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 in-
def fspath(p):
    sb = (str, bytes)
    if isinstance(p, sb):
        return p
    if hasattr(p, "fspath"):
        r = p.fspath()
        if isinstance(r, sb):
            return r
        else:
            raise TypeError
    else:
        raise TypeError

class Path:
    def fspath(self):
        return 42

if *:
p = "/dev"
else:
p = Path()
r = fspath(p)

Motivating example.
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 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, 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.

\[n\]
\[2\] In practice, we are nevertheless limited to program that do not use eval.
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. Our analysis infers the (nominal and structural) type of variables
  and tracks 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].
- We have implemented our analysis 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.

Novelty. 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. Compared to previous work on
dynamic languages, the formalization of the concretization functions parameterized by the
recency abstraction is novel, as well as the modular implementation. The scalability has
been improved to analyze real (albeit small) programs.
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.

2.1 Concrete Semantic Domain

The memory state consists in two parts: the environment $E$ and the heap $H$, defined in Fig. 2. The environment $E$ is a partial, finite map from variable identifiers $\text{Id}$ to addresses $\text{Addr}$. The set $\text{Addr}$ of heap addresses is infinite. To simplify the definition of our semantics and its abstraction, we assume that, up to isomorphism, $\text{Addr}$ has the form $\text{Location} \times \mathbb{N}$ where $\text{Location}$ is the set of program locations (in the following, a line number). We use the notation $@((l, n)) \in \text{Addr}$ to designate unambiguously the $n$-th address allocated at program
we have two possible heaps, depending on the non-deterministic choice
where
continuations: we label states using
provide a semantics for operations that do not return immediately, such as raise, we use
flow tokens: we label states using flow tokens (elements of \( F \)), so the states we consider are

Figure 2: Concrete semantic domains

| \( \text{Id} \subseteq \text{string} \) | \( \text{Addr} \equiv \text{Location} \times \mathbb{N} \) |
| \( \mathcal{E} \equiv \text{Id} \rightarrow \text{Addr} \cup \text{LocalErr} \) | \( \text{ObjN} \equiv \text{int}(a \in \text{Addr}, i \in \mathbb{Z}) \cup \text{bool}(b) \) |
| \( \mathcal{H} \equiv \text{Addr} \rightarrow \text{ObjN} \times \text{ObjS} \) | \( \cup \text{string}(a, s \in \text{string}) \cup \text{None} \cup \text{NotImpl} \) |
| \( \mathcal{F} \equiv \{ \text{cur, ret, brk}, \) | \( \cup \text{List}(a, ls), ls \in \text{Addr}^* \cup \text{Method}(a, f) \) |
| \text{cont, exn} \ a, a \in \text{Addr} \} | \( \cup \text{Fun}(f) \cup \text{Class}(c) \cup \text{Instance}(a \in \text{Addr}) \) |
| \( \cup \text{Obj}(z) \subseteq \text{Addr} \) | \( \cup \text{ObjS} \equiv \text{string} \rightarrow \text{Addr} \) |

Although everything is object in Python, we distinguish builtin objects from the other
types. 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 \( \text{int}(\@, 10) \), where \( @ \) points to the class of \( A \). This behavior was not expressible in the
previous concrete semantics [13]. In the following, the notation \( \text{int}(10) \) denotes \( \text{int}(\@, 10) \),
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 \( @\langle 1, 0 \rangle \). The instance of \( A \) created at line 6 has an attribute \( \text{val} \) being
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 \( \text{val} \) now maps to the string ‘\text{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 \( \text{atr} \) 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 \( F \)), so the states we consider are
Two possible heaps: either the heap from line 9, or:

We denote by $\mathbb{E}[e]$ the semantics of expression $e$. This semantics has the following signature: $\mathbb{E}[e] : F \times E \times H \rightarrow F \times E \times H \times (\text{Addr} \cup \{\bot\})$, 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: $F \times E \times H \rightarrow F \times E \times H$. 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).
Then, we search for object.__getattribute__ in our implementation, but not very common, so we describe only the semantics of object.__getattribute__ being either __getattribute__ or __getattr__. The first method call usually ends up with 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 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).

