure 7 Concrete semantics of exceptions.

3.1 Abstract Domain

The structure of the abstract domain $\mathcal{D}^{\sharp}$ is defined in Fig. 8, along with the concretization functions, giving concrete meaning to the abstract states in Fig. 9. It is decomposed into three parts: a recency abstraction of allocated addresses, an environment abstraction, and a heap abstraction, all explained below. The example from the concrete semantics section is revisited in Sec. 3.2.

Recency Abstraction. To over-approximate a set of concrete addresses, we use a recency abstraction, as introduced in [3]. This abstraction maintains two kinds of information about allocated addresses. Firstly, the allocation site is preserved in order to distinguish between allocations at different program locations. Secondly, allocations at a same location $l \in \mathbf{Location}$ are partitioned into two abstract addresses via a recency criterion: *the* most recent allocation is represented with the abstract address $@^{\sharp}(l, \mathbf{r})$, while *all* the previous ones are abstracted by a unique abstract address $@^{\sharp}(l, \mathbf{o})$. The concretization function $\gamma_{\mathrm{recency}}$ takes as input the set of abstract addresses currently defined ($\rho \in \mathcal{P}(\mathbf{Addr}^{\sharp})$), and yields a set of address abstraction functions, giving concrete meaning to abstract addresses, satisfying the conditions mentioned above. As addresses play an important part in every part of the abstract states, the other concretization operators are parameterized by an address abstraction $\alpha_{\mathbf{Addr}}$; we denote this parameterization with the following notation: $\gamma[\alpha_{\mathbf{Addr}}]$.

    The allocation of a new abstract address is handled by the auxiliary function `alloc_addr`:

$\mathbb{E}^{\#}[\![\,\mathtt{alloc\_addr}(l)\,]\!]\,(\varphi, \rho \in \mathcal{P}(\mathbf{Addr}^{\sharp}), \epsilon, \eta) =$

  $\text{if}\ @^{\sharp}(l, \boldsymbol{r}) \in \rho\ \text{then}\ @^{\sharp}(l, \boldsymbol{r}), \mathbb{S}^{\#}[\![\,@^{\sharp}(l, \boldsymbol{o}) \overset{\mathrm{weak}}{=} @^{\sharp}(l, \boldsymbol{r})\,]\!]\,(\varphi, \rho, \epsilon, \eta)$

  $\text{else}\ @^{\sharp}(l, \boldsymbol{r}), (\varphi, \rho \cup \{\,@^{\sharp}(l, \boldsymbol{r})\,\}, \epsilon, \eta)$

$$\mathbf{Addr}^\sharp \stackrel{\text{def}}{=} \mathbf{Location} \times \{\, \mathbf{r}, \mathbf{o} \,\}$$

$$\mathcal{F}^\sharp \stackrel{\text{def}}{=} \{\, cur, ret, brk, cont, exn \ @^\sharp,$$
$$@^\sharp \in \mathbf{Addr}^\sharp \,\}$$

$$\mathcal{E}^\sharp \stackrel{\text{def}}{=} \mathbf{Id} \rightharpoonup \mathcal{P}(\mathbf{Addr}^\sharp \cup \{\, \texttt{LocalErr} \,\})$$

$$\mathcal{H}^\sharp \stackrel{\text{def}}{=} \mathbf{Addr}^\sharp \rightharpoonup \mathcal{P}(\mathbf{ObjN}^\sharp \times \mathbf{ObjS}^\sharp)$$

$$\mathcal{D}^\sharp \stackrel{\text{def}}{=} \mathcal{F}^\sharp \rightharpoonup (\mathcal{P}(\mathbf{Addr}^\sharp) \times \mathcal{E}^\sharp \times \mathcal{H}^\sharp)$$

$$\mathbf{ObjN}^\sharp \stackrel{\text{def}}{=} \mathbf{AInt}(a) \cup \mathbf{AStr}(a)$$
$$\cup \ \mathbf{AMethod}(a, f)$$
$$\cup \ \mathbf{AClass}(c) \cup \mathbf{AFun}(f)$$
$$\cup \ \mathbf{AInst}(a), a \in \mathbf{Addr}^\sharp$$

$$\mathbf{ObjS}^\sharp \stackrel{\text{def}}{=} \{\, \top \,\} \cup$$
$$\big(\mathcal{P}(string) \times (string \rightharpoonup \mathbf{Addr}^\sharp) \big)$$

■ **Figure 8** Definition of abstract states.

$$\gamma_{\text{recency}}(\rho \in \mathcal{P}(\mathbf{Addr}^\sharp)) = \{\, \alpha_{\mathbf{Addr}} : \mathbf{Addr} \rightharpoonup \mathbf{Addr}^\sharp \mid$$
$$\big(@^\sharp(l, o) \in \rho \implies \exists m \in \mathbb{N}, \forall i \leq m, \alpha_{\mathbf{Addr}}(@(l, i)) = @^\sharp(l, o) \big) \wedge$$
$$\big(@^\sharp(l, r) \in \rho \implies \exists n \in \mathbb{N}, \big(\alpha_{\mathbf{Addr}}^{-1}(@^\sharp(l, r)) = \{\, @(l, n) \,\}$$
$$\wedge\, n = 1 + \max\{\, i \mid \alpha_{\mathbf{Addr}}(@(l, i)) = @^\sharp(l, o) \,\} \big) \big) \,\}$$

$$\gamma_{\mathcal{E}}[\alpha_{\mathbf{Addr}}](\epsilon \in \mathcal{E}^\sharp) = \{\, e \in \mathcal{E} \mid \forall v \in \text{dom } \epsilon, \alpha_{\mathbf{Addr}}(e(v)) \in \epsilon(v)$$
$$\vee\, \texttt{LocalErr} \in \epsilon(v) \implies e(id) = \texttt{LocalErr} \,\}$$

$$\gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](\mathbf{AInt}(@^\sharp)) = \{\, int(@, i) \mid i \in \mathbb{Z}, \alpha_{\mathbf{Addr}}(@) = @^\sharp \,\}$$
$$\gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](\mathbf{AStr}(@^\sharp)) = \{\, string(@, s) \mid s \in \mathcal{P}(str), \alpha_{\mathbf{Addr}}(@) = @^\sharp \,\}$$
$$\gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](\mathbf{AMethod}(@^\sharp, f)) = \{\, \mathbf{Method}(@, f) \mid \alpha_{\mathbf{Addr}}(@) = @^\sharp \,\}$$
$$\gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](\mathbf{AClass}(c)) = \{\, \mathbf{Class}(c) \,\}$$
$$\gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](\mathbf{AFun}(f)) = \{\, \mathbf{Fun}(f) \,\}$$
$$\gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](\mathbf{AInst}(@^\sharp)) = \{\, \mathbf{Instance}(@) \mid \alpha_{\mathbf{Addr}}(@) = @^\sharp \,\}$$

$$\gamma_{\mathbf{ObjS}}[\alpha_{\mathbf{Addr}}](\mu \in \mathcal{P}(string), f^\sharp \in string \rightharpoonup \mathbf{Addr}^\sharp) = \{\, f \in \mathbf{ObjS} = string \rightharpoonup \mathbf{Addr} \mid$$
$$\mu \subseteq \text{dom } f \subseteq \text{dom } f^\sharp \wedge \forall s \in \text{dom } f, \alpha_{\mathbf{Addr}}(f(s)) = f^\sharp(s) \,\}$$
$$\gamma_{\mathbf{ObjS}}[\alpha_{\mathbf{Addr}}](\top) = \mathbf{ObjS}$$

$$\gamma_{\mathcal{H}}[\alpha_{\mathbf{Addr}}](\eta \in \mathcal{H}^\sharp) = \{\, h \in \mathcal{H} \mid \forall @ \in \text{dom } \alpha_{\mathbf{Addr}}, h(@) = (n, s)$$
$$\wedge \big(\exists(n^\sharp, s^\sharp) \in \eta(\alpha_{\mathbf{Addr}}(@)), n \in \gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](n^\sharp) \wedge s \in \gamma_{\mathbf{ObjS}}[\alpha_{\mathbf{Addr}}](s^\sharp) \big) \,\}$$

$$\gamma_{\mathcal{F}}[\alpha_{\mathbf{Addr}}](exn \ @^\sharp) = \{\, exn \ @ \mid \alpha_{\mathbf{Addr}}(@) = @^\sharp \,\}$$
$$\gamma_{\mathcal{F}}[\alpha_{\mathbf{Addr}}](f \in \mathcal{F}^\sharp, f \neq exn \ \_) = \{\, f \,\}$$

$$\gamma(\varphi \in \mathcal{F}^\sharp, (\rho \in \mathcal{P}(\mathbf{Addr}^\sharp), \epsilon \in \mathcal{E}^\sharp, \eta \in \mathcal{H}^\sharp)) = \{\, (f, e, h) \in \mathcal{F} \times \mathcal{E} \times \mathcal{H} \mid$$
$$\alpha_{\mathbf{Addr}} \in \gamma_{\text{recency}}(\rho) \wedge f \in \gamma_{\mathcal{F}}[\alpha_{\mathbf{Addr}}](\varphi) \wedge e \in \gamma_{\mathcal{E}}[\alpha_{\mathbf{Addr}}](\epsilon) \wedge h \in \gamma_{\mathcal{H}}[\alpha_{\mathbf{Addr}}](\eta) \,\}$$
$$\gamma_{\mathcal{D}}(\delta \in \mathcal{D}^\sharp) = \bigcup_{\varphi \in \text{dom } \delta} \gamma(\varphi, \delta(\varphi))$$

■ **Figure 9** Concretization of the abstract states.

The semantics of `alloc_addr` is as follows. Given an allocation site $l \in \mathbf{Location}$, the function searches for the most recent allocation at the same location. If such an address $@^{\sharp}(l, \boldsymbol{r})$ exists, it should be moved to the pool of old addresses by copying its contents to the address $@^{\sharp}(l, \boldsymbol{o})$, which is done using a weak update $\mathbb{S}^{\#} [\![@^{\sharp}(l, \boldsymbol{o}) \overset{\text{weak}}{=} @^{\sharp}(l, \boldsymbol{r})]\!]$. Otherwise, the state is extended with the new address $@^{\sharp}(l, \boldsymbol{r})$. In both cases, the newly allocated abstract address is $@^{\sharp}(l, \boldsymbol{r})$.

