Static Analysis by Abstract Interpretation Collecting Types of Python Programs

Raphaël Monat

Static Analysis by Abstract Interpretation
Collecting Types of Python Programs

Raphaël Monat under the supervision of Antoine Miné, APR team, LIP6

March 15 – August 31, 2018

General context Software bugs are costly: they can have disastrous consequences on critical systems, but also on more common applications such as Web servers. It is thus interesting to possess tools helping developers detect those bugs before they ship software into production.

The most common approach to avoid those errors is to use testing, which is inefficient: tests are usually written by hand, taking time, and they cover a few executions of a program, but not all of them.

The approach I undertook during this internship is to design a static analyzer by abstract interpretation for Python. Static analyzers are programs that automatically analyze an input program and report errors this input program may contain (given a certain class of errors the static analyzer may detect). In particular, if a static analyzer is sound and reports no error on a given input, we know that this program does not contain any error from the class the static analyzer is able to discover. Abstract interpretation [11] is a theory formalizing links between a precise but costly semantics with some computable, approximated counterpart, thus guiding the design and implementation of sound static analyzers.

State of the art static analyzers include Julia [21] (analyzing Java), Astrée [8] and Sparrow [19] (for C software). Those are only analyzing statically typed languages. There are no mature static analyzers available for dynamic languages such as JavaScript and Python, although a few static analyses have already been developed and implemented.

Research problem During this internship, I developed a static analysis of Python programs collecting the types of each variable, and possible type error exceptions. Python is a popular programming language, especially used in teaching and in the scientific community, well-known for its powerful syntax. Major software projects written in Python include the Django web framework and the SageMath computer algebra system. Python is an object-oriented dynamic language, where every value is an object. Python's dynamism means that undeclared variables, type incompatibilities, ... are exceptions detected at runtime.

```
def fspath(p):
 if isinstance(p, (str, bytes)):
  return p
 elif hasattr(p, "__fspath__"):
  res = p.__fspath__()
  if isinstance(res, (str, bytes)):
   return res
  else:
   raise TypeError("...")
 else:
  raise TypeError("...")
```

Figure 1: Simplified version of `os.fspath`

Two notions of typing may be used when writing Python programs, and both have operators that may be used at runtime to inspect objects and affect the control-flow, contributing to Python's dynamic nature. Nominal typing is based on the inheritance relation: a class is a subtype of another if its inherits from it. Its associated operator is `isinstance`, taking an object and a class, and returning true only when the object's class is a subtype of the provided class. Duck typing is more flexible and focuses on attributes. In this setting, an object is a subtype of another if it behaves similarly to the other one, i.e, if the attributes used are the same. `hasattr` tests if the first parameter has the second parameter as attribute. A typical example of this duality of types is presented in Figure 1. This example is a simplified version of the *fspath* [2] function defined in the `os` module of the standard library. This function takes as parameter an object p. If p is already a string (i.e, if it is an instance of `str` or `bytes`), it is simply returned. Otherwise, p should have an attribute called `__fspath__` (otherwise the `TypeError` exception at the end is raised), that is called (if it is not a method, another `TypeError` is raised), and we check that the result is a string as well.

We stress on the fact that we aim at designing a static analysis, and not a type system. The analysis presented here uses an abstract domain of types, and is inspired by works on polymorphic typing and gradual typing, but it is formalized using abstract interpretation, and does not use proofs nor implementation techniques from the world

of type systems. This approach differs from the design of a type system in a few different ways we outline now. Contrary to a type system, our analysis will not reject any program, as it reports potential type errors – though it may report false alarms. This is especially useful as type errors may be caught later on during the execution of a Python program, and are potentially not fatal. In the setting of type systems, a more precise type system should reject less programs. Here, a more precise analysis will provide more precise type information, and less false alarms. A soundness proof will be shown in the setting of abstract interpretation, but it is different from a soundness proof of a type system. By reusing abstract domains, our analysis is naturally flow-sensitive, and could be extended to be value-sensitive. The same kind of sensitivity would be non-trivial to achieve using a type system. However, static type analyses of Python are not analyzing functions as modularly as a type system would.

In practice, static analyzers work well when the types of variables are known. There is less work on static analysis of dynamic languages. For Python, a value analysis by abstract interpretation is presented in [13]. Pytype [4], Typpete [16] and a tool from Fritz and Hage [12] are static analyzers inferring types for Python, but they have not been designed with abstract interpretation. The theoretical part of this work is inspired from Cousot who studied typing of ML-like languages using abstract interpretation [10].

We wanted to bring a few novelties by designing a new static type analysis. First, using abstract interpretation, we can provide a proof of soundness linking the concrete semantics of Python with the type analysis. Other works on Python were not stating soundness theorems. While soundness proofs using progress and preservation or logical relations exist for type systems, our proof is a simple inclusion. Second, we used more expressive types, allowing for a more precise analysis. The expressiveness of these types also allows a partially modular analysis of functions, that should bring a performance improvement compared to other tools inlining functions [4, 16, 12]. Third, due to the implementation, this analysis supports complex control flow of Python programs. In particular, type errors in Python are raised exceptions that can be caught afterwards, so we wanted to be precise when analyzing exception-handling statements. Fourth, we created an analysis that does not restrict Python's language: as an example, thanks to flow-sensitivity of the analysis, variables of both branches of an if statement are not forced to have the same type.

Contribution During this internship, I formalized a new abstract domain permitting a static type analysis of Python programs. This type analysis collects sets of types for each variable (corresponding to the types of the values each variable may have) and type errors. I started by searching into bug reports of Python projects such as Django to find motivating examples. The abstract domain I defined is relational (it can infer relationships between variables), and supports bounded parametric polymorphism. It handles both nominal and duck typing, with a preference for nominal typing, as our types are mainly instances of classes. A partially modular analysis of functions is possible thanks to the polymorphism. In particular, the analysis can express that the function in Figure 1 has for signature $\alpha \to \beta, \alpha \in \{\, str, bytes, Instance[object, \_\_fspath\_\_] \,\}, \beta \in \{\, str, bytes \,\}$. I partially implemented this analysis into a modular static analyzer called MOPSA.

Arguments supporting its validity I defined a concretization function linking our abstract domain with the concrete semantics, and I established a paper proof that the operators of the abstract domain are sound. Our abstract domain is able to infer more precise type relationships than the other tools. Once implemented, the modularity of the analysis of functions will provide a significant performance improvement over other static analyzers. The implementation is a work in progress (it does not support the modular analysis yet), and it shows that our analyzer is at least as precise as the state of the art. Python is a big language with a massive standard library, so many features are not supported yet.

Summary and future work I have formalized and implemented a sound static type analysis of Python programs, which is relational and uses bounded parametric polymorphism. It would be interesting to expand the type analysis so it supports most of the standard library, in order to analyze real-life projects. I would like to see if the type information given by my analysis can help guide the value analysis presented in [13]. Future work includes developing efficient ways permitting to analyze programs relying on Python's standard library, designing modular analyses of functions and more complex abstract domains.

1 Introduction

I undertook this internship under the supervision of Antoine Miné, in the APR (Algorithms, Programs and Resolution) team, in the LIP6 laboratory. This team works on algorithms and programming languages, on both a theoretical and practical side. Studying concurrent programming languages, programming on new architectures and developing new static analyses are the main research areas of this team. I worked with my supervisor and Abdelraouf Ouadjaout, a post-doc working on the implementation of MOPSA, and in particular on the value analysis of Python programs.

The goal of this internship was to develop a static analysis by abstract interpretation that would find type errors in Python programs. We start by giving some background material on abstract interpretation in Section 2. Then, we propose a concrete semantics for Python, which is mainly based on the semantics of Fromherz et al. [13]. Section 4 defines the abstract domain I developed, allowing for a static type analysis of Python programs. In Section 5 we focus on implementation choices, and on the integration of the analysis into the static analysis framework called MOPSA. We comment on some results provided by the implemented analysis in Section 6. We study the related work in Section 7, and compare our analysis with other static type analyses.

2 Background on Abstract Interpretation

This part is a short introduction to abstract interpretation. The goal of abstract interpretation is to provide a theory for the approximation of mathematical objects. Most of the time, abstract interpretation applies to the approximation of formal semantics. Using abstract interpretation, we can move from an uncomputable program semantics, called the concrete semantics, to an approximate, computable semantics, called abstract semantics. The latter can then be implemented in a static analyzer. The transition from an uncomputable semantics to a less precise one is provided with guarantees about the approximation of the former semantics by the latter. We will start with a simple example relating $\mathcal{P}(\mathbb{Z})$ with intervals, to introduce the main concepts of abstract interpretation. Then, we move to a more elaborate example about static type analysis.

2.1 An Abstraction of $\mathcal{P}(\mathbb{Z})$ using Intervals

Let us suppose we would like to analyze a program manipulating integers. In that case, we need to represent the set of values each variable may take, so we would like to represent $\mathcal{P}(\mathbb{Z})$. If we have variables x and y ranging over values $X, Y \in \mathcal{P}(\mathbb{Z})$, we can compute the set of values $x + y$ as $\{ x + y \mid x \in X, y \in Y \}$. The problem is that each operation has a cost linear in the size of the input sets, and that the size of the sets is unbounded. Each operation is too costly to perform an efficient program analysis.