\[
\text{letcur} (f, e, h, @e) = \text{E} \{ e \} (f, e, h) \text{ in } \\
\text{letcur} (f, e, h, @s) = \text{E} \{ s \} (f, e, h) \\
\text{if} \ \text{isinstance}(h(@e), \text{str}) \ \text{then} \\
\text{let} \ \text{string} \ s = h(@e) \ \text{in} \\
\text{let} \ c = \text{mrosearch}(\text{type}(h(@e)), s) \ \text{in} \\
\text{if} \ c \neq \bot \ \text{then} \\
\text{let} \ cs = h(\text{get_field}(c, s)) \ \text{in} \\
\text{if} \ \text{has_field}(cs, \text{"__get\_\_"}) \ \text{and} \ \text{has_field}(cs, \text{"__set\_\_"}) \ \text{then} \\
\text{E} \{ \text{get_field}(cs, \text{"__get\_\_"})(cs, h(@e), \text{type}(h(@e))) \} (f, e, h) \\
\text{else} \text{if} \ \text{has_field}(h(@e), s) \ \text{then} \text{E} \{ \text{get_field}(h(@e), s) \} (f, e, h) \\
\text{else} \ E \{ \text{get_field}(c, s) \} (f, e, h) \\
\text{else if} \ \text{has_field}(h(@e), s) \ \text{then} \text{E} \{ \text{get_field}(h(@e), s) \} (f, e, h) \\
\text{else} \ S[\text{raise} \ \text{AttributeError}] (f, e, h), \bot \\
\text{else} \ S[\text{raise} \ \text{TypeError}] (f, e, h), \bot
\]

Figure 6 Concrete semantics of attribute access

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 \bot 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, @) = e1 in e2” which unfolds into “let (f, e, h, @) = e1 in if f \neq \text{cur} then (f, e, h, @) else e2”.

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 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).

\[\text{E}[\text{object.__getattribute__}(e, s)] (f, e, h) \triangleq \]

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]).
$S[\text{raise } e](f, e, h) =$
let $\text{cur}(f, e, h, @) = E_J e() (f, e, h) \text{ in}$
if $\text{isinstance}(h(@), \text{BaseException})$ then $(\text{exn } @, e, h)$
else $S[\text{raise } \text{TypeError}](f, e, h)$

$S[\text{try : tbody except exn as v : texc}](f, e, h) =$
let $(f, e, h) = S[\text{tbody}](f, e, h) \text{ in}$
if $f \neq \text{exn } _$ then $(f, e, h)$
else let $\text{exn } @_{\text{raised}} = f \text{ in}$
  let $\text{cur}(f, e, h, @_{\text{exn}}) = E_J \text{exn} (cur, e, h) \text{ in}$
  if $\neg \text{issubclass}(h(\text{exn}), \text{BaseException})$ then
    $S[\text{raise } \text{TypeError}](\text{cur}, e, h)$
  else if $\text{isinstance}(h(\text{exn})_{\text{raised}, \text{exn}})$ then
    let $(f, e, h) = S[\text{texc}](\text{cur}, e[v \mapsto @_{\text{raised}}], h) \text{ in } (f, e \{ v \}, h)$
  else $(f, e, h)$

Figure 7 Concrete semantics of exceptions

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 $\text{BaseException}$ class. If that is the case, we change the flow token to
$\text{exn}$, 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: $\text{tbody}$ is evaluated; if no exception is raised the evaluation is finished. Otherwise, if
an exception (stored at address $@_{\text{raised}}$) is raised during this evaluation, we evaluate $\text{exn}$: if
it is an exception (i.e., if it inherits from $\text{BaseException}$), and if the raised exception is an
instance of $\text{exn}$, we evaluate $\text{texc}$, where the variable $v$ is now bound to the raised exception
(until the end of the evaluation of $\text{texc}$). 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.

3.1 Abstract Domain

The structure of the abstract domain $D^\mathcal{E}$ 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 the 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
$l \in \text{Location}$ are partitioned into two abstract addresses via a recency criterion: the most
recent allocation is represented with the abstract address $\sharp\alpha(l, r)$, while all the previous
ones are abstracted by a unique abstract address $\sharp\alpha(l, o)$. The concretization function
$\gamma_{\text{recency}}$ takes as input the set of abstract addresses currently defined ($\rho \in \mathcal{P}(\text{Addr}^\sharp)$), 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 $\alpha_{\text{Addr}}$; we denote this parameterization with the following notation: $\gamma[\alpha_{\text{Addr}}]$.