Environment Abstraction.   The domain of abstract environments $\mathcal{E}^{\sharp}$ maintains a non-relational map binding identifiers to a set of abstract addresses, with concretization $\gamma_{\mathcal{E}}[\alpha_{\mathbf{Addr}}]$. To support Python scoping, variables can also point to `LocalErr` to represent variables that can be locally undefined.

Heap Abstraction.   The domain of abstract heaps $\mathcal{H}^{\sharp}$ provides the Python objects associated to the addresses of the environment. Python objects are approximated by a nominal type abstraction $\mathbf{ObjN}^{\sharp}$ and a structural type abstraction $\mathbf{ObjS}^{\sharp}$.

The nominal part keeps only the class information of the object and forgets about its value. The concretization $\gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}]$ maps abstract addresses to concrete ones. In particular, abstract instances of built-in values (integers, strings, booleans, `None`, `NotImpl`) are concretized into the set of corresponding built-in values (along with the class from which they were instantiated, for strings and integers, to support inheriting from these builtin classes). For instance: $\gamma_{\mathbf{ObjN}}[\alpha_{\mathbf{Addr}}](\mathbf{AInt}(@^{\sharp}_{\mathbf{Class}\ int})) = \{\, \mathtt{int}(@_{\mathbf{Class\ int}}, i) \mid i \in \mathbb{Z} \,\}$, where the addresses subscripted by $\mathbf{Class\ int}$ represent the address of the built-in integer class. Similarly to the concrete, $\mathbf{AInt}$ implicitly means $\mathbf{AInt}(@^{\sharp}_{\mathbf{Class\ int}})$. `None` and `NotImpl` are abstracted as instances of their respective classes. The body of functions, methods and classes is not abstracted.

The structural part stores a map over-approximating the addresses referenced by the attributes of the object. Attributes may be added in some execution traces and not in others, hence, the map actually maintains the set of attributes that may exist at a given program point for all possible executions. We complement this map with a finite set under-approximating the set of attributes that are definitely present. This information is important to avoid raising spurious `AttributeError` exceptions for attributes that are definitely present. These properties are formally defined by the concretization $\gamma_{\mathbf{ObjS}}[\alpha_{\mathbf{Addr}}]$. The structural type abstraction may also be approximated as $\top$ by the widening, in order to avoid having an infinite number of attributes being added to an instance.

Then, the concretization $\gamma_{\mathcal{H}}[\alpha_{\mathbf{Addr}}]$ of an abstract heap η is the set of heaps h such that each address $@$ is bound to an object (n, s) where: n is a concretization of $n^{\sharp}$, and s a concretization of $s^{\sharp}$, and the abstract object $(n^{\sharp}, s^{\sharp})$ is in the abstract heap at the abstract address $\alpha_{\mathbf{Addr}}(@)$. Note that the domain of the heap is defined by the recency abstraction, i.e. $\text{dom}\ \eta = \rho$.

Full State Abstraction.   Flow tokens are also abstracted: only the exception token *exn* changes between the concrete and the abstract, to store an abstract address instead of a concrete one. This is formally described in the concretization $\gamma_{\mathcal{F}}[\alpha_{\mathbf{Addr}}]$. A whole abstract state maps flow tokens to abstract environments and heaps. To concretize a whole abstract state $\delta \in \mathcal{D}^{\sharp}$ using $\gamma_{\mathcal{D}}$, we concretize each image of δ separately and join the resulting concrete states. To concretize an element $\delta(\varphi) = (\rho, \epsilon, \eta)$ using γ, we first fix an address abstraction $\alpha_{\mathbf{Addr}}$ using the concretization of the recency abstraction ρ. Then, each part (flow token φ, environment ϵ, heap η) is concretized separately.

Line	$\epsilon \in \mathcal{E}^\sharp$	$\eta \in \mathcal{H}^\sharp$
5	$A \mapsto \{\,@^\sharp(1, \boldsymbol{r})\,\}$	$@^\sharp(1, \boldsymbol{r}) \mapsto \{\mathbf{AClass}\ A, \{\,\_\_\_\mathrm{init}\_\_\_, \mathrm{update}\,\}, \_\_\_\mathrm{init}\_\_\_ \mapsto @^\sharp(2, \boldsymbol{r})$ $\wedge\ \mathrm{update} \mapsto @^\sharp(4, \boldsymbol{r})\};$ $@^\sharp(2, \boldsymbol{r}) \mapsto \{\,\mathbf{AFun}(\_\_\_\mathrm{init}\_\_\_), \emptyset, \emptyset\,\}; @^\sharp(4, \boldsymbol{r}) \mapsto \{\,\mathbf{AFun}(\mathrm{update}), \emptyset, \emptyset\,\}$
6	$x \mapsto \{\,@^\sharp(6, \boldsymbol{r})\,\}$	$@^\sharp(6, \boldsymbol{r}) \mapsto \{\mathbf{AInst}\ @^\sharp(1, \boldsymbol{r}), \{\mathrm{val}\}, \mathrm{val} \mapsto @^\sharp(5, \boldsymbol{r})\}$ $@^\sharp(5, \boldsymbol{r}) \mapsto \{\mathbf{AInt}, \emptyset, \emptyset\}$
7	$y \mapsto \{\,@^\sharp(5, \boldsymbol{r})\,\}$	
8	$z \mapsto \{\,@^\sharp(6, \boldsymbol{r})\,\}$	
9	$y \mapsto \{\,@^\sharp(5, \boldsymbol{o})\,\}$	$@^\sharp(5, \boldsymbol{r}) \mapsto \{\mathbf{AStr}, \emptyset, \emptyset\}$ $@^\sharp(5, \boldsymbol{o}) \mapsto \{\mathbf{AInt}, \emptyset, \emptyset\}$
10		$@^\sharp(6, \boldsymbol{r}) \mapsto \{\mathbf{AInst}\ @^\sharp(1, \boldsymbol{r}), \{\mathrm{val}\}, \mathrm{val} \mapsto @^\sharp(5, \boldsymbol{r}) \wedge \mathrm{atr} \mapsto @^\sharp(10, \boldsymbol{r})\}$ $@^\sharp(10, \boldsymbol{r}) \mapsto \{\mathbf{AStr}, \emptyset, \emptyset\}$

Figure 10 Evolution of the abstract states of the example from Fig. 3.

3.2 Example

To illustrate our abstraction, let us consider the example shown previously in Fig. 3. We summarize the evolution of the abstract state in Fig. 10, for the current flow *cur*. After the declaration of the class A, the variable x is assigned the address $@^\sharp(6, \boldsymbol{r})$, representing the instance of A allocated at line 6 and having a unique attribute val. This attribute points to the address $@^\sharp(5, \boldsymbol{r})$, representing the integer result of the multiplication at line 5. After assigning the addresses $@^\sharp(5, \boldsymbol{r})$ and $@^\sharp(6, \boldsymbol{r})$ to y and z respectively (which changes the environment only), the call to z.update($'$a$'$) leads to two changes. Firstly, during the evaluation of x * 2 at line 5, a new address is allocated for the resulting string. Since the address $@^\sharp(5, \boldsymbol{r})$ already exists, it is renamed to $@^\sharp(5, \boldsymbol{o})$ to denote that it is no longer the most recent allocation. Consequently, the variable y now points to $@^\sharp(5, \boldsymbol{o})$ and the object pointed by this address remains an integer **AInt**. The second change affects the heap to ensure that the most recent allocation $@^\sharp(5, \boldsymbol{r})$ points now to a string object **AStr**. After line 10, a new string is allocated and assigned to the attribute atr belonging to the address $@^\sharp(6, \boldsymbol{r})$. Since this change is performed in only one branch of the if statement, atr is not added to the under-approximation of attributes.

We illustrate our concretization by linking the final abstract state, at line 10 in Fig. 10, to the concrete one in Fig. 4. In the abstract, $\rho = \{\,@^\sharp(1, \boldsymbol{r}), @^\sharp(5, \boldsymbol{o}), @^\sharp(5, \boldsymbol{r}), @^\sharp(6, \boldsymbol{r}), @^\sharp(10, \boldsymbol{r})\,\}$. Let us define $\alpha^{\mathrm{ex}}_{\mathbf{Addr}} = @(1, 0) \mapsto @^\sharp(1, \boldsymbol{r}); @(6, 0) \mapsto @^\sharp(6, \boldsymbol{r}); @(5, 0) \mapsto @^\sharp(5, \boldsymbol{o}); @(5, 1) \mapsto @^\sharp(5, \boldsymbol{r}); @(10, 0) \mapsto @^\sharp(10, \boldsymbol{r})$. We can check that $\alpha^{\mathrm{ex}}_{\mathbf{Addr}}$ is *one* of the abstraction functions defined in $\gamma_{\mathrm{recency}}(\rho)$. In addition, it is *the* abstraction function whose domain is the set of addresses defined in the concrete example (Fig. 4). The concretization of the environment is unambiguous as the abstract addresses always represent only one concrete address (adding another call to update in the example would mean that two concrete addresses would be mapped to $@^\sharp(5, \boldsymbol{o})$ at the end). Continuing the example, we get that $\gamma_{\mathbf{ObjN}}[\alpha^{\mathrm{ex}}_{\mathbf{Addr}}](\mathbf{AInst}(@^\sharp(1, \boldsymbol{r}))) = \{\,\mathbf{Instance}@(1, 0)\,\}$. The structural type concretization concerning the attributes of the instance of A yields two different cases: $\gamma_{\mathbf{ObjS}}[\alpha^{\mathrm{ex}}_{\mathbf{Addr}}](\{\,\mathrm{val}\,\}, \mathrm{val} \mapsto @^\sharp(5, \boldsymbol{r}) \wedge \mathrm{atr} \mapsto @^\sharp(10, \boldsymbol{r})) = \{\,f_1, f_2\,\}$, with $f_1 = \mathrm{val} \mapsto @(5, 1)$, and $f_2 = f_1[\mathrm{atr} \mapsto @(10, 0)]$ (depending on the addition of atr to the instance). The concrete heaps mentioned in Fig. 4 are part of the concretization of the abstract heap.