An alternative is to use intervals $\mathcal{I} = \{ [a, b] \mid a \in \mathbb{Z} \cup \{ -\infty \}, b \in \mathbb{Z} \cup \{ +\infty \}, a \leq b \} \cup \{ \bot \}$, where $\bot$ represents the empty interval. Of course, intervals will be less precise than elements of $\mathcal{P}(\mathbb{Z})$, but each operation has a constant cost, which is much better. Now, if we abstract our sets X and Y by intervals, we would like to have operations such as the addition on intervals as well. Moreover, we would like those interval operations to be sound: if there is z such that $z = x + y, x \in X, y \in Y$, we would like z to be in the interval resulting from the interval addition. In this case, intervals act as an overapproximation of a set of integers, and we can find functions converting sets of integers into intervals and conversely.

In 1977, Radhia and Patrick Cousot introduced Abstract Interpretation [11]. At the heart of this theory is the notion of Galois connection, establishing formally a relationship between an abstract, approximate world and a more concrete world.

Definition 1 (Galois connection)**.** Let $(C, \leq)$ and $(A, \sqsubseteq)$ be two partially ordered sets (posets), and $\alpha : C \to A, \gamma : A \to C$. (α, γ) is a Galois connection if:

$$\forall a \in A, \forall c \in C, c \leq \gamma(a) \iff \alpha(c) \sqsubseteq a$$

This is usually written: $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$. C represents the more complicated, concrete object, while A is the simplified, abstracted version. α is called the abstraction function, and γ the concretization function.

Example 2. We use the following order on intervals: $\forall x \in \mathcal{I}, \bot \sqsubseteq x$, and $[a,b] \sqsubseteq [c,d] \iff c \leq a \wedge b \leq d$. Using this definition, $(I, \sqsubseteq)$ is a poset. In our case, we have:

$$\alpha(\emptyset) = \bot \qquad \gamma(\bot) = \emptyset$$
$$\alpha(X) = [\inf X; \sup X] \qquad \gamma([a,b]) = \{\, x \in \mathbb{Z} \mid a \leq x \leq b \,\}$$

We have the following Galois connection: $(\mathcal{P}(\mathbb{Z}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{I}, \sqsubseteq)$.

Proof. Let $[a,b] \in \mathcal{I}, C \in \mathcal{P}(\mathbb{Z})$.

$$\begin{aligned} C \subseteq \gamma([a,b]) &\iff \forall c \in C, c \in \gamma([a,b]) \iff \forall c \in C, a \leq c \leq b \\ &\iff a \leq \inf C \wedge \sup C \leq b \iff \alpha(C) \sqsubseteq [a,b] \end{aligned}$$

The case of $\bot$ is straightforward: $C \subseteq \gamma(\bot) \iff C = \emptyset \iff \alpha(C) \sqsubseteq \bot$ □

Now, we formally define what a sound abstract operator is:

Definition 3 (Sound operator abstraction)**.** Let γ be a concretization function from an abstract domain $(A, \sqsubseteq)$ to a concrete domain $(C, \leq)$. Let $f : C \to C$, and $g : A \to A$. We say that g is a sound abstraction of f if $\forall a \in A, f(\gamma(a)) \leq \gamma(g(a))$. Intuitively, this condition says that the concretized result of the abstract operator is an overapproximation of the concrete operator.

We can use the definition of Galois connections to see that the best abstraction of an operator f is $\alpha \circ f \circ \gamma$, thus getting a constructive way to define the best abstraction operators.

Example 4. Let us consider $f(X,Y) = \{\, x + y \mid x \in X, y \in Y \,\}$. Let $[a,b], [c,d] \in \mathcal{I}$ (the cases where at least one argument is $\bot$ are straightforward).

$$\begin{aligned} \alpha(f(\gamma([a,b]), \gamma([c,d]))) &= \alpha(\{\, x + y \mid a \leq x \leq b \wedge c \leq y \leq d \,\} \\ &= \alpha(\{\, x + y \mid a + c \leq x + y \leq b + d \,\}) = [a + c, b + d] \end{aligned}$$

So, $f^\sharp([a,b],[c,d]) = [a + c, b + d]$ is the best abstraction of the addition on intervals. There are other sound abstractions of the addition: for example, $\tilde{f}(\_,\_) = [-\infty, +\infty]$ is one, but it is less precise.

Another important concept of abstract interpretation is widening. Widening is an operator accelerating the computation of fixpoints. It does so by extrapolating what it knows. For example, let us consider a while loop incrementing a variable i while $i < 1000$, we would like to automatically compute the value of i at the end of the loop, assuming that i is initialized to 0. To do so, we would like to find a *loop invariant*, that is a predicate on i holding whenever the loop body is executed. Supposing that we have a loop invariant I, we want the following equation to hold:

$$I \sqsupseteq [i = 0] \cup \mathbb{S}[\![i = i + 1]\!](\mathbb{C}[\![i < 1000]\!]I)$$

The notation $\mathbb{S}[\![stmt]\!]S$ is the semantics of *stmt*, it executes the effect of *stmt* on the memory state S. $\mathbb{C}[\![expr]\!]$ filters the input environment to keep the states satisfying *expr*. We can see that I is a set of memory environments that contains the initial environment (i = 0), and is stable when the loop body is executed (that is, each time the condition is verified and the statement is executed). In particular $I = 0 \leq i \leq 1000$ is a loop invariant, and the tightest one. We can notice that the computation of the tightest loop invariant can be seen as a *least fixpoint*: I is the least fixpoint of $\lambda X.[i = 0] \cup \mathbb{S}[\![i = i + 1]\!](\mathbb{C}[\![i < 1000]\!]X)$. Thus, the semantics of a while loop can be defined as:

$$\mathbb{S}[\![\texttt{while } e \texttt{ do } s \texttt{ done}]\!]R = \mathbb{C}[\![\neg e]\!](\mathrm{lfp}\, \lambda S.R \cup \mathbb{S}[\![s]\!](\mathbb{C}[\![e]\!]S))$$

Using a result called Kleene's theorem, we know that we can compute the above least fixpoint as $\mathrm{lfp}\, F = \cup_{i \in \mathbb{N}} F^i(\emptyset)$. Intuitively, this corresponds to executing the body of the loop at most n times, and stopping when the result is stable (here, at $n = 1000$). However, this computation requires the loop body to be analyzed 1000 times, which is inefficient (even more so if the number of iterations is unbounded). We now define what a widening operator is, we define one on intervals, and we show how it helps in computing loop invariants.

Definition 5 (Widening operator)**.** $\nabla : A \times A \to A$ is a widening operator on an abstract domain $(A, \sqsubseteq, \sqcup)$ if:

- it overapproximates the join: $\forall x, y \in A, x \sqcup y \sqsubseteq x \nabla y$;
- it forces convergence in finite time: for any sequence $(y_i)_{i \in \mathbb{N}}$, the sequence $x_0 = y_0$, $x_{n+1} = x_n \nabla y_{n+1}$ converges: $\exists k, \forall n \geq k, x_k = x_n$

Example 6. We define the following widening operator on intervals:

$$[a, b] \nabla [c, d] = \left[\begin{cases} a & \text{if } a \leq c \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ +\infty & \text{otherwise} \end{cases} \right]$$

Now, going back to the example of the while loop, let $F(X) = [0, 0] \cup \mathbb{S}^\sharp[\![i = i + 1]\!](\mathbb{C}^\sharp[\![i < 1000]\!]X)$ be the semantical interpretation of the while loop, and $y_i = F^i(\emptyset)$. Using the notations of the previous definition, we have $y_i = [0, min(i, 1000)]$. Applying the widening, we get: $x_0 = [0, 0]$, $x_1 = x_0 \nabla y_1 = [0, 0] \nabla [0, 1] = [0, +\infty]$, and $\forall j \geq 1, x_j = x_1$. So, $[0, +\infty]$ is a valid overapproximation of the values of i while in the loop, and we get that after the loop $i \in [1000, +\infty]$. To gain back precision, we can perform a decreasing iteration: we inject x_1 back into the loop body F. We see that to enter the loop, $i \in [0, 999]$. After the incrementation, $i \in [1, 1000]$, and after applying the loop guard $i \in [1000, 1000]$. Thus, the widening operator accelerates convergence of our computations. This may come with a loss in precision, but in this example, we have seen that we can recover the precise result we wanted using a decreasing iteration.

2.2 Static Type Analysis of a Simple Functional Language

Now that we have defined the concepts of Galois connection and soundness of an abstract operator, we show that the semantics of typing of a simple functional language can be seen as an abstraction of the concrete semantics of that same language. This connection is studied in great length in [10], we present a simplified case here. Contrary to the previous part where we derived the abstract semantics (on intervals) from the concrete one (on $\mathcal{P}(\mathbb{Z})$), we start by defining the concrete and the abstract semantics before introducing a concretization function and showing the soundness of the abstract semantics with respect to the concrete one.

We study a simple functional language over integers, called FUN, and defined as $\mathcal{E}$ in Figure 2. An expression of FUN is either a variable, a binding of a variable to an expression, the application of an expression to another one, an integer, the sum of two expressions, or a conditional expression.

$$\begin{array}{ll} \mathcal{E} ::= x \in \mathbb{V} \mid \text{fun } x \to \ e \mid e_1\ e_2 \mid z \in \mathbb{Z} \mid e_1 + e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 & \\ \mathcal{V} ::= z \in \mathbb{Z} \mid \lambda x.f \mid \omega & \Sigma = \mathbb{V} \to \mathcal{V} \\ \mathcal{T} ::= int \mid t_1 \to t_2 & \Delta = \mathbb{V} \to \mathcal{T} \end{array}$$

Figure 2: Definition of the FUN functional language

The values of FUN are elements of $\mathcal{V}$: they are semantical objects, consisting in integers, functions, or errors (for example if we add a function to an integer). Environments over values are written $\sigma \in \Sigma$, they are just mapping from variables to values.