The allocation of a new abstract address is handled by the auxiliary function $\text{alloc_addr}$:

$$
\mathcal{E}^\sharp[\text{alloc_addr}(l)](\varphi, \rho \in \mathcal{P}(\text{Addr}^\sharp), \epsilon, \eta) = \\
\begin{cases}
\text{if } \sharp\alpha(l, r) \in \rho \text{ then } \sharp\alpha(l, r), \mathcal{S}^\sharp[\sharp\alpha(l, o) \overset{\text{weak}}{=} \sharp\alpha(l, r)](\varphi, \rho, \epsilon, \eta) \\
\text{else } \sharp\alpha(l, r), (\varphi, \rho \cup \{\sharp\alpha(l, r)\}, \epsilon, \eta)
\end{cases}
$$

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

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 $\gamma_{\mathcal{E}}[\alpha_{\text{Addr}}]$.
To support Python scoping, variables can also point to $\text{LocalErr}$ to represent variables that
can be locally undefined.

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

The nominal part keeps only the class information of the object and forgets about its
value. The concretization $\gamma_{\text{ObjN}}[\alpha_{\text{Addr}}]$ maps abstract addresses to concrete ones. In
particular, abstract instances of built-in values (integers, strings, booleans, None, NotImp)
\[
\begin{align*}
\text{Addr}^2 & \overset{\text{def}}{=} \text{Location} \times \{r, o\} \\
\mathcal{F}^2 & \overset{\text{def}}{=} \{\text{cur, ret, brk, cont, exn } \bar{x}\}, \\
\mathcal{E}^2 & \overset{\text{def}}{=} \text{Id} \rightarrow \mathcal{P}(\text{Addr}^2 \cup \{\text{LocalErr}\}) \\
\mathcal{H}^2 & \overset{\text{def}}{=} \text{Addr}^2 \rightarrow \mathcal{P}(\text{ObjN}^2 \times \text{ObjS}^2) \\
\mathcal{D}^2 & \overset{\text{def}}{=} \mathcal{F}^2 \rightarrow (\mathcal{P}(\text{Addr}^2) \times \mathcal{E}^2 \times \mathcal{H}^2) \\
\text{ObjN}^2 & \overset{\text{def}}{=} \text{AInt}(a) \cup \text{AStr}(a) \cup \text{AMethod}(a, f) \cup \text{AClass}(c) \cup \text{AFun}(f) \\
\text{ObjS}^2 & \overset{\text{def}}{=} \{\top\} \cup \{\text{ret}(\text{string}) \times (\text{string} \rightarrow \text{Addr}^2)\}
\end{align*}
\]

\textbf{Figure 8} \hspace{1em} Definition of abstract states

\[
\begin{align*}
\gamma_{\text{recency}}(\rho \in \mathcal{P}(\text{Addr}^2)) & = \{\alpha_{\text{Addr}} : \text{Addr} \rightarrow \text{Addr}^2 \mid \\
\langle \bar{x}(l, o) \rangle & \in \rho \implies \exists m \in \mathbb{N}, \forall i \leq m, \alpha_{\text{Addr}}(\bar{x}(l, i)) = \bar{x}(l, o) \land \\
\langle \bar{x}(l, r) \rangle & \in \rho \implies \exists n \in \mathbb{N}, \{\alpha_{\text{Addr}}^{-1}(\bar{x}(l, r)) = \{\bar{x}(l, n)\} \land \\
& n = 1 + \max\{i \mid \alpha_{\text{Addr}}(\bar{x}(l, i)) = \bar{x}(l, o)\}\}\}\}
\end{align*}
\]