3.3 Abstract Transfer Functions

The abstract evaluation of expressions and statements is very close to the concrete one. We show in Fig. 11 the transfer functions of the assignment, object instantiation, and attribute addition. The signature of the abstract evaluation $\mathbb{E}^{\#}[\![e]\!]$ is $(\mathcal{F}^{\sharp} \times \mathcal{P}(\mathbf{Addr}^{\sharp}) \times \mathcal{E}^{\sharp} \times H^{\sharp}) \rightarrow (\mathcal{F}^{\sharp} \times \mathcal{P}(\mathbf{Addr}^{\sharp}) \times \mathcal{E}^{\sharp} \times \mathcal{H}^{\sharp}) \times (\mathbf{Addr}^{\sharp} \cup \{\bot\})$, so $\mathbb{E}^{\#}[\![e]\!]$ returns the abstract address where the object associated with e is stored, along with the updated abstract state. The semantics of statements has a similar signature except that it only returns the updated abstract state. Similarly to the lift from γ to $\gamma_{\mathcal{D}}$, both semantics can be implicitly lifted to $\mathcal{D}^{\sharp}$, in the case of disjunctive evaluations or of multiple flow tokens (when an expression is evaluated into different types, or nondeterministically raises an exception).[4]

To perform an assignment `x = e`, we evaluate `e` in the abstract, and change the abstract environment ϵ accordingly. The evaluation of `e` may be disjunctive (if the expression may evaluate in multiple abstract addresses, or raises an exception in some cases) and in this case, states are merged by their flow tokens.

`object.__new__` is the function used to instantiate most classes. To analyze this call, we evaluate `e`. If it is a class, we call the recency abstraction to allocate an instance, and return the result of this evaluation, where the abstract heap η is extended with this new address. Otherwise, a type error is raised.

`object.__setattr__` is the function usually called for an attribute update: `x.attr = e`. It is similar in its complexity and shape to the concrete attribute access (Fig. 6). The complexity is due to the notion of *data descriptors*, stored in a parent class of an instance: they can preempt attribute addition and process it as a call to their own `__set__` method. In most cases, however, the *set_field*$^{\sharp}$ function will be called. In this case, we take the evaluation of `x` as an address $@_{x}^{\sharp}$; we then fetch the attribute abstraction for this address. We update the abstraction map and store it as f'_{x}. Then, if the address is recent, we know that it represents only one address in the concrete. Thus, the attribute will be always defined in the object, and we can add it to the underapproximation of the attributes. If the address is old, it may summarize multiple concrete addresses, and the attribute will only be modified in f_{x} by the execution of the assignment. Note that Python also supports attribute update through the `setattr` function. Contrary to the assignment `x.attr = e` where Python's syntax ensures that `attr` is a constant string, `setattr` can take into argument an arbitrary string, which would result in the structural abstraction of the targeted object to be put to top. In that case, we can enable a constant string abstraction to refine the abstract value of the attribute name and help regain precision.

Join Operator and Widening. Going back to the abstract state definition (Fig. 8), we notice that only $\mathbf{ObjS}^{\sharp}$ can be infinite. We thus define a widening operator lifting the structural type abstraction to $\top$ if too many attributes are added. The set of addresses is finite due to the finite number of program locations. Joining two abstract states is done pointwise: by merging states having the same flow tokens, joining the sets for the recency abstraction and the maps for the abstract environment and for the abstract heap.

Analysis of Functions. The analysis of functions is performed in a context-sensitive fashion, by inlining: when a function call is reached, we substitute the call by the body of the function and analyze it. This schemes supports easily dynamic dispatch as well as calling anonymous functions defined using `lambda`.

---

[4] i.e, $\mathbb{S}^{\#}[\![stmt]\!](\delta \in \mathcal{D}^{\sharp}) = \cup_{\varphi \in \text{dom } \delta, (\rho, \epsilon, \eta) \in \delta(\varphi)} \mathbb{S}^{\#}[\![stmt]\!](\varphi, \rho, \epsilon, \eta)$

$$\mathbb{S}^{\#}[\![\, x = e \,]\!]\,(\varphi, \rho, \epsilon, \eta) \overset{\text{def}}{=}$$

$\quad$ letcur $(\varphi, \rho, \epsilon, \eta), @^{\sharp} = \mathbb{E}^{\#}[\![\, e \,]\!]\,(\varphi, \rho, \epsilon, \eta)$ in $\varphi, \rho, \epsilon[x \mapsto \{\, @^{\sharp} \,\}], \eta$

$$\mathbb{E}^{\#}[\![\, \texttt{object.\_\_new\_\_}(e)^{\text{loc}} \,]\!]\,(\varphi, \rho, \epsilon, \eta) \overset{\text{def}}{=}$$

$\quad$ letcur $(\varphi, \rho, \epsilon, \eta), @^{\sharp}_e = \mathbb{E}^{\#}[\![\, e \,]\!]\,(\varphi, \rho, \epsilon, \eta)$ in

$\quad$ if $(\text{fst} \circ \eta)(@^{\sharp}_e) = \mathbf{AClass}\ c$ then

$\qquad$ letcur $(\varphi, \rho, \epsilon, \eta), @^{\sharp} = \mathbb{E}^{\#}[\![\, \text{alloc\_addr(loc)} \,]\!]\,(\varphi, \rho, \epsilon, \eta)$ in $(\varphi, \rho, \epsilon, \eta[@^{\sharp} \mapsto \emptyset]), @^{\sharp}$

$\quad$ else $\mathbb{S}^{\#}[\![\, \texttt{raise TypeError} \,]\!]\,(\varphi, \rho, \epsilon, \eta), \bot$

$$\mathbb{E}^{\#}[\![\, \texttt{object.\_\_setattr\_\_}(x, attr \in string, e) \,]\!]\,(\varphi, \rho, \epsilon, \eta) \overset{\text{def}}{=}$$

$\quad$ letcur $(\varphi, \rho, \epsilon, \eta), @^{\sharp}_x = \mathbb{E}^{\#}[\![\, x \,]\!]\,(\varphi, \rho, \epsilon, \eta)$ in

$\quad$ let $c = mrosearch^{\sharp}(type^{\sharp}(@^{\sharp}_x), attr)$ in

$\quad$ if $c \neq \bot$ then let $f = get\_field^{\sharp}(type^{\sharp}(@^{\sharp}_x), attr)$ in

$\qquad$ if $has\_field^{\sharp}(f, "\_\_\_get\_\_\_") \land has\_field^{\sharp}(f, "\_\_\_set\_\_\_")$ then

$\qquad\quad$ $\mathbb{E}^{\#}[\![\, (get\_field^{\sharp}(f, "\_\_\_set\_\_\_"))(c, @^{\sharp}_x, e) \,]\!]$

$\qquad$ else $\mathbb{E}^{\#}[\![\, set\_field^{\sharp}(@^{\sharp}_x, attr, e) \,]\!]\,(\varphi, \rho, \epsilon, \eta)$

$\quad$ else $\mathbb{E}^{\#}[\![\, set\_field^{\sharp}(@^{\sharp}_x, attr, e) \,]\!]\,(\varphi, \rho, \epsilon, \eta)$

$$\mathbb{E}^{\#}[\![\, set\_field^{\sharp}(@^{\sharp}_x, attr, e) \,]\!]\,(\varphi, \rho, \epsilon, \eta) \overset{\text{def}}{=}$$

$\quad$ letcur $(\varphi, \epsilon, \eta), @^{\sharp}_e = \mathbb{E}^{\#}[\![\, e \,]\!]\,(\varphi, \rho, \epsilon, \eta)$ in let $(t_x, (u_x, f_x)) = \eta(@^{\sharp}_x)$ in

$\quad$ let $f'_x = f_x[attr \mapsto @_e]$ in

$\quad$ if $recent\_addr\ @^{\sharp}_x$ then $(\varphi, \rho, \epsilon, \eta[@^{\sharp}_x \mapsto (t_x, (u_x \cup \{\, attr \,\}, f'_x))])$

$\quad$ else $(\varphi, \rho, \epsilon, \eta[@^{\sharp}_x \mapsto (t_x, (u_x, f'_x))])$

Figure 11 Examples of abstract transfer functions.

▶ **Theorem 1.** *Our analysis is sound: the abstract states computed by our abstract transfer functions over-approximate the concrete states reachable during any program execution. More formally, for any Python statement s:* $\forall \delta \in \mathcal{D}^{\sharp}, \mathbb{S}[\![\, s \,]\!] \circ \gamma_{\mathcal{D}}(\delta) \subseteq \gamma_{\mathcal{D}} \circ \mathbb{S}^{\#}[\![\, s \,]\!](\delta)$

This theorem is proved by mutual structural induction on the structure of Python statements and expressions. The proof is not detailed due to space constraints. The abstract transfer functions of statements and expressions are close to the concrete ones, which makes the proof simple. For example, the semantics of `object.__getattribute__` is the same in the concrete and in the abstract, up to the low-level operators get_field, has_field, *type, isinstance.*

4 Relational Analysis using Parametric Polymorphism

The analysis presented in Sec. 3 is polymorphic, as a variable may be abstracted as a set of addresses of different types. However, bounded parametric polymorphism *à la ML* is impossible to express in this abstraction as we cannot infer that two variables pointing to multiple addresses have the same type. From an abstract interpretation point of view, we lack a relational domain.

Example.   Consider the following program:

```
1  if *: x, y = 1, 2
2  else: x, y = 'a', 'b'
3  z = x + y
```

Our non-relational analysis can infer after line 2 that both x and y have type int or str. However, it cannot show that x and y are either both int or both str, and thus it raises a false TypeError alarm when evaluating x + y.