We define the semantics of FUN in a denotational style in Fig 3. By definition of the environments, the value of a variable x in a given environment σ is $\sigma(x)$. Given an environment σ, a function binding variable x to an expression e is interpreted as a function taking an input argument i returning the interpretation of e, in a modified environment σ where x is mapped to i. This modification of the environment corresponds to the substitution by i of x. The semantics of the application is simple: if the first expression e_1 is interpreted as a function, then this function is called with parameter the interpretation of the second expression e. Otherwise, an error is thrown. The semantics of an integer is the corresponding integer, while the interpretation of the addition of two expressions

$$
\begin{aligned}
&\mathbb{S}[\![\, expr \,]\!] : \Sigma \to \mathcal{V} \\
&\mathbb{S}[\![\, x \in \mathbb{V} \,]\!]\sigma = \sigma(x) \\
&\mathbb{S}[\![\, \text{fun } x \to\ e \,]\!]\sigma = \lambda i.\mathbb{S}[\![\, e \,]\!]\sigma[x \mapsto i] \\
&\mathbb{S}[\![\, e_1\ e_2 \,]\!]\sigma = \begin{cases} f[x \mapsto \mathbb{S}[\![\, e_2 \,]\!]\sigma] & \text{if } \mathbb{S}[\![\, e_1 \,]\!]\sigma = \lambda x.f \\ \omega & \text{otherwise} \end{cases} \\
&\mathbb{S}[\![\, z \in \mathbb{Z} \,]\!]\sigma = z \\
&\mathbb{S}[\![\, e_1 + e_2 \,]\!]\sigma = \begin{cases} z_1 + z_2 & \text{if } \mathbb{S}[\![\, e_1 \,]\!]\sigma = z_1 \in \mathbb{Z} \text{ and } \mathbb{S}[\![\, e_2 \,]\!]\sigma = z_2 \in \mathbb{Z} \\ \omega & \text{otherwise} \end{cases} \\
&\mathbb{S}[\![\, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \,]\!]\sigma = \begin{cases} \mathbb{S}[\![\, e_2 \,]\!]\sigma & \text{if } \mathbb{S}[\![\, e_1 \,]\!]\sigma = z \text{ and } z \neq 0 \\ \mathbb{S}[\![\, e_3 \,]\!]\sigma & \text{if } \mathbb{S}[\![\, e_1 \,]\!]\sigma = z \text{ and } z = 0 \\ \omega & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 3: Definition of the concrete semantics of FUN

$$
\begin{aligned}
&\mathbb{T}[\![\, expr \,]\!] \in \mathcal{P}(\mathcal{T} \times \Delta) \\
&\mathbb{T}[\![\, x \in \mathbb{V} \,]\!] = \{\, \delta(x), \delta \mid \delta \in \Delta \,\} \\
&\mathbb{T}[\![\, \text{fun } x \to\ e \,]\!] = \{\, m_x \to m_e, \delta \mid (m_e, \delta[x \mapsto m_x]) \in \mathbb{T}[\![\, e \,]\!], m_x \in \mathcal{T} \,\} \\
&\mathbb{T}[\![\, e_1\ e_2 \,]\!] = \{\, m_o, \delta \mid (m_2, \delta) \in \mathbb{T}[\![\, e_2 \,]\!] \wedge (m_2 \to m_o, \delta) \in \mathbb{T}[\![\, e_1 \,]\!] \,\} \\
&\mathbb{T}[\![\, z \in \mathbb{Z} \,]\!] = \{\, int, \delta \mid \delta \in \Delta \,\} \\
&\mathbb{T}[\![\, e_1 + e_2 \,]\!] = \{\, int, \delta \mid (int, \delta) \in \mathbb{T}[\![\, e_1 \,]\!] \cap \mathbb{T}[\![\, e_2 \,]\!] \,\} \\
&\mathbb{T}[\![\, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \,]\!] = \{\, (m, \delta) \mid (int, \delta) \in \mathbb{T}[\![\, e_1 \,]\!] \wedge (m, \delta) \in \mathbb{T}[\![\, e_2 \,]\!] \cap \mathbb{T}[\![\, e_3 \,]\!] \,\}
\end{aligned}
$$

Figure 4: Definiton of the abstract semantics of FUN

e_1 and e_2 is the sum of these interpretations, provided that both interpretations result in integers (otherwise, an error is thrown). The semantics of conditional statements is straightforward: an error is raised if the conditional expression is not an integer, and otherwise any non-zero integer is interpreted as true, resulting in the execution of the first branch, whereas zero is interpreted as false.

Let us try our semantics on (fun $x \to$ fun $y \to\ x$) 42 43:

$$
\begin{aligned}
\mathbb{S}[\![\, \text{fun } x \to \text{ fun } y \to\ x \,]\!]\sigma &= \lambda x_i.\mathbb{S}[\![\, \text{fun } y \to\ x \,]\!]\sigma[x \mapsto x_i] \\
&= \lambda x_i.\lambda y_i.\mathbb{S}[\![\, x \,]\!]\sigma[x \mapsto x_i, y \mapsto y_i] \\
&= \lambda x_i.\lambda y_i.x_i \\
\mathbb{S}[\![\, (\text{fun } x \to \text{ fun } y \to\ x)\ 42\ 43 \,]\!]\emptyset &= (\lambda x_i.\lambda y_i.x_i)\ 42\ 43 = 42
\end{aligned}
$$

Now, we define the semantics of types for FUN in Fig 4. It consists in sets of types and environment types (defined as $\mathcal{T}$ and Δ in Figure 2) in which a given expression will not raise any error ω. A variable x has type $\delta(x)$, for every possible type environment $\delta \in \Delta$. The type of a function is an arrow type $m_x \to m_e$, whenever e has type m_e in the type environment δ where x is mapped to type m_x. Assuming that e_2 has type m_2 and e_1 has type $m_2 \to m_o$, the application of expressions $e_1\ e_2$ has type m_o. The type of an integer is int, whatever $\delta \in \Delta$ is. The type of an addition of two expressions is int, provided that the two expressions are integers as well. The type of a conditional if e_1 then e_2 else e_3 is m, whenever m is the type of both e_2 and e_3, and whenever e_1 is a valid condition (i.e, e_1 has the integer type).

Going back to the example of (fun $x \to$ fun $y \to x$) 42 43, we get:

$$\begin{aligned}
\mathbb{T}[\![\,\text{fun } y \to x\,]\!] &= \{\,(m_y \to \delta(x), \delta) \mid \delta \in \Delta, m_y \in \mathcal{T}\,\} \\
\mathbb{T}[\![\,\text{fun } x \to \text{fun } y \to x\,]\!] &= \{\,(m_x \to m_e, \delta) \mid (m_e, \delta[x \mapsto m_x]) \in \mathbb{T}[\![\,\text{fun } y \to x\,]\!], m_x \in \mathcal{T}\,\} \\
&= \{\,(m_x \to m_e, \delta) \mid m_e = m_y \to m_x, \delta \in \Delta, m_x, m_y \in \mathcal{T}\,\} \\
&= \{\,(m_x \to m_y \to m_x, \delta) \mid \delta \in \Delta, m_x, m_y \in \mathcal{T}\,\} \\
\mathbb{T}[\![\,(\text{fun } x \to \text{fun } y \to x)\ 42\,]\!] &= \{\,(m_o, \delta) \mid (m_2, \delta) \in \mathbb{T}[\![\,42\,]\!] \wedge \\
&\qquad\qquad (m_2 \to m_o, \delta) \in \mathbb{T}[\![\,\text{fun } x \to \text{fun } y \to x\,]\!]\,\} \\
&= \{\,(m_o, \delta) \mid (int \to m_o, \delta) \in \mathbb{T}[\![\,\text{fun } x \to \text{fun } y \to x\,]\!]\,\} \\
&= \{\,(m_y \to int, \delta) \mid \delta \in \Delta, m_y \in \mathcal{T}\,\} \\
\mathbb{T}[\![\,(\text{fun } x \to \text{fun } y \to x)\ 42\ 43\,]\!] &= \{\,(int, \delta) \mid \delta \in \Delta\,\}
\end{aligned}$$

We now study the concretization going from the abstract, type based world to the concrete, value-based one. We build the concretizations incrementally, starting from one between values and types. Using these concretizations, we will be able to prove that the semantics of types is a sound abstraction of the concrete semantics of FUN.