\[
\begin{align*}
\gamma_{\text{ObjN}}[\alpha_{\text{Addr}}](\text{int}(\bar{x})) & = \{\text{int}(\bar{x}) \mid i \in \mathbb{Z}, \alpha_{\text{Addr}}(\bar{x}) = \bar{x}^2\} \\
\gamma_{\text{ObjN}}[\alpha_{\text{Addr}}](\text{string}(\bar{x})) & = \{\text{string}(\bar{x}, s) \mid s \in \mathcal{P}(\text{str}), \alpha_{\text{Addr}}(\bar{x}) = \bar{x}^2\} \\
\gamma_{\text{ObjN}}[\alpha_{\text{Addr}}](\text{Method}(\bar{x}, f)) & = \{\text{Method}(\bar{x}, f) \mid \alpha_{\text{Addr}}(\bar{x}) = \bar{x}^2\} \\
\gamma_{\text{ObjN}}[\alpha_{\text{Addr}}](\text{Class}(c)) & = \{\text{Class}(c)\} \\
\gamma_{\text{ObjN}}[\alpha_{\text{Addr}}](\text{Fun}(f)) & = \{\text{Fun}(f)\} \\
\gamma_{\text{ObjN}}[\alpha_{\text{Addr}}](\text{Instance}(\bar{x})) & = \{\text{Instance}(\bar{x}) \mid \alpha_{\text{Addr}}(\bar{x}) = \bar{x}^2\} \\
\gamma_{\text{ObjS}}[\alpha_{\text{Addr}}](\text{string}) & = \{f \in \text{ObjS} = \text{string} \rightarrow \text{Addr} \mid \\
& f \subseteq \text{dom} f \subseteq \text{dom} f^2 \land \forall s \in \text{dom} f, \alpha_{\text{Addr}}(f(s)) = f^2(s)\} \\
\gamma_{\text{ObjS}}[\alpha_{\text{Addr}}](\top) & = \text{ObjS} \\
\gamma_{\mathcal{H}}[\alpha_{\text{Addr}}](\eta \in \mathcal{H}^2) & = \{h \in \mathcal{H} \mid \forall \bar{x} \in \text{dom} \alpha_{\text{Addr}}, h(\bar{x}) = (n, s) \land \\
& \exists (n^2, s^2) \in \eta(\alpha_{\text{Addr}}(\bar{x})), n \in \gamma_{\text{ObjN}}[\alpha_{\text{Addr}}](n^2) \land s \in \gamma_{\text{ObjS}}[\alpha_{\text{Addr}}](s^2)\}\}\}
\end{align*}
\]

\textbf{Figure 9} \hspace{1em} Concretization of the abstract states

\begin{align*}
\gamma_{\text{recency}}(\rho \in \mathcal{P}(\text{Addr}^2), \epsilon \in \mathcal{E}^2, \eta \in \mathcal{H}^2) & = \{f, e, h \in \mathcal{F} \times \mathcal{E} \times \mathcal{H} \mid \\
& \alpha_{\text{Addr}} \in \gamma_{\text{recency}}(\rho) \land f \in \gamma_{\mathcal{F}}[\alpha_{\text{Addr}}](\varphi) \land e \in \gamma_{\mathcal{E}}[\alpha_{\text{Addr}}](\epsilon) \land h \in \gamma_{\mathcal{H}}[\alpha_{\text{Addr}}](\eta)\} \\
\gamma_{\mathcal{P}}(\delta \in \mathcal{D}^2) & = \bigcup_{\varphi \in \text{dom} \delta} \gamma_{\mathcal{P}}(\delta(\varphi))
\end{align*}

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 built-in classes). For instance: \(\gamma_{\text{ObjN}}[\alpha_{\text{Addr}}](\text{AInt}(\bar{x}^2_{\text{Class.int}})) = \{\text{int}(\bar{x}^2_{\text{Class.int}}, i) \mid i \in \mathbb{Z}\}, \)
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 @2(6, r), representing the instance of A allocated at line 6 and having a unique attribute val. This attribute