Type Equality Abstract Domain.   We introduce an abstract domain $\mathcal{Q}^\sharp \stackrel{\text{def}}{=} \mathbf{Id} \rightharpoonup \mathbb{N}$ to track type equalities between variables. It is defined as a partitioning of program identifiers $\mathbf{Id}$ into equivalence classes of equally typed variables. Given $\kappa \in \mathcal{Q}^\sharp$, we ensure that two variables x and y verifying $\kappa(\mathtt{x}) = \kappa(\mathtt{y})$ will have the same nominal type. More precisely, we define an abstract equivalence relation $\equiv^\sharp_{\eta \in \mathcal{H}^\sharp} \subseteq \mathbf{ObjN}^\sharp \times \mathbf{ObjN}^\sharp$ between nominal types:

$$\equiv^\sharp_\eta \stackrel{\text{def}}{=} \{\,(\mathbf{AInt}(@^\sharp_1), \mathbf{AInt}(@^\sharp_2)) \mid (\text{fst} \circ \eta)(@^\sharp_1) \equiv^\sharp_\eta (\text{fst} \circ \eta)(@^\sharp_2)\,\}$$
$$\cup \{\,(\mathbf{AStr}(@^\sharp_1), \mathbf{AStr}(@^\sharp_2)) \mid (\text{fst} \circ \eta)(@^\sharp_1) \equiv^\sharp_\eta (\text{fst} \circ \eta)(@^\sharp_2)\,\}$$
$$\cup \{\,(\mathbf{AFun} -, \mathbf{AFun} -)\,\} \cup \{\,(\mathbf{AClass}\ c, \mathbf{AClass}\ c)\,\}$$
$$\cup \{\,(\mathbf{AMethod}(@^\sharp_1, -), \mathbf{AMethod}(@^\sharp_2, -)) \mid (\text{fst} \circ \eta)(@^\sharp_1) \equiv^\sharp_\eta (\text{fst} \circ \eta)(@^\sharp_2)\,\}$$
$$\cup \{\,(\mathbf{AInst}(@^\sharp_1), \mathbf{AInst}(@^\sharp_2)) \mid (\text{fst} \circ \eta)(@^\sharp_1) \equiv^\sharp_\eta (\text{fst} \circ \eta)(@^\sharp_2)\,\}$$

The concretization function $\gamma_\mathcal{Q} \in \mathcal{Q}^\sharp \rightarrow \mathcal{P}(\mathcal{F} \times \mathcal{E} \times \mathcal{H})$ gives the set of concrete states verifying the equality constraints of an abstract element in $\mathcal{Q}^\sharp$:

$$\gamma_\mathcal{Q}(\kappa) \stackrel{\text{def}}{=} \{\,(f, e, h) \mid \forall \mathtt{x}, \mathtt{y} \in \text{dom}\ \kappa : \kappa(\mathtt{x}) = \kappa(\mathtt{y}) \implies (\text{fst} \circ h \circ e)(\mathtt{x}) \equiv_h (\text{fst} \circ h \circ e)(\mathtt{y})\,\}$$

where $\equiv_h \subseteq \mathbf{ObjN} \times \mathbf{ObjN}$ is the concrete equivalence relation between nominal types in $h \in \mathcal{H}$, derived from $\equiv^\sharp_\eta$ as:

$$n_1 \equiv_h n_2 \iff \exists n^\sharp_1, n^\sharp_2 \in \mathbf{ObjN}^\sharp, \exists \eta \in \mathcal{H}^\sharp, \exists \alpha_\mathbf{Addr} \in \gamma_\text{recency}(\text{dom}\ \eta) : n^\sharp_1 \equiv^\sharp_\eta n^\sharp_2 \wedge$$
$$n_1 \in \gamma_\mathbf{ObjN}[\alpha_\mathbf{Addr}](n^\sharp_1) \wedge n_2 \in \gamma_\mathbf{ObjN}[\alpha_\mathbf{Addr}](n^\sharp_2) \wedge h \in \gamma_\mathcal{H}[\alpha_\mathbf{Addr}](\eta)$$

Reduced Product.   In order to perform a type analysis with bounded parametric polymorphism, we construct a reduced product $\mathcal{D}^\sharp_P$ of the equality domain $\mathcal{Q}^\sharp$ and the non-relational domains $\mathcal{E}^\sharp$ and $\mathcal{H}^\sharp$ of Sec. 3 as follows:

$$\mathcal{D}^\sharp_P \stackrel{\text{def}}{=} \mathcal{F}^\sharp \rightarrow (\mathcal{P}(\mathbf{Addr}^\sharp) \times \mathcal{E}^\sharp \times \mathcal{H}^\sharp \times \mathcal{Q}^\sharp)$$

$$\gamma_P(\delta_p \in \mathcal{D}^\sharp_P) = \bigcup_{\substack{\varphi \in \text{dom}\ \delta_p \\ \delta_p(\varphi) = (\rho, \epsilon, \eta, \kappa)}} \gamma(f, (\rho, \epsilon, \eta)) \cap \gamma_\mathcal{Q}(\kappa)$$

Two reduction operators $\psi_\uparrow, \psi_\downarrow \in (\mathcal{P}(\mathbf{Addr}^\sharp) \times \mathcal{E}^\sharp \times \mathcal{H}^\sharp \times \mathcal{Q}^\sharp) \rightarrow (\mathcal{P}(\mathbf{Addr}^\sharp) \times \mathcal{E}^\sharp \times \mathcal{H}^\sharp \times \mathcal{Q}^\sharp)$ are proposed to refine product states by propagating information between domains (they are extended pointwise so that $\dot{\psi}_\downarrow, \dot{\psi}_\uparrow \in \mathcal{D}^\sharp_P \rightarrow \mathcal{D}^\sharp_P$):

1. The reduction $\psi_\uparrow$ enriches κ with new type equalities. It searches for variables x and y such that both of them point to singleton objects with equivalent nominal types:

$$\begin{array}{cc} \epsilon(\mathtt{x}) = \{@^\sharp_\mathtt{x}\} & \eta(@^\sharp_\mathtt{x}) = \{(n_\mathtt{x}, \_)\} \\ & \wedge \qquad \qquad \qquad \wedge \quad n_\mathtt{x} \equiv^\sharp_\eta n_\mathtt{y} \\ \epsilon(\mathtt{y}) = \{@^\sharp_\mathtt{y}\} & \eta(@^\sharp_\mathtt{y}) = \{(n_\mathtt{y}, \_)\} \end{array}$$

In such case, we add the type equality $\kappa(\mathtt{x}) = \kappa(\mathtt{y})$.

Before reduction	After reduction
$\epsilon = x \mapsto @^{\sharp}(1, \boldsymbol{o}) \wedge y \mapsto @^{\sharp}(1, \boldsymbol{r})$,	ϵ
$\eta = @^{\sharp}(1, \boldsymbol{o}) \mapsto \{\, \mathbf{AInt}, \emptyset, \emptyset \,\} \wedge @^{\sharp}(1, \boldsymbol{r}) \mapsto \{\, \mathbf{AInt}, \emptyset, \emptyset \,\}$,	η
$\kappa = \bot$	$\kappa = x \mapsto 0, y \mapsto 0$

■ **Figure 12** Example of $\psi_{\downarrow}$ reduction.

2. The reduction operator $\psi_{\downarrow}$ refines the non-relational heap η whenever two variables x and y are equally typed in κ and the type of x is more precise. We do so by pruning away the objects referenced by y that are not equivalent to any object pointed by x.

▶ **Theorem 2.** *The reduced product is sound, meaning that the reduction operators do not affect the global product concretization:* $\forall \delta \in \mathcal{D}_P^{\sharp}, \gamma_P(\delta) = \gamma_P(\dot{\psi}_{\uparrow}(\delta)) = \gamma_P(\dot{\psi}_{\downarrow}(\delta))$

Example.   Let us consider again the previous motivating example. After the assignment x, y = 1, 2, both x and y point to singleton integer objects, which allows us to apply $\psi_{\uparrow}$ in order to infer the type equality of x and y (the state is shown in Fig. 12). The same reasoning is applied after the assignment x, y = 'a', 'b' in the else branch. Consequently, the equality is preserved after joining the two abstract states at line 3. When evaluating x in the addition expression, a disjunction with two cases is created, one for each referenced abstract object. In each case, the reduction operator $\psi_{\downarrow}$ is applied to refine the type of y according to the type of x. Therefore, at the end of the program, we infer that no TypeError is raised. Moreover, the reduction $\psi_{\uparrow}$ will find that x, y, and z have the same type.

Bounded Parametric Polymorphism.   In the motivating example, our analysis morally infers that x, y, z have type $\alpha \in \{\, \mathtt{int}, \mathtt{str} \,\}$. We believe this is close to bounded parametric polymorphism. In future work, we want to combine relationality with partial function summaries to deduce that f has type $\alpha \to \alpha, \alpha \in \{\, \mathtt{int}, \mathtt{str} \,\}$ in the program below.

```python
1  def f(x, y): return x + y
2  f(1, 2)
3  f('a', 'b')
```

5 Independent Container Abstractions

Containers are abstracted independently from the rest of the analysis. We show the example of a smashing abstraction [6] for lists. The analysis of dictionaries is implemented similarly: their keys and their values are smashed separately. We have also implemented an expansion-based analysis for tuples.

The smashing abstraction summarizes all the list elements into one "content" variable. Hence, we can infer whether an abstract address is a list, and we can moreover infer the type of list elements using the content variable. As the content variable can have arbitrary abstract values, the abstraction can represent heterogeneous as well as nested lists, which are supported in Python.

Abstract Domain.   We add a new nominal type for lists in $\mathbf{ObjN}^{\sharp}$, denoted as $\mathbf{AList}(@^{\sharp} \in \mathbf{Addr}^{\sharp})$, the address representing the class from which the list is instantiated, to handle classes inheriting from lists ($\mathbf{AList}$ implicitly means $\mathbf{AList}(@^{\sharp}_{\mathbf{Class\ list}})$). We also extend the set of identifiers into $\mathbf{Id}^{+}$, adding a new kind of identifiers, List $@^{\sharp}$, to denote content variables ($@^{\sharp}$ is the address of the list).