γ_{tv} concretizes types into sets of values. As expected, the concretization of integers is the set of integers, while the concretization of functions from t_1 to t_2 is defined recursively as the set of functions that given a value of type t_1 return a value of type t_2.

$$\begin{aligned}
&\gamma_{tv} : \mathcal{T} \to \mathcal{P}(\mathbb{V}) \\
&\gamma_{tv}(int) = \mathbb{Z} \\
&\gamma_{tv}(t_1 \to t_2) = \{\,\lambda x.f \mid \forall v_1 \in \gamma_{tv}(t_1), f[x \mapsto v_1] \in \gamma_{tv}(t_2)\,\}
\end{aligned}$$

γ_{env} is just the lifting of γ_{tv} to environments (reminder: $\Delta = \mathbb{V} \to \mathcal{T}, \Sigma = \mathbb{V} \to \mathcal{V}$).

$$\gamma_{env} : \left\{ \begin{array}{ccl} \Delta & \to & \mathcal{P}(\Sigma) \\ \delta & \mapsto & \{\,\sigma \mid \forall x \in \mathbb{V}, \sigma(x) \in \gamma_{tv}(\delta(x))\,\} \end{array} \right.$$

$\gamma_{t,env}$ concretizes one type with a given type environment into functions mapping concrete environments to values that respect the type and the type environment. These functions are the semantical interpretations of FUN programs, so we are in a sense, concretizing a type with a type environment into a set of well-typed programs.

$$\gamma_{t,env} : \left\{ \begin{array}{ccl} \mathcal{T} \times \Delta & \to & \mathcal{P}(\Sigma \to \mathcal{V}) \\ (t, \delta) & \mapsto & \{\,\phi \mid \forall \sigma \in \gamma_{env}(\delta), \phi(\sigma) \in \gamma_{tv}(t)\,\} \end{array} \right.$$

γ concretizes sets of types in their environments to semantical interpretations of FUN programs. In particular, if $\gamma(X) = \emptyset$, this means that the set of types in their environments contains a contradiction. For example, if $(int, \delta) \in X$ and $(t_1 \to t_2, \delta) \in X$, there is no FUN expression having both types, so the result of the concretization is $\emptyset$. This previous example shows that defining γ as $\gamma(X) = \bigcup_{(t,\delta)\in X} \gamma_{t,env}(t, \delta)$ is not what we want, as the contradiction would not be taken into account. Instead, an intersection is necessary to handle the polymorphism that may be expressed by $\mathcal{P}(\mathcal{T} \times \Delta)$. For example, the function fun $x \to x$ has types $X = \{\,(t \to t, \delta) \mid \delta \in \Delta, t \in \mathcal{T}\,\}$. While (t, δ) ranges over X, the intersection refines the candidate functions given their types. For example, fun $x \to 0$ is part of $\gamma_{t,env}(int \to int, \delta)$, and is kept in the result at this point. However, when considering $((int \to int) \to (int \to int), \delta)$, fun $x \to 0$ will be ruled out.

$$\gamma : \left\{ \begin{array}{ccl} \mathcal{P}(\mathcal{T} \times \Delta) & \to & \mathcal{P}(\Sigma \to \mathcal{V}) \\ X & \mapsto & \bigcap_{(t,\delta)\in X} \gamma_{t,env}(t, \delta) \end{array} \right.$$

Now that γ is defined, we can establish the following soundness theorem:

Theorem 7. $\mathbb{T}[\![\cdots]\!]$ *is a sound approximation of* $\mathbb{S}[\![\cdots]\!]$*, i.e:*

$$\forall expr \in \mathcal{E}, \mathbb{S}[\![\, expr \,]\!] \in \gamma(\mathbb{T}[\![\, expr \,]\!])$$

Proof. By unfolding the definitions of γ and $\gamma_{t,env}$, this is equivalent to showing that:

$$\forall expr \in \mathcal{E}, \forall (t, \delta) \in \mathbb{T}[\![\, expr \,]\!], \forall \sigma \in \gamma_{env}(\delta), \mathbb{S}[\![\, expr \,]\!]\sigma \in \gamma_{tv}(t)$$

This proof is done by structural induction on the expressions.

- **Case** $\boldsymbol{x \in \mathbb{V}}$: let $(\delta(x), \delta) \in \mathbb{T}[\![\, x \,]\!], \sigma \in \gamma_{env}(\delta)$. Unfolding γ_{env}, we get that $\forall v \in \mathbb{V}, \sigma(v) \in \gamma_{tv}(\delta(v))$. In particular, we get that $\mathbb{S}[\![\, x \,]\!]\sigma = \sigma(x) \in \gamma_{tv}(\delta(x))$.

- **Case fun** $\boldsymbol{x \rightarrow\ e}$: let m_x, m_e, δ such that $(m_x \rightarrow m_e, \delta) \in \mathbb{T}[\![\, \text{fun } x \rightarrow\ e \,]\!]$ (in particular, $(m_e, \delta[x \mapsto m_x]) \in \mathbb{T}[\![\, e \,]\!]$). Let $\sigma \in \gamma_{env}(\delta)$, we want to prove that $\mathbb{S}[\![\, \text{fun } x \rightarrow\ e \,]\!]\sigma = \lambda x_i.\mathbb{S}[\![\, e \,]\!]\sigma[x \mapsto x_i] \in \gamma_{tv}(m_x \rightarrow m_e)$. We unfold the definition of γ_{tv}: let $v_x \in \gamma_{tv}(m_x)$, we need to show that $\mathbb{S}[\![\, e \,]\!]\sigma[x \mapsto v_x] \in \gamma_{tv}(m_e)$.

  By induction hypothesis (with $t, \delta = m_e, \delta[x \mapsto m_x]$), we get that $\forall \sigma' \in \gamma_{env}(\delta[x \mapsto m_x]), \mathbb{S}[\![\, e \,]\!]\sigma' \in \gamma_{tv}(m_e)$. As $v_x \in \gamma_{tv}(m_x)$, we get $\sigma[x \mapsto v_x] \in \gamma_{env}(\delta[x \mapsto m_x])$, so $\mathbb{S}[\![\, e \,]\!]\sigma[x \mapsto v_x] \in \gamma_{tv}(m_e)$.

- **Case** $\boldsymbol{e_1\ e_2}$: let m_o, m_2, δ such that $(m_o, \delta) \in \mathbb{T}[\![\, e_1\ e_2 \,]\!]$, and $(m_2, \delta) \in \mathbb{T}[\![\, e_2 \,]\!] \wedge (m_2 \rightarrow m_o, \delta) \in \mathbb{T}[\![\, e_1 \,]\!]$. Let $\sigma \in \gamma_{env}(\delta)$, we show that $\mathbb{S}[\![\, e_1\ e_2 \,]\!]\sigma \in \gamma_{tv}(m_o)$. The first induction hypothesis yields $\mathbb{S}[\![\, e_1 \,]\!]\sigma \in \gamma_{tv}(m_2 \rightarrow m_o)$ (as $(m_2 \rightarrow m_o, \delta) \in \mathbb{T}[\![\, e_1 \,]\!]$). Unfolding $\gamma_{tv}(m_2 \rightarrow m_o)$, we get that $\mathbb{S}[\![\, e_1 \,]\!]\sigma = \lambda x_2.f$, with $\forall v_2 \in \gamma_{tv}(m_2), f[x_2 \mapsto v_2] \in \gamma_{tv}(m_o)$. Using the second induction hypothesis, we have: $\mathbb{S}[\![\, e_2 \,]\!]\sigma \in \gamma_{tv}(m_2)$ (as $(m_2, \delta) \in \mathbb{T}[\![\, e_2 \,]\!]$). Thus $\mathbb{S}[\![\, e_1\ e_2 \,]\!]\sigma = f[x_2 \mapsto \mathbb{S}[\![\, e_2 \,]\!]\sigma] \in \gamma_{tv}(m_o)$.

- **Case** $\boldsymbol{z \in \mathbb{Z}}$: let $(int, \delta) \in \mathbb{T}[\![\, z \,]\!]$, $\sigma \in \gamma_{env}(\delta)$. $\mathbb{S}[\![\, z \,]\!]\sigma = z \in \gamma_{tv}(int)$.

- **Case** $\boldsymbol{e_1 + e_2}$: let $(int, \delta) \in \mathbb{T}[\![\, e_1 + e_2 \,]\!] \cap \mathbb{T}[\![\, e_1 \,]\!] \cap \mathbb{T}[\![\, e_2 \,]\!]$. Let $\sigma \in \gamma_{env}(\delta)$. By induction hypothesis, for all $i \in \{\, 1, 2 \,\}$, $\mathbb{S}[\![\, e_1 \,]\!]\sigma \in \gamma_{tv}(int)$. Thus $\mathbb{S}[\![\, e_1 + e_2 \,]\!]\sigma = \mathbb{S}[\![\, e_1 \,]\!]\sigma + \mathbb{S}[\![\, e_2 \,]\!]\sigma \in \gamma_{tv}(int)$.

- **Case if** $\boldsymbol{e_1}$ **then** $\boldsymbol{e_2}$ **else** $\boldsymbol{e_3}$: let $(m, \delta) \in \mathbb{T}[\![\, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \,]\!]$. We additionally know that $(int, \delta) \in \mathbb{T}[\![\, e_1 \,]\!]$, and $(m, \delta) \in \mathbb{T}[\![\, e_2 \,]\!] \cap \mathbb{T}[\![\, e_3 \,]\!]$. Let $\sigma \in \gamma_{env}(\delta)$. By induction hypothesis, we have that $\mathbb{S}[\![\, e_1 \,]\!]\sigma \in \gamma_{tv}(int)$, and $\forall i \in \{\, 2, 3 \,\}, \mathbb{S}[\![\, e_i \,]\!]\sigma \in \gamma_{tv}(m)$. Thus, there exists $i \in \{\, 2, 3 \,\}$ such that $\mathbb{S}[\![\, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \,]\!]\sigma = \mathbb{S}[\![\, e_i \,]\!]\sigma$. In both cases, $\mathbb{S}[\![\, e_i \,]\!]\sigma \in \gamma_{tv}(m)$

□

Summary This section introduced the main concepts of abstract interpretation, namely *Galois connections*, *sound abstract operators* and *widening*. Subsection 2.1 focused on the abstract domain of intervals $\mathcal{I}$, using a Galois connection, while Subsection 2.2 focused on proving that the semantics of typing was a sound abstraction of the concrete semantics of FUN, using only a concretization function. A useful feature that was used a lot in the last part was to build the concretization functions incrementally and combine them. We will continue to use this incremental approach for both the concretization functions and the definition of abstract operators when defining the type analysis for Python, but we start by introducing the concrete semantics of Python.