<table>
<thead>
<tr>
<th>Line</th>
<th>$\epsilon \in \mathcal{E}$</th>
<th>$\eta \in \mathcal{H}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>$A \mapsto {@^2(1, r)}$</td>
<td>$@^2(1, r) \mapsto {\text{AClass } A, {_\text{init}, _\text{update}}, _\text{init} \mapsto {@^2(2, r) \wedge _\text{update} \mapsto @^2(4, r)}}$</td>
</tr>
<tr>
<td>6</td>
<td>$x \mapsto {@^2(6, r)}$</td>
<td>$@^2(6, r) \mapsto {\text{AFun}(_\text{init}) \mapsto \emptyset, \emptyset; @^2(4, r) \mapsto {\text{AFun}(\text{update}), \emptyset, \emptyset}$</td>
</tr>
<tr>
<td>7</td>
<td>$y \mapsto {@^2(5, r)}$</td>
<td>$@^2(5, r) \mapsto {\text{AInt}, \emptyset, \emptyset}$</td>
</tr>
<tr>
<td>8</td>
<td>$z \mapsto {@^2(6, r)}$</td>
<td>$@^2(5, o) \mapsto {\text{AInt}, \emptyset, \emptyset}$</td>
</tr>
<tr>
<td>9</td>
<td>$y \mapsto {@^2(5, o)}$</td>
<td>$@^2(6, r) \mapsto {\text{AInt}, @^2(1, r), {\text{val}}, \text{val} \mapsto @^2(5, r) \wedge \text{atr} \mapsto @^2(10, r)}$</td>
</tr>
<tr>
<td>10</td>
<td>$y \mapsto {@^2(10, r)}$</td>
<td>$@^2(10, r) \mapsto {\text{AInt}, \emptyset, \emptyset}$</td>
</tr>
</tbody>
</table>

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

where the addresses subscripted by Class int represent the address of the built-in integer class. Similarly to the concrete, AInt implicitly means AInt(@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 $\gamma_{\text{ObjS}}[\alpha_{\text{Addr}}]$. 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 $\gamma_{\mathcal{H}}[\alpha_{\text{Addr}}]$ of an abstract heap $\eta$ is the set of heaps $h$ such that each address $@$ is bound to an object $(n, s)$ where: $n$ is a concretization of $n^2$, and $s$ a concretization of $s^2$, and the abstract object $(\#(n^2, s^2))$ is in the abstract heap at the abstract address $\alpha_{\text{Addr}}(\@)$. Note that the domain of the heap is defined by the recency abstraction, i.e. $\text{dom } \eta \equiv \rho$.

**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 $\gamma_{\mathcal{F}}[\alpha_{\text{Addr}}]$. A whole abstract state maps flow tokens to abstract environments and heaps. To concretize a whole abstract state $\delta \in \mathcal{D}$ using $\gamma_{\mathcal{P}}$, we concretize each image of $\delta$ separately and join the resulting concrete states. To concretize an element $\delta(\varphi) = (\rho, \epsilon, \eta)$ using $\gamma$, we first fix an address abstraction $\alpha_{\text{Addr}}$ using the concretization of the recency abstraction $\rho$. Then, each part (flow token $\varphi$, environment $\epsilon$, heap $\eta$) is concretized separately.

### 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 @2(6, r), representing the instance of A allocated at line 6 and having a unique attribute val. This attribute
points to the address @♯(5, r), representing the integer result of the multiplication at line 5. After assigning the addresses @♯(5, r) and @♯(6, 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 @♯(5, r) already exists, it is renamed to @♯(5, o) to denote that it is no longer the most recent allocation. Consequently, the variable y now points to @♯(5, 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 @♯(5, 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 @♯(6, 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, γ = { @♯(1, r), @♯(5, o), @♯(5, r), @♯(6, r), @♯(10, r) }. Let us define α♯Addr = @♯(1, r) → @♯(1, r); @♯(6, r) → @♯(6, r); @♯(5, o) → @♯(5, o); @♯(5, 1) → @♯(5, r); @♯(10, 0) → @♯(10, r). We can check that α♯Addr is one of the abstraction functions defined in γrecency(ρ). 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 @♯(5, o) at the end). Continuing the example, we get that γObj[α♯Addr](AInst[@♯(1, r)]) = { Instance@♯(1, 0) }. The structural type concretization concerning the attributes of the instance of A yields two different cases: γObj[α♯Addr](\{ val, val → @♯(5, r) & atr → @♯(10, r) = \{ f1, f2 \}, with f1 = val → @♯(5, 1), and f2 = f1[atr → @♯(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 E♯[e] is \( (F^2 × P(Addr^2)) × E^♯ × H^♯) → (F^♯ × P(Addr^♯)) × E^♯ × H^♯ × (Addr^♯ \cup \{ \⊥ \}) \), so 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 γD, both semantics can be implicitly lifted to D^♯, in the case of disjunctive evaluations or of multiple flow tokens (when an expression is evaluated into different types, or nondeterministically raises an exception).