$\mathbb{E}^{\#}[\![\,[e_1, \ldots, e_n]^l\,]\!]\,(f, \rho, \epsilon, \eta) =$

   letcur $(\varphi, \rho, \epsilon, \eta), @^{\sharp} = \mathbb{E}^{\#}[\![\,\mathrm{alloc\_addr}(l)\,]\!]\,(\varphi, \rho, \epsilon, \eta)$ in

   letcur $(\varphi, \rho, \epsilon, \eta) = \mathbb{S}^{\#}[\![\,\mathrm{List}\ @^{\sharp} = e_1\,]\!]\,(\varphi, \rho, \epsilon, \eta)$ in

   letcur $(\varphi, \rho, \epsilon, \eta) = \mathbb{S}^{\#}[\![\,\mathrm{List}\ @^{\sharp} \overset{\mathrm{weak}}{=} e_2\,]\!]\,(\varphi, \rho, \epsilon, \eta)$ in

   $\ldots$

   letcur $(\varphi, \rho, \epsilon, \eta) = \mathbb{S}^{\#}[\![\,\mathrm{List}\ @^{\sharp} \overset{\mathrm{weak}}{=} e_n\,]\!]\,(\varphi, \rho, \epsilon, \eta)$ in $(f, \rho, \epsilon, \eta), @^{\sharp}$

$\mathbb{E}^{\#}[\![\,\mathtt{list.append}(l, e)\,]\!]\,(f, \rho, \epsilon, \eta) =$

   letcur $(\varphi, \rho, \epsilon, \eta), @^{\sharp}_l = \mathbb{E}^{\#}[\![\,l\,]\!]\,(\varphi, \rho, \epsilon, \eta)$ in

   letcur $(\varphi, \rho, \epsilon, \eta), @^{\sharp}_e = \mathbb{E}^{\#}[\![\,e\,]\!]\,(\varphi, \rho, \epsilon, \eta)$ in

   if $\mathrm{fst} \circ \eta(@^{\sharp}_l) = \mathbf{AList}(-)$ then

      $\mathbb{E}^{\#}[\![\,\mathtt{None}\,]\!] \circ \mathbb{S}^{\#}[\![\,\mathrm{List}\ @^{\sharp}_l \overset{\mathrm{weak}}{=} e\,]\!]\,(\varphi, \rho, \epsilon, \eta)$

   else $\mathbb{S}^{\#}[\![\,\mathtt{raise\ TypeError}\,]\!]\,(\varphi, \rho, \epsilon, \eta), \bot$

$\gamma_{\mathrm{lists}}(\varphi, \rho, \epsilon, \eta) = \{\,(f, e, h) \mid \alpha_{\mathbf{Addr}} \in \gamma_{\mathrm{recency}}(\rho) \wedge (f', e', h') \in \gamma[\alpha_{\mathbf{Addr}}](\varphi, (\rho, \epsilon, \eta))$

  $\big(\forall v \in \mathrm{dom}\ e', e(v) = e'(v)\big) \wedge \big(\forall @ \in \mathrm{dom}\ h', h(@) = h'(@)\big) \wedge \forall v \in \mathrm{dom}\ \epsilon,$

  $\big(@^{\sharp}(l, m) \in \epsilon(v) \wedge (\mathbf{AList}(@^{\sharp}_c), \emptyset, \emptyset) \in \eta(@^{\sharp}(l, m))\big) \implies \big(\alpha_{\mathbf{Addr}}(e(v)) = @^{\sharp}(l, m)$

  $\wedge\, \exists n \in \mathbb{N}, h(e(v)) = (\mathbf{List}(@_c, (@_1, \ldots, @_n)), \emptyset) \wedge$

  $\alpha_{\mathbf{Addr}}(@_c) = @^{\sharp}_c \wedge \forall 1 \leq i \leq n, \alpha_{\mathbf{Addr}}(@_i) \in \epsilon(\mathrm{List}\ @^{\sharp}(l, m))\big)\,\}$

■ **Figure 13** List transfer functions & extended concretization.

Transfer Functions. Fig. 13 presents the abstract semantics of list allocation. We start by allocating the address of the list through the recency abstraction. Then, we assign, using weak updates, each element of the list to the content variable, and return the address of the list. Adding an element `el` to the list `l` using the function `list.append(l, el)` is also simple. First, we evaluate `l` into an object and check that it is a list. From the evaluation of `l`, we get the address location $@^{\sharp}(l, m)$, letting us access the content variable List $@^{\sharp}(l, m)$. Then, we perform a weak update with the element `el`.

    We emphasize that these transfer functions are independent from most of the analysis: they only need an abstract domain handling address allocation, and another handling assignments. It could for example be reused in the case of a value analysis, or with another allocation-site abstraction.

Concretization. We extend the concretization γ from Fig. 9 into a concretization γ_{lists} taking the list abstraction into account, presented in Fig. 13. To simplify the presentation, $\gamma_{\mathrm{lists}}(\varphi, \rho, \epsilon, \eta)$ employs a variant of γ where the address abstraction $\alpha_{\mathbf{Addr}}$ is fixed, denoted as $\gamma[\alpha_{\mathbf{Addr}}]$. Given an address abstraction $\alpha_{\mathbf{Addr}} \in \gamma_{\mathrm{recency}}(\rho)$, $\gamma[\alpha_{\mathbf{Addr}}]$ provides partially concretized states (f', e', h') that ignore content identifiers as well as the **AList** nominal type. We then extend the (e', h') states into concrete states (e, h): identifiers v that may be lists are added to the environment, the list defined in v is allocated in the concrete heap h at address $e(v)$ with an arbitrary size, and its element addresses are constrained to match with content of the abstract environment of the content variable.

Example. Consider the program $\mathtt{l} = [\mathtt{'a'}^{l_1}, \mathtt{'b'}^{l_2}, \mathtt{'c'}^{l_3}]$. We use labels l_i to denote the program location of each string (program location are actually line numbers and column ranges). In the current flow *cur*, we get the following abstract environment and heap:

$$\epsilon(l) = \{\, @^\sharp(1, \boldsymbol{r}) \,\} \qquad\qquad \epsilon(\mathrm{List}\ @^\sharp(1, \boldsymbol{r})) = \{\, @^\sharp(l_1, \boldsymbol{r}), @^\sharp(l_2, \boldsymbol{r}), @^\sharp(l_3, \boldsymbol{r}) \,\}$$
$$\eta(@^\sharp(1, \boldsymbol{r})) = (\mathbf{AList}, \emptyset, \emptyset) \qquad \eta(@^\sharp(l_i, \boldsymbol{r})) = (\mathbf{AStr}, \emptyset, \emptyset), 1 \le i \le 3$$

$\epsilon(l)$ is bound to the abstract list address, allocated at location 1 and being a recent address. The list variable List $@^\sharp(1, \boldsymbol{r})$ may now point to three strong addresses, each representing one of the strings.

Nested Lists. Our encoding also works for nested lists. In the case of two nested lists, the outermost list variable would point to the address of the innermost list abstract address. These two abstract addresses would also be different because their program locations are different. For example, $\mathtt{l} = [\mathtt{1}^{l_1}, [\mathtt{2.3}^{l_2}]^i]^o$ yields the following abstract state (l_1, l_2 are program locations for the numbers, while i, o are program locations from the inner and the outer list respectively). The variable corresponding to the outer list List $@^\sharp(o, \boldsymbol{r})$ maps to two addresses, including the one of the inner list $@^\sharp(i, \boldsymbol{r})$.

$$\epsilon(l) = \{\, @^\sharp(o, \boldsymbol{r}) \,\} \qquad\qquad \eta(@^\sharp(o, \boldsymbol{r})) = (\mathbf{AList}, \emptyset, \emptyset)$$
$$\epsilon(\mathrm{List}\ @^\sharp(o, \boldsymbol{r})) = \{\, @^\sharp(l_1, \boldsymbol{r}), @^\sharp(i, \boldsymbol{r}) \,\} \quad \eta(@^\sharp(i, \boldsymbol{r})) = (\mathbf{AList}, \emptyset, \emptyset)$$
$$\epsilon(\mathrm{List}\ @^\sharp(i, \boldsymbol{r})) = \{\, @^\sharp(l_2, \boldsymbol{r}) \,\} \qquad \eta(@^\sharp(l_1, \boldsymbol{r})) = (\mathbf{AInt}, \emptyset, \emptyset)$$
$$\eta(@^\sharp(l_2, \boldsymbol{r})) = (\mathbf{AFloat}, \emptyset, \emptyset)$$

This also works for arbitrary nesting. For example, let us consider the following program:

```
1   x = 1
2   for i in range(10): x = [x]
```

With the usual accelerated fixpoint computation, we reach an overapproximation of the concrete state, where x is (an integer or) a nested list containing only integers, but we lose the nest level.

$$\epsilon(x) = \{\, @^\sharp(1, \boldsymbol{r}), @^\sharp(2, \boldsymbol{r}) \,\} \qquad\qquad \eta(@^\sharp(1, \boldsymbol{r})) = (\mathbf{AInt}, \emptyset, \emptyset)$$
$$\epsilon(\mathit{List}\ @^\sharp(2, \boldsymbol{r})) = \{\, @^\sharp(1, \boldsymbol{r}), @^\sharp(2, \boldsymbol{o}) \,\} \quad \eta(@^\sharp(2, \boldsymbol{r})) = (\mathbf{AList}, \emptyset, \emptyset)$$
$$\epsilon(\mathit{List}\ @^\sharp(2, \boldsymbol{o})) = \{\, @^\sharp(1, \boldsymbol{r}), @^\sharp(2, \boldsymbol{o}) \,\} \quad \eta(@^\sharp(2, \boldsymbol{o})) = (\mathbf{AList}, \emptyset, \emptyset)$$

Containers & Polymorphism. The content variables of each container have a name defined by their abstract address, depending on the allocation site. This means that if a variable l is assigned different lists in two different conditional branches (as in the example below), two different element variables will be created, and no polymorphic relationship will be inferred. To counter this issue, we start by unifying both abstract states before performing the join, renaming both element variables into a single one. Our analysis is then able to infer that x has the same type as the element of the list l, which is either integers or strings:

```
1   if *: l = [1,2,3]
2   else: l = ['a', 'b', 'c']
3   x = l[0]
```

6   Implementation and Experimental Evaluation

6.1   Modular Implementation into Mopsa

We have implemented our analysis into Mopsa, a framework aiming at easing the development of static analyses by abstract interpretation [20, 24]. Mopsa currently supports the analysis of subsets of the C and Python programming languages. It is written in OCaml. The framework uses domain modules with a uniform signature to describe abstract domains, control-flow iterators, etc. This ensures that the domains are loosely coupled; they can be easily combined and reused. In addition, domains can rewrite expressions and statements dynamically, which makes it easier to reuse existing abstractions defined over a different syntax or semantics. For instance, Python loops are first rewritten into a canonical shape, while the fixpoint computation is handled by another more generic module, used to handle C loops as well. More details about Mopsa can be found in [20]. The type analysis consists in 2000 lines of OCaml code, the container abstraction consists in 1600 lines of OCaml, and there are 5000 lines of OCaml code defining the iterators and data model of Python.