3 Concrete Semantics of Python

The semantics of Python is officially defined as the implementation of Python's official interpreter, called CPython. It is thus not a formal semantics, but just the C code defining CPython. We use a slight evolution from the semantics proposed by Fromherz et al. in [13]. We have changed the semantics to get closer to the real Python language. In the previous semantics, builtin objects such as integers were considered as just a primitive value, without any object structure. A consequence of this restriction was that creating a class inheriting from builtin classes was impossible. In this new semantics, every Python object is represented as a primitive value combined

with a representation of the object, so the mentioned limitation is now lifted. We start by defining the memory environment on which a Python program acts, and describe a few parts of the concrete semantics of Python. The rest of the semantics is presented in Appendix B.

The memory environment consists in two parts: the environment $\mathcal{E}$ and the heap $\mathcal{H}$, formally defined in Figure 5. The environment $\mathcal{E}$ is a finite map from variable identifiers **Id** to addresses **Addr**. Due to the scope of variables in Python, variable identifiers may also be locally undefined `Undef`. The heap $\mathcal{H}$ maps addresses to tuples of objects **Obj** and primitive values **Val**. Although everything is object in Python, we need to keep track of primitive values and store them in the heap. Primitive values are either integers, or strings, or booleans, the `None` value, or the `NotImplemented` value. Objects are finite maps from strings (corresponding to attributes and methods) to addresses. To keep track of the complex, non-local control flow of Python programs, while defining semantics by induction on the syntax, we use continuations: in addition to keeping the current state of the analysis, states found before a jump point in the control flow will be kept for later use (for example, the state when an exception is `raised` will be kept for analysis of later `except` statements). To store those continuations in the program states, we label states using *flow tokens* (elements of $\mathcal{F}$), so the states we consider are elements of $\mathcal{P}(\mathcal{F} \times \mathcal{E} \times H)$ (the semantics of raise and try/except illustrate how these flow tokens may be used, in Figure 24 and Figure 25, Appendix B, page 32 and 33). Flow token *cur* represents the current flow on which most instructions operate. *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.

$$\mathcal{E} \stackrel{\text{def}}{=} \mathbf{Id} \rightharpoonup \mathbf{Addr} \cup \texttt{Undef} \qquad \mathbf{Id} \subseteq \mathit{string}$$
$$\mathcal{H} \stackrel{\text{def}}{=} \mathbf{Addr} \rightharpoonup \mathbf{Obj} \times \mathbf{Val} \qquad \mathbf{Obj} \stackrel{\text{def}}{=} \mathit{string} \rightharpoonup \mathbf{Addr}$$
$$\mathcal{F} \stackrel{\text{def}}{=} \{\, \mathit{cur}, \mathit{ret}, \mathit{brk}, \mathit{cont}, \mathit{exn} \,\} \qquad \mathbf{Val} \stackrel{\text{def}}{=} \mathbb{Z} \cup \mathit{string} \cup \{\texttt{True}, \texttt{False}, \texttt{None}, \texttt{NotImpl}\}$$

Figure 5: Environment of Python programs

We denote by $\mathbb{E}[\![\, e \,]\!]$ the semantics of expression e. This semantics has the following signature: $\mathbb{E}[\![\, e \,]\!] : \mathcal{F} \times \mathcal{E} \times \mathcal{H} \to \mathcal{F} \times \mathcal{E} \times \mathcal{H} \times \mathbf{Addr}$, so $\mathbb{E}[\![\, e \,]\!]$ returns the address where the object associated with e has been allocated. The semantics of statements is written $\mathbb{S}[\![\, s \,]\!]$ and has for signature $\mathcal{F} \times \mathcal{E} \times \mathcal{H} \to \mathcal{F} \times \mathcal{E} \times \mathcal{H}$. Note that in the following, both semantics may be implicitly lifted to the powerset.

To illustrate the complexity of Python's semantics, we define the semantics of the addition of two expressions e_1 and e_2 in Figure 6. Figure 6 uses the following notation: "letif $(f, \epsilon, \sigma, a) =$ ... in ", which unfolds into "let $(f, \epsilon, \sigma, a) =$ ... in if $f \neq \mathit{cur}$ then (f, ϵ, σ, a) else ". If the flow token asks for the current evaluation, Python starts by evaluating the expressions e_1 and e_2. If both expressions evaluate correctly, CPython tries to call the method $\_\_add\_\_$ of the left argument. If it does not exist or is not implemented, and the types of the evaluated expressions of e_1 and e_2 are not the same, the interpreter tries to call the reflected method $\_\_radd\_\_$ of the right argument. If none of this works (or the returned result is that the function is not implemented) a `TypeError` is raised. To show how the flows are handled, we also define the semantics of a variable assignment $id = e$ in Figure 6. The interpreter starts by evaluating the expression e; it then returns the non-current flows found during the evaluation of e, as well as the current flow where the environment is modified to bind id to the address a of the object representing the evaluation of e.

4 Type Analysis for Python

We define the type abstract domain, before establishing a concretization from the type abstract domain into the concrete environment. We finish by defining some classical abstract operators needed for the type analysis. We opted for an in-depth description of the concretization functions and the abstract operators in subsections 4.2 and 4.3 that we believe is necessary to accurately explain our analysis.

$$
\begin{array}{l}
\mathbb{E}[\![\, e_1 + e_2 \,]\!]\,(f, \epsilon, \sigma) \stackrel{\text{def}}{=} \\
\quad \text{if } f \neq \mathit{cur} \text{ then } (f, \epsilon, \sigma, \mathit{addr}_{\texttt{None}}) \text{ else} \\
\quad \text{letif } (f_1, \epsilon_1, \sigma_1, a_1) = \mathbb{E}[\![\, e_1 \,]\!]\,(f, \epsilon, \sigma) \text{ in} \\
\quad \text{letif } (f_2, \epsilon_2, \sigma_2, a_2) = \mathbb{E}[\![\, e_2 \,]\!]\,(f_1, \epsilon_1, \sigma_1) \text{ in} \\
\quad \text{if } \mathit{has\_field}(a_1, \_\_\mathit{add}\_\_, \sigma_2) \text{ then} \\
\qquad \text{letif } (f_3, \epsilon_3, \sigma_3, a_3) = \mathbb{E}[\![\, a_1.\_\_\mathit{add}\_\_(a_2) \,]\!]\,(f_2, \epsilon_2, \sigma_2) \text{ in} \\
\qquad \text{if } \sigma_3(a_3) = (\_, \texttt{NotImpl}) \text{ then} \\
\qquad\quad \text{if } \mathit{has\_field}(a_2, \_\_\mathit{radd}\_\_, \sigma_3) \wedge \mathit{typeof}(a_1) \neq \mathit{typeof}(a_2) \text{ then} \\
\qquad\qquad \text{letif } (f_4, \epsilon_4, \sigma_4, a_4) = \mathbb{E}[\![\, a_2.\_\_\mathit{radd}\_\_(a_1) \,]\!]\,(f_3, \epsilon_3, \sigma_3) \text{ in} \\
\qquad\qquad \text{if } \sigma_4(a_4) = (\_, \texttt{NotImpl}) \text{ then } \texttt{TypeError}(f_4, \epsilon_4, \sigma_4) \\
\qquad\qquad \text{else } (f_4, \epsilon_4, \sigma_4, a_4) \\
\qquad\quad \text{else } \texttt{TypeError}(f_3, \epsilon_3, \sigma_3) \\
\qquad \text{else } f_3, \epsilon_3, \sigma_3, a_3 \\
\quad \text{else if } \mathit{has\_field}(a_2, \_\_\mathit{radd}\_\_, \sigma_2) \wedge \mathit{typeof}\, a_1 \neq \mathit{typeof}\, a_2 \text{ then} \\
\qquad \text{letif } (f_3, \epsilon_3, \sigma_3, a_3) = \mathbb{E}[\![\, a_2.\_\_\mathit{radd}\_\_(a_1) \,]\!]\,(f_2, \epsilon_2, \sigma_2) \text{ in} \\
\qquad \text{if } \sigma_3(a_3) = (\_, \texttt{NotImpl}) \text{ then } \texttt{TypeError}(f_3, \epsilon_3, \sigma_3) \\
\qquad \text{else } (f_3, \epsilon_3, \sigma_3, a_3) \\
\quad \text{else } \texttt{TypeError}(f_2, \epsilon_2, \sigma_2)
\end{array}
$$

$$
\begin{array}{l}
\mathbb{S}[\![\, \mathit{id} = e \,]\!]\, S \stackrel{\text{def}}{=} \\
\quad \text{let } S_e, A_e = \mathbb{E}[\![\, e \,]\!]\, S \text{ in} \\
\quad \{\, (f, \epsilon, \sigma) \mid (f, \epsilon, \sigma) \in S_e \wedge f \neq \mathit{cur} \,\} \cup \\
\quad \{\, (\mathit{cur}, \epsilon[\mathit{id}] \leftarrow a, \sigma) \mid (\mathit{cur}, \epsilon, \sigma, a) \in (S_e, A_e) \,\}
\end{array}
$$

Figure 6: Excerpt of the concrete semantics of Python

In a given Python program, a Python variable may have different types, corresponding to the types of the possible values the variable may hold during any execution of the program, so we are interested in finding the *set* of possible types for each variable. Moreover, Python programs may catch errors later on during the execution: a program may raise a `TypeError` in a given context, but this `TypeError` may be caught later on by an `except` statement, so we need to *collect* types of programs. Thus, we are aiming at *collecting the set of types of each variable*, as well as the type error exceptions. In this regard, our goal is different from the type analysis presented in Section 2.2, which was accepting typable programs and rejecting the others.