To perform an assignment x = e, we evaluate e in the abstract, and change the abstract environment e 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 =

---

4 i.e., S♯[stmt](δ ∈ D^♯) = ∪_{φ ∈ dom} δ(ρ, e, η) ∈ δ(φ) S♯[stmt](φ, ρ, e, η)
We update the abstraction map and store it as finite due to the finite number of program locations. Joining two abstract states is done through the Join Operator and Widening.

### Join Operator and Widening

Going back to the abstract state definition (Fig. 8), we notice that only $\text{Obj}S^2$ 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`.

▶ **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 $s$:

$$
\forall \delta \in D^2, S[s] \circ \gamma_D(\delta) \subseteq \gamma_D \circ S^\#[s](\delta)
$$

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:

```python
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 $\mathbb{Q}^2 \mathrel{\overset{\text{def}}{=} } \mathbb{Id} \twoheadrightarrow \mathbb{N}$ to track type equalities between variables. It is defined as a partitioning of program identifiers $\mathbb{Id}$ into equivalence classes of equally typed variables. Given $\kappa \in \mathbb{Q}^2$, we ensure that two variables $x$ and $y$ verifying $\kappa(x) = \kappa(y)$ will have the same nominal type. More precisely, we define an abstract equivalence relation $\equiv^\#_\eta \mathrel{\overset{\text{def}}{=} } \text{ObjN}^2 \times \text{ObjN}^2$ between nominal types:

$$
\equiv^\#_\eta \mathrel{\overset{\text{def}}{=} } \{ (\text{AInt}(\alpha_1^1), \text{AInt}(\alpha_2^1)) \mid (\text{fst}\eta)(\alpha_1^1) \equiv^\#_\eta (\text{fst}\eta)(\alpha_2^1) \}
\cup \{ (\text{AInt}(\alpha_1^2), \text{AInt}(\alpha_2^2)) \mid (\text{fst}\eta)(\alpha_1^2) \equiv^\#_\eta (\text{fst}\eta)(\alpha_2^2) \}
\cup \{ (\text{AFun}\lambda, \text{AFun}\lambda) \} \cup \{ (\text{AClass}\ c, \text{AClass}\ c) \}
\cup \{ (\text{AMethod}(\alpha_1^1, \lambda), \text{AMethod}(\alpha_2^1, \lambda)) \mid (\text{fst}\eta)(\alpha_1^1) \equiv^\#_\eta (\text{fst}\eta)(\alpha_2^1) \}
\cup \{ (\text{AMethod}(\alpha_1^2, \lambda), \text{AMethod}(\alpha_2^2, \lambda)) \mid (\text{fst}\eta)(\alpha_1^2) \equiv^\#_\eta (\text{fst}\eta)(\alpha_2^2) \}
\cup \{ (\text{AInst}(\alpha_1^1), \text{AInst}(\alpha_2^1)) \mid (\text{fst}\eta)(\alpha_1^1) \equiv^\#_\eta (\text{fst}\eta)(\alpha_2^1) \}
\cup \{ (\text{AInst}(\alpha_1^2), \text{AInst}(\alpha_2^2)) \mid (\text{fst}\eta)(\alpha_1^2) \equiv^\#_\eta (\text{fst}\eta)(\alpha_2^2) \}
$$
The concretization function $\gamma_Q : Q^d \rightarrow P(F \times E \times H)$ gives the set of concrete states verifying the equality constraints of an abstract element in $Q^d$:

$$\gamma_Q(\kappa) \overset{\text{def}}{=} \{(f, e, h) | \forall x, y \in \text{dom } \kappa : \kappa(x) = \kappa(y) \implies (\text{fst } \circ h \circ e)(x) \equiv_h (\text{fst } \circ h \circ e)(y)\}$$

where $\equiv_h \subseteq \text{ObjN } \times \text{ObjN}$ is the concrete equivalence relation between nominal types in $h \in H$, derived from $\equiv_D^h$ as:

$$n_1 \equiv_h n_2 \iff \exists n_1^h, n_2^h \in \text{ObjN}^h, \exists \alpha_{\text{Addr}} \in \gamma_{\text{recency}}(\text{dom } \eta) : n_1^h \equiv_{\eta} n_2^h \wedge n_1 \in \gamma_{\text{ObjN}}[\alpha_{\text{Addr}}](n_1^h) \wedge n_2 \in \gamma_{\text{ObjN}}[\alpha_{\text{Addr}}](n_2^h) \wedge h \in \gamma_H[\alpha_{\text{Addr}}](\eta)$$