6.2   Optimizations & Extensions

During our initial testing of our analysis, we noticed that it was slowed down by two factors: the number of exceptions that were raised (creating a large number of abstract states to store), and the analysis of function calls (where the same functions were analyzed many times). This lead us to two optimizations described below. We then explain how we use Python type annotations to analyze more programs.

Exception Abstraction.   When an exception is raised, we store the current abstract state with the exception flow token for the rest of the analysis, in order to reuse it if this exception is caught later on. However, unprecise analyses may raise exceptions frequently. For example, the smashing abstraction handling the list analysis needs (in order to be sound) to raise a potential `IndexError` at each list access, as the analysis does not keep track of the list size. This created many different exceptions stored in the analysis state, but most were never used. To solve the problem, we abstracted sets of such exceptions for which the analysis is deemed *a priori* unprecise (which can be parameterized by the user) into a single abstract exception, joining the corresponding abstract states into one. By default, the exceptions abstracted are `IndexError`, `KeyError` and `ValueError`.

Towards a Partially Modular Function Analysis.   We have implemented a partially modular function analysis, which keeps the abstract input state, the abstract output state and the result of function calls in a cache. When analyzing a function call, the cache is checked: if this function has already been analyzed with the *same* abstract input state, the analysis result is taken directly from the cache. Otherwise, the function is inlined, and the analysis result cached afterwards. In particular, using this cache does not reduce the precision of the analysis, but greatly improves its running times. The experiments displayed in Table 16 show that this cache, combined with the exception abstraction can provide a 32x speedup over the inlining-based analysis (`regex_v8.py`), while the memory usage increased by 15%. In some cases, the inlining-based analysis and the cache-based analysis have the same running times: this may be due to a program having less user-defined functions to analyze, or the cache not being hit because the calling contexts are too different. We believe this cache is particularly efficient because we compute types rather than values: while the abstract state would change a lot during a value analysis (e.g, as loop indexes increase), the abstract state in the case of a type analysis is more stable.

We can also reuse the cache when the current input state is less than the input state kept in the cache. This is actually used in our implementation. In the benchmarks below, choosing this relaxed version improves the running times by 40% in one case (`choose.py`), but introduces imprecision in another case (22 out of the 25 alarms detected in `hexiom.py`).

Note that we keep analyzing functions on demand, at each call-site, knowing their calling context. We believe that performing a sound, context-free function call analysis, as done in most type systems, would not be practical for Python programs, as functions rely on implicit assumptions and may have side effects on their arguments or other variables not defined in the function scope. The cache-based analysis could still be improved to keep only the relevant parts of the whole abstract input and output states, such as the parts that may be read or changed by the function. This extension, which would help reuse more of the analysis results kept in the cache, is left as future work.

Using Type Annotations. As the Python standard library is huge, and partly written in C, we needed a way to support the C-written part without too much manual work. We decided to leverage the work from the Typeshed project [37], which offers type annotations for a substantial part of the standard library. This project uses the standard type annotations recently introduced by the PEP 484 into Python [36]. These type annotations are quite powerful (they feature possibly bounded polymorphism using `TypeVar`, structural subtyping support with `Protocol`, disjunctive function signatures with the `@overload` decorator, ...). For example, they can completely specify the signature of the `fspath` function described in the introduction:

```python
T = TypeVar('T', str, bytes)          @overload
class PathL(Protocol[T]):             def fspath(path: str) -> str: ...
  def fspath(self) -> T: ...          @overload
@overload                             def fspath(path: bytes) -> bytes: ...
def fspath(path: PathL[T]) -> T: ...
```

These annotations remain less expressive than our analysis, as side-effects (such as raised exceptions, aliasing) cannot be expressed yet, but a type-and-effect system [23] could be used. When a stubbed module is imported, our analyzer parses the corresponding annotated file and stores its functions (similarly for classes and variables). Then, when a stubbed function is called, we check that the arguments match the function signature. In that case, we assume that the function has no side effects and returns an object of the annotated return type, which we convert into an abstract object. Note that the use of these annotations changes the soundness of our analyzer: exceptions raised by concrete functions where we used their annotated counterpart will not be reported.

6.3 Experimental Evaluation

In this part, we evaluate our implementation on several benchmarks. We compare our analysis with four tools aiming at detecting incorrect programs potentially reaching runtime errors: the abstract-interpretation-based value analysis of Python [13], and three other tools having close goals: a tool by Fritz & Hage [12], Typpete [16] and Pytype [42]. We also include the static analysis part of RPython [2] in our comparison, whose goal is to compile a restricted subset of Python into more efficient programs. [12, 16, 42] try to infer a static type that ensures the absence of dynamic typing errors, while we go further and check whether dynamic typing errors can occur and result in exceptions that stop the program (hence, we

can successfully analyze correct programs that are not typable but are nevertheless correct as they recover from dynamic type errors). Both [13] and our analyzer generate as output the set of exceptions that may escape to the toplevel, with detailed exception messages close to those given by Python. While this naturally includes type-related exceptions, we also take into account that even type errors can be caught and handled by the program, in which case they are not reported as errors. Contrary to [12, 16, 42], we also detect other errors (such as out of bound list accesses), in order to have a sound analysis (though we are unprecise in most cases). We also show the performance gain of the optimizations described in Section 6.2.

Competing Tools.   The tool developed by Fritz and Hage performs a data-flow analysis, computing the type of each variable. While the original paper [12] experiments various tradeoffs between performance and precision (using different widenings, flow-sensitivity, context-sensitivity, . . .), we used the default arguments of the provided artifact. As mentioned in their paper, this tool does not handle exceptions nor generators. Its output is a dump of the data-flow map, associating to each program point the type of each variable. A program is untypable for this tool when the analyzer puts reachable variables to the bottom type.

Typpete encodes type inference of Python programs into a MaxSMT problem, and passes it to Z3 to solve it. If Z3 yields unsat, the program is untypable. Otherwise, the output of Typpete is a type annotation of the input program. It comes with around 40 examples on which we were able to test our analyzer. Typpete restricts its input to Python programs where variables have a single type in a program (but it handles subtyping: a variable having both types `int` and `str` will have type `object`) and dynamic attribute addition is not supported. When there is a type error, Z3 finds the inference problem to be unsatisfiable and Typpete shows a line in relationship with the type error. As the structure of the program is lost during the MaxSMT encoding, the line shown by Typpete is not always the line where the error will occur at runtime. Typpete supports the basics of the PEP 484 type annotations, and uses them for its stubs, or to guide the analysis on an input program.

Pytype is a tool developed by Google and actively used to maintain their codebase, hence it is more mature than the other tools. It performs an analysis that is not described formally, but it has a wide language and library support (it also uses Typeshed), allowing it to scale to large codebases. It outputs the last type of each variable when the typing is successful, and can produce a type annotation of the input program. It also produces clear error messages looking like the exceptions raised by Python when it detects an erroneous program.

We obtained the analyzer developed by Fromherz et al. [13]. It performs a value analysis by abstract interpretation. Its output is a set of potentially uncaught exceptions.

RPython performs a data-flow analysis to check that a program is part of the subset it can efficiently compile. It outputs the control-flow graph with the inferred types.

We compare the analysis of these five tools to two different configurations of our analyzer:

- **Conf. 1** using inlining and no exception smashing of the alarms;
- **Conf. 2** using partially modular analysis and exception smashing in alarms (see Sec. 6.2).

We have not noticed any improvement using the relational domain in the benchmarks, so the results below use the non-relational analysis. The relational analysis increases the analyses times by a factor 5 at most.

Benchmarks.   We chose 5 of the biggest benchmarks from Typpete's unit tests (prefixed with 🐇 in Table 16). We also took 12 benchmarks from Python's reference interpreter [39] (prefixed with 🐍 in the table). Out of the 44 benchmarks currently available, we chose 12

with no external dependencies and few standard library module dependencies, so that most tools are able to analyze them. We argue that while the benchmarks are not very long, these Python programs are realistic and may call a lot of functions. For example, calling Python profiler `cProfile` on `chaos.py` shows more than 469,000 function calls. We also add three small tests focusing on characteristics we believe are paramount to performing a sound analysis of Python programs: taking into account object mutation and aliasing, as shown in Fig. 3 (file `mutation.py`); being able to precisely analyze introspection operators such as `isinstance` and `hasattr`, in order to analyze precisely a program calling the function `fspath` from Fig. 1 for example (file `isinstance.py`, Fig. 14); and analyzing precisely exception-related control-flow operators in order to have a precise analysis and avoid raising type errors later caught by an `except TypeError` statement for example (file `exception.py`, Fig. 15). Finally, we analyze the two main parts (`processInput.py, choose.py`) of a real-world command-line utility from Facebook, called PathPicker (prefixed with ; the LOC for these files consists in the size of the file and all the PathPicker files imported by this one). These two parts are multifile projects depending on other modules from PathPicker (which are inlined and analyzed by our tool), as well as some standard library modules, including `re, subprocess, json, curses, posixpath, argparse, configparser, os, stat, locale, bz2, lzma` respectively handling regular expressions, external process calls, json files, curses command-line interfaces, file-related functions, argument parsing, configuration file parsing, operating-system and file status, internationalization of output and compression algorithms. As all these modules are at least partially written in C, we used the annotations from Typeshed [37] to support them. All program constructs used in the benchmarks are supported by our tool, meaning our analysis is sound on them.