4.1 Description of the Type Analysis

We start by describing the types a variable may possess. Then, we will define the abstract domain enabling a relational analysis with bounded parametric polymorphism, before giving some examples.

Types The analysis will use both polymorphic and monomorphic types. Polymorphic types are defined in Figure 7. Types can be:

– Top or bottom.

– Class definitions (Eq. (2)).

– Standard library containers (such as lists or tuples) holding elements of type v (Eq. (3)).

– Instances of another type v that *must* have some attributes given in AddedAttr and that *may* have some attributes defined in OptionalAttr. Both AddedAttr and OptionalAttr can be seen as sets of tuples, consisting in attributes with their types. This combination of both an underapproximation (must attributes) and an overapproximation (may attributes) is used to gain precision, and we explain why now. Without an underapproximation of the attributes, we would only know that an attribute *may* exist. Thus, each attribute access would yield a spurious alarm about an `AttributeError`, and the analysis would be imprecise. Concerning the need for an overapproximation, let us consider we analyze a conditional statement, where variable x is an instance of a class C that has an attribute a in one branch and variable x is an instance of the same class C without attribute a in the other branch. At the end of the if statement, and due to the flow-sensitivity of our analysis, we would like to have variable x keeping the "union" of both instances. In the concrete, this union would be performed on the two singletons, and would result in a set of length 2. However, this solution is not efficient when describing the abstract operations a static analyzer may perform. In our lattice of types, the abstract union is called the join. Here, if the join is performed (without keeping the overapproximation of attributes), we would get an instance of the class C (as we cannot keep any information on attribute a). This would have the unfortunate consequence of forcing the concretization of a given instance type to be this instance, with arbitrarily many attributes added, resulting in imprecise analyses. Using a combination of an underapproximation and an overapproximation of the attributes, we get that the join is an instance of class C, that *may* have attribute a. This resolves our precision issues and creates a more precise concretization function, so we keep both approximations.

– Type variables, written using Greek letters (Eq. (6)), will be used to express the bounded parametric polymorphism. They will represent sets of types. These type variables are useful to analyze part of Python programs that are polymorphic. For example, let us consider a program where variables x and y have the same type, being either `int` or `float` and storing the maximum of both variables into a variable z. Using type variables, we obtain a very compact representation stating that all three variables have type $\alpha \in \{\, Instance[int], Instance[float] \,\}$ – the set of monormophic types over which α ranges will be stored in a type variable environment. Moreover, if variables x and y are elements of lists (denoted respectively lx and ly), the type of lx and ly is quite concise: $List[\alpha]$.

– Function summaries (Eq. (5)) are used to strike a balance between a fully modular function analysis, and an inlining-based analysis. While the former analysis may be desirable, we would not know anything on the types of the input variables, and the analysis would be too imprecise. The inlining-based analysis offers the best precision but is less efficient. A compromise is to use summaries to memoize the results of the previous analyses of function f. For example, if we go back to the example of `os.fspath` (Figure 1), and fspath has already been analyzed with a string, the summary would be: $Summary[fspath, [(str, str)]]$. Now, if $fspath$ is called with an instance of a class Foo having a method $\_\_fspath\_\_$ returning a string, we would get: $Summary[fspath, [(str, str), (Instance[Foo, \_\_fspath\_\_], str)]]$. Afterwards, if $fspath$ is called with p being a string, the analysis has already been performed and we can return the result. If $fspath$ is called with p being an instance of *bytes*, the summary becomes:

$$Summary[fspath, [(\alpha, \alpha), (Instance[Foo, \_\_fspath\_\_], str)]] \text{ (with } \alpha \in \{\, str, bytes \,\})$$

The set of monomorphic types $V_m^\sharp$, is the set of polymorphic types $V^\sharp$, from which type variables V_Σ have been removed. We will see that type variables are associated to sets of monomorphic types.

Structure of the abstract domain The relationality and the bounded parametric polymorphism of our analysis is ensured by splitting our abstract domain into three parts, defined in Figure 8. The relationality of the analysis means that our analysis can express that two variables have the same types.

$$
\begin{aligned}
V^\sharp &= \bot \mid \top && (1)\\
&\mid c, c \in \mathit{Classes} && (2)\\
&\mid \mathit{List}[v] \mid \mathit{Tuple}[v] \mid \ \ldots, v \in V^\sharp && (3)\\
&\mid \mathit{Instance}[v, \mathit{AddedAttr}, \mathit{OptionalAttr}], v \in V^\sharp && (4)\\
&\mid \mathit{Summary}[f, (\mathit{InputArgs}, \mathit{OutputArgs})\ \mathit{list}] && (5)\\
&\mid V_\Sigma \in \{\, \alpha, \beta, \ldots \,\} && (6)
\end{aligned}
$$

Figure 7: Definition of polymorphic types

$$
\begin{aligned}
\mathcal{D}^\sharp &= \mathcal{D}_1^\sharp \times \mathcal{D}_2^\sharp \times \mathcal{D}_3^\sharp\\
\mathcal{D}_1^\sharp &= \mathbf{Id} \rightharpoonup \mathit{TypeId} \times \mathit{Aliasing}\\
\mathcal{D}_2^\sharp &= \mathit{TypeId} \rightharpoonup V^\sharp\\
\mathcal{D}_3^\sharp &= V_\Sigma \rightharpoonup \mathcal{P}(V_m^\sharp)
\end{aligned}
$$

Figure 8: Definition of the abstract domain

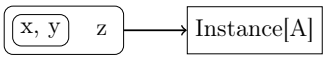
The first part, $\mathcal{D}_1^\sharp$, makes this abstract domain relational by packing variables having the same type into the same group. This partitioning by types is expressed using a mapping from variable identifiers to type identifiers ($\in \mathit{TypeId}$). These partitions can be seen as equivalence classes for the type equality relation. In each equivalence class, we refine the partitions by keeping which variables must alias to each other. In the following, we consider that $\mathit{TypeId} = \mathbb{N}$ and $\mathit{Aliasing} = \mathbb{N}_\bot = \mathbb{N} \cup \{\, \bot \,\}$, where $\bot$ stands for variables not aliasing to any other variable.

The second part, $\mathcal{D}_2^\sharp$, maps each partition (or each equivalence class) to a polymorphic type. The third part, $\mathcal{D}_3^\sharp$, is an environment for polymorphic type variables. It maps each polymorphic type variable ($\in V_\Sigma$) to a set of monomorphic types.

To get a well-formed abstract domain, we impose the following constraints: elements of $\mathcal{D}_2^\sharp$ and $\mathcal{D}_3^\sharp$ should be injective mappings. This forces the abstract domain to have an optimal representation. In particular the comparison with TypeId and equivalence class numbers can be further developed as the first constraint means that two equivalent elements should be in the same equivalence class.

Examples We give two examples of our abstract domain in Table 9.

Example code	Graphical representation	Abstract representation

`x = A()`
`y = x`
`if *: z = A()`
`else: z = y`
		d_1 $x \mapsto (0,0), y \mapsto (0,0), z \mapsto (0,\bot)$
		d_2 $0 \mapsto \mathit{Instance}[A]$

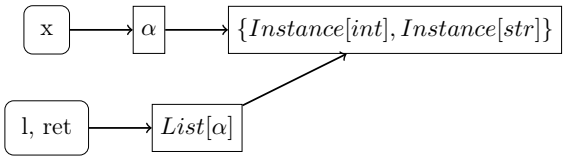
`def filter(f, l):`
`  ret = []`
`  for x in l:`
`  if f(x):`
`  ret.append(x)`
`  return ret`
		d_1 $x \mapsto (0,\bot), l \mapsto (1,\bot), \mathit{ret} \mapsto (1,\bot)$
		d_2 $0 \mapsto \alpha, 1 \mapsto \mathit{List}[\alpha]$
		d_3 $\alpha \mapsto \{\mathit{Instance}[\mathit{int}], \mathit{Instance}[\mathit{str}]\}$

Table 9: Examples of abstract elements

We assume that A is a user-defined class. In the first example, variables x and y alias to the same instance of the object A, while variable z is also an instance of A, but it may or may not be aliased to y, depending on the execution of the non-deterministic conditional statement. All three variables have the same type, so d_1 maps each variable to the same partition number which is 0 here. Variable z does not alias, so its aliasing coordinate is $\bot$. The concrete environment corresponding to this abstract element is the set of memory environments $(x \mapsto a, y \mapsto a, z \mapsto a_z)$, and the heap contains at most two distinct addresses a and a_z pointing to instances of class A.