**Reduced Product.** In order to perform a type analysis with bounded parametric polymorphism, we construct a reduced product $D^2_P$ of the equality domain $Q^d$ and the non-relational domains $E^d$ and $H^d$ of Sec. 3 as follows:

$$D^2_P \overset{\text{def}}{=} F^d \rightarrow (P(\text{Addr}^d) \times E^d \times H^d \times Q^d)$$

$$\gamma_P(\delta_p \in D^2_P) = \bigcup_{\varphi \in \text{dom } \delta_p} \gamma(f, (\rho, \epsilon, \eta)) \cap \gamma_Q(\kappa)$$

Two reduction operators $\psi_\uparrow, \psi_\downarrow \in (P(\text{Addr}^d) \times E^d \times H^d \times Q^d) \rightarrow (P(\text{Addr}^d) \times E^d \times H^d \times Q^d)$ are proposed to refine product states by propagating information between domains (they are extended pointwise so that $\psi_\uparrow, \psi_\downarrow \in D^2_P \rightarrow D^2_P$):

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

$$\epsilon(x) = \{ @x^1 \} \quad \epsilon(y) = \{ @y^1 \} \quad \eta(\text{addr } x) = \{ n_x, \_ \} \quad \eta(\text{addr } y) = \{ n_y, \_ \} \quad n_x \equiv_{\eta} n_y$$

In such case, we can add the equality $\kappa(x) = \kappa(y)$.

2. The reduction operator $\psi_\downarrow$ refines the non-relational heap $\eta$ whenever two variables $x$ and $y$ are equally typed in $\kappa$ 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: $\forall \delta \in D^2_P, \gamma_P(\delta) = \gamma_P(\psi_\uparrow(\delta)) = \gamma_P(\psi_\downarrow(\delta))$.

**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 $x, y = \text{}\text{'a'}, \text{}\text{'b'}$ 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 Type Error is raised. Moreover, the reduction $\psi_\downarrow$ will find that $x, y, z$ have the same type.

![Figure 12](image-url) Example of $\psi_\downarrow$ reduction
Bounded Parametric Polymorphism. In the motivating example, our analysis morally infers that $x, y, z$ have type $\alpha \in \{\text{int}, \text{str}\}$. We believe this is close to bounded parametric polymorphism. In future work, we want to combine relationality with function summaries to deduce that $f$ has type $\alpha \rightarrow \alpha, \alpha \in \{\text{int}, \text{str}\}$ in the program below.

```python
def f(x, y): return x + y
f(1, 2)
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 $\text{ObjN}^2$, denoted as $\text{AList}(\alpha^\sharp \in \text{Addr})$, the address representing the class from which the list is instantiated, to handle classes inheriting from lists ($\text{AList}$ implicitly means $\text{AList}(\alpha^\sharp_{\text{Class list}})$). We also extend the set of identifiers into $\text{Id}^+$, adding a new kind of identifiers, $\text{List}^\sharp_{\alpha^\sharp}$, to denote content variables ($\alpha^\sharp$ is the address of the list).

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 $\text{el}$ to the list $\text{l}$ using the function $\text{l}.\text{append}(\text{l}, \text{el})$ is also simple. First, we evaluate $\text{l}$ into an object and check that it is a list. From the evaluation of $\text{l}$, we get the address location $\alpha^\sharp_{\text{Addr}}(\text{l}, m)$, letting us access the content variable $\text{List}^\sharp_{\alpha^\sharp}(\text{l}, m)$. Then, we perform a weak update with the element $\text{el}$.