Performance and Precision Evaluation. We test the language support, the performance and the precision of each tool. An analyzer may crash due to an unsupported construction (🐛), or may timeout after one hour of analysis (⊘). We measured the analysis time five times for each benchmark and tool, and the mean is displayed. All tools are deterministic. In the evaluation of our tool in its most efficient configuration (Conf. 2), the column ⚠ displays the number of false alarms raised (the precision is identical in Conf. 1), with the smashed exceptions (corresponding to the unprecise exceptions raised by the list and dictionary abstractions) separated. The results are displayed in Table 16.

We notice that our analysis is able to scale to benchmarks a few thousand lines long, within a reasonable analysis time. Some benchmarks take longer to analyze: for example `hexiom.py` has a lot of nested loops, and functions are called multiple times, so the analyzer has a lot of fixpoint computations and inlining to perform (it performs 1770 analyses of 5-levels nested loops). It seems that the other type analyzers [12, 16, 42] do not perform fixpoint computations over loops (which at least for the case of Typpete seems sound as it infers more abstract types). Similarly, Typpete is able to perform an efficient analysis, although it lacks library support to analyze some Python benchmarks, and is unable to analyze programs where a variable is initiliazed in a (potentially unexecuted) loop. The

```
1  if isinstance(x, int): y = 4
2  else: y = 'a'
3  z = 2 + y
```

Figure 14 isinstance.py.

```
1  try: z = 2 + 'a'
2  except: z = 3.14
3  a = z+1
```

Figure 15 exception.py.

Table 16 Analysis of Python benchmarks.

Name	LOC	Conf. 1	Conf. 2	⚠	[12]	Pytype	Typpete	[13]	RPython
isinstance.py	3	42ms	40ms	0	1.2s	0.78s	0.67s	10ms	4.9s
exception.py	3	37ms	34ms	0	1.3s	0.70s	0.57s	9ms	🐞
mutation.py	12	34ms	34ms	0	1.3s	0.75s	0.68s	11ms	🐞
🐢 disjoint__sets.py	45	70ms	59ms	0†	0.92s	0.91s	1.2s	🐞	8.8s
🐢 functions.py	58	41ms	39ms	0†⚷	1.2s	0.84s	1.1s	🐞	8.0s
🐍 fannkuch.py	59	76ms	69ms	0†	1.2s	0.80s	🐞	0.31s	🐞
🐢 bellman__ford.py	61	0.17s	0.24s	0†	1.4s	0.99s	1.4s	2.4m	7.1s
🐍 float.py	63	0.13s	82ms	0†	1.7s	0.92s	1.3s	0.84s	5.6s
🐢 coop__concat.py	64	45ms	43ms	0†	1.8s	0.81s	1.3s	20ms	🐞
🐍 spectral__norm.py	74	0.32s	0.19s	1	1.6s	0.98s	🐞	🐞	🐞
🐢 crafting.py	132	0.48s	0.41s	0†⚷	1.6s	0.97	1.7s	🐞	🐞
🐍 nbody.py	157	1.4s	0.80s	1†⚷⚹	1.7s	1.3s	🐞	🐞	🐞
🐍 chaos.py	324	8.9s	2.3s	0†⚹	13s	11s	🐞	🐞	🐞
🐍 raytrace.py	411	3.5s	1.5s	7⚹	36s	2.8s	🐞	🐞	🐞
🐍 scimark.py	416	0.85s	0.55s	2†	8.5s	4.4s	🐞	🐞	🐞
🐍 richards.py	426	11s	5.0s	2†⚹	38s	2.4s	🐞	🐞	7.8s
🐍 unpack__seq.py	458	13s	4.2s	0⚹	1.1s	7.4s	2.7s	14s	🐞
🐍 go.py	461	4.0m	15s	32†⚹	8.5s	3.4s	🐞	🐞	🐞
🐍 hexiom.py	674	6.9m	22s	25†⚷⚹	🐞	4.2s	🐞	🐞	🐞
🐍 regex__v8.py	1792	8.2m	15s	0†	4.9s	🕐	1.7m	🐞	🐞
f processInput.py	1417	6.1s	4.8s	7†⚷⚹	2.4s	11s	🐞	🐞	🐞
f choose.py	2562	8.6m	46s	17⚷†⚹	1.7s	15s	🐞	🐞	🐞

🐞 unsupported by the analyzer (crash) 🕐 timeout (after 1h)
Smashed Exceptions: KeyError ⚷, IndexError †, ValueError ⚹

tool from Fritz and Hage is quite fast (the running times are measured by running a docker container due to the software dating from 2011), but we will see later that it is unsound in most cases. It fails on `hexiom.py` due to a parsing error. Pytype is a more mature analyzer, and it does not fail on any of the benchmarks, but times out in the `regex_v8.py` benchmark (after reaching out to Google, it appears to be a performance bug from Pytype in its analysis of big dictionaries). The value analysis [13] is unable to support the standard library functions needed for most benchmarks (supporting new library functions in the value analysis is more time-consuming, as it requires to include the effect of this function on the abstract values). On the benchmarks it is able to pass, our analysis is in average 8.5× faster than the value analysis; it also scales to benchmarks 5× longer. RPython is able to type 6 out of 22 benchmarks. In the 16 other cases, 5 seem to be due to internal bugs, while the 11 last cases are due to constructs unsupported by RPython. Compared to RPython, our analysis is able to fully analyze invalid programs (it will not stop at the first type exception, which can be caught later on).

Our analysis raises a few alarms (as all programs are correct, all alarms are false alarms here). As the programs did not mix types implicitly, our analysis was sufficiently precise to avoid raising false alarms over type and attribute errors. However, the smashing abstraction of the lists and the dictionaries creates some false alarms: dictionary values having different types (and heterogeneously-typed lists) are smashed into content variables, triggering imprecision over the types in the rest of the analysis. In addition, the smashing abstraction currently does

not keep track of the (potential) emptiness of lists: this creates a few alarms, as variables initially defined during a loop iteration over a list may be undefined in the rest of the program if the list is empty (raising `Unboundlocalerror` in `spectral_norm.py`, `nbody.py`, `bm_raytrace.py`, `bm_scimark.py` and 22 in `hexiom.py`). More generally, the absence of information on the length of the list means that each list access should raise a potential `IndexError` (we could reduce the number of false alarms by adding a small domain keeping track of the abstract length of lists; this is left as future work). Similarly, `KeyError` are raised upon each dictionary access, and `ValueError` may be raised during list unpacking. The spurious `IndexError` are not raised by the value analysis [13], which is able to track the length of lists. The alarms are not raised by the three other analyzers, as they focus on type errors only, and not on finding which exceptions may be raised. As each analyzer has its own output, we were unable to compare their precision in all cases, and only study the precision on the first three small examples. In the `isinstance.py` example, both our tool, [13] and Pytype are precise, but the others are unprecise ([12] yields an unsound result, Typpete declares the program incorrect). For `exception.py`, [12] does not support exceptions; while [13], Pytype and Typpete declare the program incorrect; our tool does not raise any alarm. Concerning `mutation.py`, both Pytype, [13] and our tool are precise. Typpete is unprecise (it declares some integers and strings to be of object type), and [12] infers a variable holding a string as an integer.

Soundness Evaluation. We experimentally check the soundness of the analyzers. We believe soundness is important in order to detect all potential errors. As each benchmark file was a correct Python program, we created erroneous variants having one type error (by introducing a string into an integer variable), in order to check the soundness of each analyzer (similarly to the evaluation of Typpete). We then ran each analyzer on those files (the correct and the erroneous one each time), and checked whether the inferred types and alarms were matching the behavior of the program: either the analysis seemed sound as the types and alarms were correctly raised, or the analysis was unsound (no error was detected in the erroneous variant). The injection of type errors to evaluate the soundness is simplistic, as our goal was to quickly test the soundness of the other tools. Our analyzer is sound by construction but may include implementation bugs, and it would be interesting to automate error injection to experimentally check the soundness more thoroughly.

We find that our analysis catches the errors in all erroneous variants and is thus sound – as expected – in these cases. Typpete is sound over the programs it can analyze. The tool from Fromherz et al. is unsound in the case of `unpack_seq.py` due to an implementation error. The artifact from [12] is unable to detect errors in all cases except `fannkuch.py`. Pytype is not sound in a few cases (`bellman_ford.py`, `crafting_challenge.py`, `float.py`, `richards.py`, `spectral_norm.py`, `unpack_seq.py`).

Evaluation Summary. Our analysis is sound, it reports a few false alarms. Preliminary results indicate that our analyzer is able to scale, at least on programs a few thousand lines long. The soundness evaluation showed that even simple errors such as replacing an integer with a string may go unnoticed for unsound analyzers.

Comparatively, Pytype is the most advanced tool: it is able to scale and seems to support most of the standard library. It is however unsound in some cases. Both Typpete and [13] perform a sound analysis, but they lack some language or library support in the bigger benchmarks. The tool from Fritz and Hage is able to analyze programs very quickly, and supports most benchmarks, but it is unsound in most cases. RPython has a different goal, as it focuses on compiling a more static subset of Python efficiently. Most of the benchmarks use constructs too dynamic for RPython to compile them efficiently.

7   Related Work

In this section, we discuss related work, focusing on formal analyses for the two most popular dynamic programming languages: JavaScript and Python.