The second example shows the use of bounded parametric polymorphism. This situation may happen while

analyzing a function filtering elements of a list, defined in the first column of Table. 9. l is the input list, x is an element of the list, and ret is the returned list. In that case, we know that x has type α, being either an integer or a string. On the other hand l and ret are either both lists of integers, or both lists of strings. We have two partitions, one being identified with number 0 and the other with number 1. A more concrete representation of this abstract element using only monormophic types is the following: $\{ x \mapsto Instance[int], l \mapsto List[Instance[int]], ret \mapsto List[Instance[int]] \} \cup \{ x \mapsto Instance[str], l \mapsto List[Instance[str]], ret \mapsto List[Instance[str]] \}$.

4.2 Concretization

Due to the structure of the abstract domain, we define the concretization functions incrementally.

We start by defining γ_v, concretizing monomorphic types into objects having the corresponding type, allocated on the heap at a given address: $\mathcal{P}(\mathbf{Addr} \times \mathcal{H})$. Then γ_{part} concretizes partitions in a given type variable environment (i.e, elements of $\mathcal{D}_2^\sharp \times \mathcal{D}_3^\sharp$) into maps from type identifiers to monotypes, by substituting each type variable by a possible monomorphic type. After that, γ concretizes elements of $\mathcal{D}^\sharp$ into memory environments. All three concretizations are defined in Figure 10.

The definition of γ_v stems from the structure of the heap. We define an operator computing addresses and heaps in which an expression e is stored: $\tilde{\mathbb{E}}[\![e]\!]h = \{ (a, h) \mid (cur, \_, h, a) \in \mathbb{E}[\![e]\!](cur, \epsilon, h') \}$. The concretization of an instance of the integer class is any element of $\tilde{\mathbb{E}}[\![z]\!]h'$, with $z \in \mathbb{Z}, h' \in \mathcal{H}$. The concretization of a list holding elements of type $v^\sharp$ is the evaluation of any list of arbitrary size, having elements being of type $v^\sharp$. $\sqsubseteq_c$ is the inheritance order: a class c is less than d if c inherits from d.

The definition of γ_{part} is more complex, as we want the substitution of type variables to be global on elements of $\mathcal{D}_2^\sharp$. If we consider the second example of Figure 9, we want to get two functions: $\gamma_{part}(d_2, d_3) = \{ 0 \mapsto Instance[int], 1 \mapsto List[Instance[int]] \} \cup \{ 0 \mapsto Instance[str], 1 \mapsto List[Instance[str]] \}$. However, we do not want $\{ 0 \mapsto Instance[str], 1 \mapsto List[Instance[int]] \}$ to be an element of our concretization function, as this would destroy the purpose of type variables.

In the definition of γ, we want to ensure two things. The first one is that each variable identifier has a good

$$\gamma_v : \begin{cases} V_m^\sharp & \to & \mathcal{P}(\mathbf{Addr} \times \mathcal{H}) \\ \bot & \mapsto & \emptyset \\ \top & \mapsto & \mathbf{Addr} \times \mathcal{H} \\ Class\ c & \mapsto & \{ (a, h) \mid h(a) = (b, \bot), b \sqsubseteq_c c \} \\ Instance[int] & \mapsto & \cup_{h \in \mathcal{H}, z \in \mathbb{Z}} \tilde{\mathbb{E}}[\![int(z)]\!]\, h \\ Instance[Class\ c, (u_1 : \tau_1, \ldots, u_n : \tau_n), \emptyset] & \mapsto & \{ (a, h) \mid (a, h) \in \tilde{\mathbb{E}}[\![\tilde{c} = b(); \tilde{c}.u_1 = t_1; \ldots; \tilde{c}.u_n = t_n; \tilde{c}]\!]h', \\ & & \quad h' \in \mathcal{H}, \forall 1 \leq i \leq n, (t_i, h) \in \gamma_v(\tau_i), b \sqsubseteq_c c \} \\ Instance[Class\ c, U, O] & \mapsto & \bigcup_{o \in \mathcal{P}(O)} \gamma_v(Instance[Class\ c, U \cup o, \emptyset]) \\ List[v^\sharp] & \mapsto & \bigcup_{\substack{i \in \mathbb{N}, h \in \mathcal{H} \\ (a_j, h)_{0 \leq j \leq i} \in \gamma_v(v^\sharp)}} \tilde{\mathbb{E}}[\![list(a_1, \ldots, a_i)]\!]h \end{cases}$$

$$\gamma_{part} : \begin{cases} \mathcal{D}_2^\sharp \times \mathcal{D}_3^\sharp & \to & \mathcal{P}(TypeId \to V_m^\sharp) \\ (d_2, d_3) & \mapsto & \bigcup_{\tau_i \in d_3(\alpha_i), \alpha_i \in V_\Sigma} \{ \lambda t. d_2(t)[\alpha_i \mapsto \tau_i] \} \end{cases}$$

$$\gamma : \begin{cases} \mathcal{D}^\sharp & \to & \mathcal{P}(\mathcal{E} \times \mathcal{H}) \\ (d_1, d_2, d_3) & \mapsto & \bigcup_{P \in \gamma_{part}(d_2, d_3)} \{ (e, h) \mid \forall i \in \mathbf{Id}, d_1(i) = (tid, \_) \implies (e(i), h) \in \gamma_v(P(tid)) \wedge \\ & & \quad \forall i, j \in \mathbf{Id}, d_1(i) = d_1(j) \wedge \mathrm{snd}\, d_1(i) \neq \bot \implies e(i) = e(j) \} \end{cases}$$

Figure 10: Definition of the concretization of the abstract domain

type, while the heap is the same. The second ensures that aliased variables in the abstract are aliased in the concrete memory environment, i.e, they have the same address. Note that aliasing is weak – or equivalently, that we perform a must-alias analysis: if two variables are not explicitly aliased, they may still alias in the concrete. Conversely, if two variables are aliased, they must alias in the concrete. In the case of the example mentioned above, the result of the concretization is the set of memory environments such that variables x, l and ret point to addresses, these addresses respectively pointing to: an integer, a list of integers and a list of integers, or to: a string, a list of strings and a second list of strings.

4.3 Abstract Operators

Similarly to concretizations, abstract operators are defined incrementally. However, abstract operators act more locally than the concretization function, so we can define them in a more modular way. As before, we start by working on monomorphic types ($V_m^\sharp$). We continue with polymorphic types in a provided type variable environment ($V^\sharp \times \mathcal{D}_3^\sharp$, rather than $\mathcal{D}_2^\sharp \times \mathcal{D}_3^\sharp$). Finally, we lift the operator definition to $\mathcal{D}^\sharp$.

Subset testing This operator is mainly used to check if the analysis of a loop is stabilized or not: in the abstract, the analysis of a loop consists in computing a limit of a function computing a widening. To check if the widening is stable, (using notations of Def. 5) we can weaken the test $x_{n+1} = x_n$ into $x_{n+1} \sqsubseteq x_n$ (as ∇ overapproximates $\sqsubseteq$).

To test if an element d of $D^\sharp$ is less than an element d', we check that for every variable v of d, the polymorphic type in d is less than the one in d'. The order on polymorphic types $\sqsubseteq_p$ is defined below on $V^\sharp \times \mathcal{D}_3^\sharp$, and the type variable environments are left implicit when they are unused.

- $\forall v \in V^\sharp, \bot \sqsubseteq_p v$ and $v \sqsubseteq_p \top$
- $\forall u, v \in V^\sharp, u \sqsubseteq_p v \implies List[u] \sqsubseteq_p List[v]$
- $\forall cl, cl' \in Classes, cl \sqsubseteq_c cl' \implies Class\ cl \sqsubseteq_p Class\ cl'$ (where $\sqsubseteq_c$ is the inheritance order for the classes)
- Let $c, c' \in v^\sharp$. $Instance[c, u, o] \sqsubseteq_p Instance[c', u', o']$ if the following conditions are met:
  - $c \sqsubseteq_p c'$
  - Given a tuple (a, τ) of an attribute and its type in u, (a, τ') exists in u' and $\tau \sqsubseteq_p \tau'$
  - Given a tuple (a, τ) in o, (a, τ') exists in u' or o' and $\tau \sqsubseteq_p \tau'$.
- Let $\alpha \in V_\Sigma, v \in V^\sharp, d_3, d_3' \in \mathcal{D}_3^\sharp$. If $\forall m \in d_3(\alpha), (m, d_3) \sqsubseteq_p (v, d_3')$, then $(\alpha, d_3) \sqsubseteq_p (m, d_3')$.
- Let $\alpha \in V_\Sigma, v \in V^\sharp, d_3, d_3' \in \mathcal{D}_3^\sharp$. If v is monormophic, and $v \in d_3'(\alpha)$, then $(v, d_3) \sqsubseteq_p (\alpha, d_3')$.
- Let $\alpha \in V_\Sigma, v \in V^\sharp, d_3, d_3' \in \mathcal{D}_3^\sharp$. If v is not monormophic, and $\forall u \in \gamma_p(v, d_3), (u, d_3) \sqsubseteq_p (\alpha, d_3')$, then $(v, d_3) \sqsubseteq_p (\alpha, d_3')$.

Remark 8 (Lattice of Classes, Nominal Typing). One structure we used during the definition of $\sqsubseteq_p$ is the lattice of classes, $(\text{Classes} \cup \{\bot\}, \sqsubseteq_c, \cup^c, \cap^c)$. In Python, the top element of this lattice is the `object` class, as every class inherits from `object`. The order defined by $\sqsubseteq_c$ is the following: everything is greater than $\bot$ and less than `object`, and a class c is less than c' if c inherits from c'. The join $\cup^c$ of c and c' is defined as the smallest (for $\sqsubseteq_c$) class r such that $c \sqsubseteq_c r$ and $c' \sqsubseteq_c r$, while the meet is the biggest of the two. Two graphs showing the inheritance relation for Python are shown in Appendix C, page 34.