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 $\gamma$ concretization from Fig. 9 into a concretization $\gamma_{\text{lists}}$ taking the list abstraction into account, presented in Fig. 13. To simplify the presentation, $\gamma_{\text{lists}}(\varphi, \rho, \epsilon, \eta)$ employs a variant of $\gamma$ where the address abstraction $\alpha_{\text{Addr}}$ is fixed, denoted as $\gamma[\alpha_{\text{Addr}}]$. Given an address abstraction $\alpha_{\text{Addr}} \in \gamma_{\text{recency}}(\rho)$, $\gamma[\alpha_{\text{Addr}}]$ provides partially concretized states $(f', e', h')$ that ignore content identifiers as well as the $\text{AList}$ 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.
The list variable \(List\) respectively). The variable corresponding to the outer list \(List\) of the strings.

Consider the program 

\[
\begin{align*}
\text{l} &= \left[\text{\\textquoteleft.a\textquoteright}, \text{\\textquoteleft.b\textquoteright}, \text{\\textquoteleft.c\textquoteright}\right] \\
\text{o} &= \left[\text{\\textquoteleft.1\textquoteright}, \text{\\textquoteleft.2\textquoteright}, \text{\\textquoteleft.3\textquoteright}\right]
\end{align*}
\]

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

\[
\begin{align*}
\text{l} &= \left[\text{\\textquoteleft.a\textquoteright}, \text{\\textquoteleft.b\textquoteright}, \text{\\textquoteleft.c\textquoteright}\right] \\
\text{o} &= \left[\text{\\textquoteleft.1\textquoteright}, \text{\\textquoteleft.2\textquoteright}, \text{\\textquoteleft.3\textquoteright}\right]
\end{align*}
\]
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.

$$
\epsilon(x) = \{ @^{x}(1, r), @^{x}(2, r) \} \\
\eta(@^{x}(1, r)) = (AInt, \emptyset, \emptyset) \\
\epsilon(List @^{x}(2, r)) = \{ @^{x}(1, r), @^{x}(2, o) \} \\
\eta(@^{x}(2, r)) = (AList, \emptyset, \emptyset) \\
\epsilon(List @^{x}(2, o)) = \{ @^{x}(1, r), @^{x}(2, o) \} \\
\eta(@^{x}(2, o)) = (AList, \emptyset, \emptyset)
$$

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:

```python
1  if *:
2    l = [1, 2, 3]
3  else:
4    l = [a', b', c']
5  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 imprecise (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 Fig. 16 show that this cache, combined with the exception abstraction can provide a 22x 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: 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.
Contrary to the other tools, Pytype is a tool developed by Google and actively used to maintain their codebase. 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 instanceof.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 mutifile 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, lma 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.

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

![Figure 14 instanceof.py](Image)

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

![Figure 15 exception.py](Image)
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 to be 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 initialized in a (potentially unexecuted) loop. The 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 \(5 \times\) faster than the value analysis; it also scales to benchmarks \(5 \times\) 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 \texttt{UnboundLocalError} in \texttt{spectral\_norm.py}, \texttt{nboby.py}, \texttt{bm\_raytrace.py}, \texttt{bm\_scimark.py} and 22 in \texttt{hexiom.py}). More generally, the absence of information on the length of the list means that each list access should raise a potential \texttt{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, \texttt{KeyError} are raised upon each dictionary access, and \texttt{ValueError} may be raised during list unpacking. The spurious \texttt{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 \texttt{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 \texttt{exception.py}, [12] does not support exceptions; while [13], Pytype and Typpete declare the program incorrect; our tool does not raise any alarm. Concerning \texttt{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 \texttt{unpack_seq.py} due to an implementation error. The artifact from [12] is unable to detect errors in all cases except \texttt{fannkuch.py}. Py-type is not sound in a few cases (\texttt{bellman_ford.py}, \texttt{crafting_challenge.py}, \texttt{float.py}, \texttt{richards.py}, \texttt{spectral_norm.py}, \texttt{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, \( \lambda_\pi \), as well as a compiler from Python to \( \lambda_\pi \), and a \( \lambda_\pi \) 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