JavaScript.   JavaScript is defined by a standard, and has been formalized in Coq [7] and in K [26]. [18] presents the first static analysis by abstract interpretation for JavaScript, and provides an implementation called TAJS. [19] builds upon TAJS to define a more efficient interprocedural analysis. As strings play a wide role in the semantics of JavaScript, precise string abstractions are studied in [1, 22]. [21] uses a static type analysis to optimize numerical computations datatypes. [17] proposes a method to soundly translate some `eval` statements into code, in order to improve the precision of their analysis. An analysis of asynchronous JavaScript built upon TAJS is presented in [32].

Python.   In [28], the authors define a mechanized semantics for a restricted subset of Python, consisting in basic values (integers, booleans) and control structures (loops, conditionals), but not taking objects into account. [31] proposes a semantics of Python 2.5 under the form of a Haskell interpreter. [27] defines a small-step semantics for a core Python language, λ_π, as well as a compiler from Python to λ_π, and a λ_π interpreter written in Racket. [15] shows a rewriting semantics for Python using the K framework [29]. [13] defines an interpreter-like semantics on which the concrete semantics presented in Section 2 is based.

Pyannotate [40] and MonkeyType [38] are tools performing a dynamic analysis: they collect the types of a Python program during its execution. Contrary to static analyses where an abstraction of the set of program traces is computed, dynamic analyses only run on one trace, meaning that non-determinism due to inputs or random choices will not be taken into account. While this approach helps developers move to type-annotated Python codes, the collected types correspond to one execution only, and are thus not sound.

A middle-end between dynamic and static type analysis is gradual typing [30, 14]. In that case, the programmer annotates parts of the program, which can then be typechecked. The unannotated parts of the program have an unknown type called top, from which any static type can be cast to and from. The soundness theorem of gradual typing then guarantees that if a program gradually typechecks, the only type errors that may occur at runtime are casts concerning variables having type top. Gradual typecheckers for Python include Mypy [35] and Pyre [41]. Both tools restrict the input language, as annotated variables can have only one type during the program execution (this type can be a union of types). By contrast, our type analysis is more permissive as it does not restrict the dynamic typing features of Python. We also do not require any annotation to run our analysis.

The closest approaches to our work [12, 42, 16, 2] have been described in the experimental evaluation (Sec. 6.3). It should be noted that Fritz & Hage [12] test many different parameter instantiations of their data-flow analysis. We believe that in the context of formal verification, a precise, context-sensitive, sound type analysis is useful. The flow-sensitivity is needed to precisely analyze exception catching statements, but neither have we tested this hypothesis on a larger scale nor have we tried selective flow-sensitivity, contrary to [12]. [34] presents a predictive analysis based on symbolic execution for Python. It consistently finds bugs and scales to projects of thousands of lines of codes, but it does not cover all executions, and is thus not sound. [13] performs a static value analysis by abstract interpretation. It uses abstract values similar to the ones presented in [18]. This analysis is not strictly more expressive than ours: while it focuses on values, it is relational over numerical datatypes,

but not over types. Our type analysis is more scalable in its implementation, as supporting new constructs consists in providing a type signature (and knowing the side effects of this function, including the potentially raised exceptions). To scale more quickly, we can also reuse Typeshed and its type annotations to support most of the standard library (though we will lose any side effect of the annotated function in that case). The type analysis also uses less memory and is quicker: we store type information rather than abstract values, and the fixpoint computations during the analysis of loops converge more quickly (types vary less than values, for example during loop iterations). The experiments of this value analysis consisted in some of Python's unit tests and some of Python's benchmarks. As the unit tests consist mostly in equality assertions over values, our type analyzer is unable to verify these. However, the running times for the type analysis on those tests are similar to the ones described in [13]. The benchmarks were shown in Table 16.

8 Conclusion

We have developed a static type analysis of Python programs by abstract interpretation, which collects uncaught exceptions that may be raised during a program execution. This analysis is sound, and its modular implementation scales to benchmarks a few thousand lines long. In addition, we found that compared to other type analyses, we uniquely take into account dynamic Python features such as object mutability, introspection operators, and exception-based control-flow statements.

Future work includes: speeding-up the inlining-based analysis with an efficient, summary-based function analysis; exploring the abstraction-precision trade-offs of the analysis (e.g, using an expansion-based container abstraction); analyzing bigger programs; combining this analysis with a value analysis (e.g, to keep track of the abstract length of summarized lists, in order to remove `IndexError`-based false alarms); finally, we will consider using the type information inferred by the analysis to optimize the execution of Python programs.

─── **References** ───

1   Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald Søndergaard, Peter J. Stuckey, and Chenyi Zhang. Combining string abstract domains for JavaScript analysis: An evaluation. In *TACAS (1)*, volume 10205 of *LNCS*, pages 41–57, 2017.

2   Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. Rpython: a step towards reconciling dynamically and statically typed OO languages. In *DLS*, pages 53–64. ACM, 2007.

3   Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In *SAS*, volume 4134 of *LNCS*, pages 221–239. Springer, 2006.

4   Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. In *OOPSLA*, pages 69–82. ACM, 1996.

5   Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Static analysis and verification of aerospace software by abstract interpretation. *Foundations and Trends in Programming Languages*, 2(2-3):71–190, 2015.

6   Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation*, volume 2566 of *LNCS*, pages 85–108. Springer, 2002.

7   Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised JavaScript specification. In *POPL*, pages 87–100. ACM, 2014.

8   Patrick Cousot. Types as abstract interpretations. In *POPL*, pages 316–331. ACM Press, 1997.

9   Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

10 Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - A software analysis perspective. In *SEFM*, volume 7504 of *LNCS*, pages 233–247. Springer, 2012.

11 David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS*, volume 5825 of *LNCS*, pages 53–69. Springer, 2009.

12 Levin Fritz and Jurriaan Hage. Cost versus precision for approximate typing for Python. In *PEPM*, pages 89–98. ACM, 2017.

13 Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. Static value analysis of Python programs by abstract interpretation. In *NFM*, volume 10811 of *LNCS*, pages 185–202. Springer, 2018.

14 Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *POPL*, pages 429–442. ACM, 2016.

15 Dwight Guth. A formal semantics of Python 3.3. Technical report, University of Illinois, 2013. URL: `https://www.ideals.illinois.edu/bitstream/handle/2142/45275/Dwight_Guth.pdf?sequence=1&isAllowed=y`.

16 Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. MaxSMT-based type inference for Python 3. In *CAV (2)*, volume 10982 of *LNCS*, pages 12–19. Springer, 2018.

17 Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *ISSTA*, pages 34–44. ACM, 2012.

18 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *SAS*, volume 5673 of *LNCS*, pages 238–255. Springer, 2009.

19 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *SAS*, volume 6337 of *LNCS*, pages 320–339. Springer, 2010.

20 M. Journault, A. Miné, R. Monat, and A. Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyzer. In *Proc. of the 11th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE19)*, pages 1–17, July 2019.

21 Francesco Logozzo and Herman Venter. RATA: Rapid atomic type analysis by abstract interpretation - application to JavaScript optimization. In *CC*, volume 6011 of *LNCS*, pages 66–83. Springer, 2010.

22 Magnus Madsen and Esben Andreasen. String analysis for dynamic field access. In *CC*, volume 8409 of *LNCS*, pages 197–217. Springer, 2014.

23 Daniel Marino and Todd D. Millstein. A generic type-and-effect system. In *TLDI*, pages 39–50. ACM, 2009.

24 A. Miné, A. Ouadjaout, and M. Journault. Design of a modular platform for static analysis. In *Proc. of 9h Workshop on Tools for Automatic Program Analysis (TAPAS'18)*, LNCS, page 4, 28 August 2018.

25 Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for C-like languages. In *PLDI*, pages 229–238. ACM, 2012.

26 Daejun Park, Andrei Stefanescu, and Grigore Rosu. KJS: A complete formal semantics of JavaScript. In *PLDI*, pages 346–356. ACM, 2015.

27   Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: The full monty. In *OOPSLA*, pages 217–232. ACM, 2013.

28   Ranson, Hamilton, and Fong. A semantics of Python in Isabelle/HOL. Technical report, University of Regina, 2008. URL: `http://www.cs.uregina.ca/Research/Techreports/2008-04.pdf`.

29   Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebr. Program.*, 79(6):397–434, 2010.

30   Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *ECOOP*, volume 4609 of *LNCS*, pages 2–27. Springer, 2007.

31   Gideon Joachim Smeding. An executable operational semantics for Python. *Universiteit Utrecht*, 2009. URL: `http://gideon.smdng.nl/wp-content/uploads/thesis.pdf`.

32   Thodoris Sotiropoulos and Benjamin Livshits. Static analysis for asynchronous JavaScript programs. In *ECOOP*, volume 134 of *LIPIcs*, pages 8:1–8:30. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

33   Fausto Spoto. The julia static analyzer for Java. In *SAS*, volume 9837 of *LNCS*, pages 39–57. Springer, 2016.

34   Zhaogui Xu, Peng Liu, Xiangyu Zhang, and Baowen Xu. Python predictive analysis for bug detection. In *SIGSOFT FSE*, pages 121–132. ACM, 2016.

35   Mypy. `http://mypy-lang.org/`, 2018. Accessed: 2018-07-22.

36   Python enhancement proposal 484, about type hints. `https://www.python.org/dev/peps/pep-0484/`, 2018. Accessed: 2018-07-23.

37   Typeshed. `https://github.com/python/typeshed/`, 2018. Accessed: 2018-07-22.

38   Monkeytype. `https://github.com/Instagram/MonkeyType`, 2019. Accessed: 2019-10-22.

39   Performance benchmarks from Python's reference interpreter. `https://github.com/python/pyperformance/`, 2019. Accessed: 2019-10-22.

40   Pyannotate. `https://github.com/dropbox/pyannotate`, 2019. Accessed: 2019-10-22.

41   Pyre-check. `https://github.com/facebook/pyre-check`, 2019. Accessed: 2019-10-22.

42   Pytype. `https://github.com/google/pytype`, 2019. Accessed: 2019-10-22.

43   Pathpicker. `https://github.com/facebook/pathpicker/`, 2020. Accessed: 2020-01-03.