Join We show how to define the abstract union operator, called the join. This join is used when multiple control-flow branches that have been analyzed separately need to be merged together (for example, at the end of a conditional statement). This does not happen in a classical type system, but is used in our analysis, due to its flow-sensitivity. The join operator is written $\cup^\sharp$. As the definition of the join on monomorphic types and polymorphic types in a given environment are really similar, we only define the latter, written $\cup^v$. $\cup^v$ is defined by case analysis:

- $(\bot, d_3) \cup^v (u, d'_3) = (u, d'_3) \cup^v (\bot, d_3) = (u, d'_3|_{Vars(u)})$, where $d'_3|_{Vars(u)}$ is the restriction of d'_3 keeping only the type variables used by u.
- $(\top, d_3) \cup^v (u, d'_3) = (u, d'_3) \cup^v (\top, d_3) = (\top, \emptyset)$
- $cl \cup^c cl' \neq object \implies (Class\ cl, d_3) \cup^v (Class\ cl', d'_3) = (Class(cl \cup^c cl'), \emptyset)$ where $\cup^c$ is the least common ancestor of two classes for the inheritance relation.
- $((u, d_3) \cup^v (v, d'_3) = (w, r_3)) \implies (List[u], d_3) \cup^v (List[v], d'_3) = (List[w], r_3)$
- Assuming we have two instances $Instance[Class\ cl, \{ (u_i, \tau_i^u) \}, \{ (o_i, \tau_i^o) \}]$ and $Instance[Class\ cl', \{ (u'_i, \tau_i'^u) \}, \{ (o'_i, \tau_i'^o) \}]$, of classes cl and cl' such that $cl \cup^c cl' \neq object$, the result of the join is an instance of a class $cl \cup^c cl'$. The underapproximation of the attributes is the set of attributes that must exist in each instance, with each type being the monormophic join of the two types. The overapproximation of the attributes are the other attributes defined in at least one of the two instances, with each type being the monomorphic join of the two types. The overapproximation of the attributes also contains attributes that are defined by cl or cl', but are not defined in $cl \cup^c cl'$. This set is defined as $A = \{ (a, \tau) \in attrs(cl \cup^c cl') \setminus (attr(cl) \cup attr(cl')) \}$. More formally: let $U = \{ u_i \} \cap \{ u'_i \}$ and $O = \{ o_i \} \cup \{ o'_i \} \cup (\{ u_i \} \cup \{ u'_i \}) \setminus U$. The join of two instances is then:

$$Instance[Class\ (cl \cup^c cl'), \bigcup_{\substack{u \in U, s.t. \\ u = u_i = u'_j}} \{ (u, \tau_i^u \cup^v \tau_j'^u) \}, \bigcup_{\substack{o \in O, a, a' \in \{ u, o \} \\ s.t.\ o = a_i = a'_j}} \{ (o, \tau_i^a \cup^v \tau_j'^a) \} \cup A]$$

- The join of two summaries of the same function $Summary[f, l_1]$ and $Summary[f, l_2]$ is $Summary[f, l_1 \cup^l l_2]$, where $\cup^l$ is the concatenation of two lists where duplicates are removed.
- If none of the cases above apply, we use the following fallback. In this case, u and v are too different types to be joined using one of the previous cases. We make use of the bounded parametric polymorphism to keep both types, by using a fresh type variable α: $(u, d_3) \cup^v (v, d'_3) = \alpha, [\alpha \mapsto \gamma_p(u, d_3) \cup \gamma_p(v, d'_3)]$. γ_p is the concretization of a polymorphic type, and its definition stems from the one of γ_{part}:

$$\gamma_p : \begin{cases} V^\sharp \times \mathcal{D}_3^\sharp & \longrightarrow & \mathcal{P}(V_m^\sharp) \\ (v, d_3) & \mapsto & \{ v[\alpha_i \mapsto \tau_i] \mid \alpha_i \in V_\Sigma \text{ appearing in } v,\ \tau_i \in d_3(\alpha_i) \} \end{cases}$$

Remark 9 (On $cl \cup^c cl' \neq object$). The condition $cl \cup^c cl' \neq object$, used in the cases of the join of two classes or two instances is used to avoid losing too much precision. If two classes have a common parent that is not object, we assume that those classes are still similar enough to be merged into one class. Otherwise, the fallback case will be used, and the two classes will be kept separately using a type variable. This condition could be weakened to perform the join only when a common parent is not "too far" from both classes, and where the fallback case using type variables would be used otherwise.

The proof of soundness of $\cup^v$ is available in Appendix A.1.

To compute the join of two of the types abstract elements (d_1, d_2, d_3) and (d'_1, d'_2, d'_3), we proceed roughly as follows. Given the partitions specified by d_1 and d'_1, we compute the intersection of those partitions and store them into r_1. Then, we iterate over each partition of r_1. As each partition of r_1 represents just one type identifier for d_1 and for d'_1, we name t and t' those type identifiers. We compute the join $(d_2(t), d_3) \cup^v (d'_2(t'), d'_3)$ of the types for that partition and store it. In the end, the following invariant should hold by construction of the intersection of the partitions:

$$\forall i \in \mathbf{Id}, (d_2(d_1(i)), d_3) \cup^v (d'_2(d'_1(i)), d'_3) \sqsubseteq_p (r_2(r_1(i)), r_3) \tag{7}$$

This basically means that for each variable identifier i, the type given by the partition r_1 is more general than the union of the types given by the partition of d_1 and d'_1.

The proof of soundness of the join operator on $\mathcal{D}^\sharp$ is given in Appendix A.2.

Meet The abstract counterpart of the intersection operator is called the meet. It is defined similarly to the join.

Adding a variable To add the fact that variable x has type t in a domain (d_1, d_2, d_3), we search (in d_2) if there is a partition having type t (i.e, we search for $i \in TypeId$ such that $d_2(i) = t$). If that is the case, we give back $(d_1[x \mapsto (i, \bot)], d_2, d_3)$. Otherwise, we create a new partition, having a fresh type identifier $j \in TypeId$, and yield $(d_1[x \mapsto (j, \bot)], d_2[j \mapsto t], d_3)$. A similar operator exists to add an equality constraint between two variables (and this time, aliasing has to be taken into account).

Widening As seen in Section 2.1, the widening operator is used to find in a finite number of iterations a (post)fixpoint of a transfer function, such as a loop. For example, if we want to know what type x has after executing $x = list(x)$ n times, we would like to avoid unrolling the loop n times and have an analysis that converges in a small number of steps. The widening operator enjoys an incremental construction similar to that of the join. We describe how to compute $x \nabla y$ informally. If some partitions given by y are not included in those of x, those partitions are collapsed into one, and mapped to the $\top$ type. Otherwise, if the partitions are stable but the types are not, we can define a widening on polymorphic types, moving unstable types to $\top$. As this widening operator is really unprecise when partitions are unstable, we delay the widening twice, as this may help stabilizing the partitions and result in a more precise analysis. For example, if we have to widen $List[\alpha], \alpha \in \{ Instance[int], Instance[str] \}$ with $List[List[Instance[int]]]$, the result will be $List[\top]$. Another option for this widening operator would be to limit the size of the sets in $\mathcal{D}_3^\sharp$: going back to previous example, and assuming that the limit is 4, we would have $List[\alpha] \nabla List[List[Instance[int]]] = List[\beta], \beta \in \{ Instance[int], Instance[str], List[Instance[int]] \}$. Then, $List[\beta] \nabla List[List[List[Instance[int]]]] = List[\top]$.

Filtering Given a condition, filtering operators split a domain into two parts: one where the given condition may be satisfied, and one where the condition may not be satisfied. We defined two filtering operators, handling expressions using `isinstance` or `hasattr`. These two filtering operators correspond to the two kinds of typing that may be used in Python, i.e, nominal and duck typing (as mentioned in the cover section). For example, given a domain d where variable x has a type $\alpha \in \{ Instance[bool], Instance[int], Instance[float] \}$, we would get that $filter\_instance(x, int, d) = d', d''$, where variable x has type $\alpha \in \{ Instance[bool], Instance[int] \}$ in d' and variable x has type $Instance[float]$ in d'' (because *bool* is a subclass of *int*).

We describe how the instance filtering works on polymorphic types. Given a polymorphic type in a type environment, and a class I, we output two tuples of polymorphic types in their type environment: the first tuple is a subset of the input where all elements are instances of class I, while the second tuple is a subset of the input where all elements are not instances of class I.

- If the input is $(\bot, d_3)$ or $(\top, d_3)$, the output is a copy of the input.
- To filter a list $(List[t], d_3)$, we check if the the list class is less than class I using the inheritance order $\sqsubseteq_c$: if that is the case, the result is $(List[t], d_3), (\bot, d_3)$, and otherwise the result is $(\bot, d_3), (List[t], d_3)$.
- Given an instance of a class C, if $C \sqsubseteq_c I$ the first tuple will contain the instance and the second will be at bottom. Otherwise, the two tuples are exchanged.
- Given a polymorphic type variable α in a context d_3, we proceed as follow: we start by concretizing (using γ_p) the type α, and then we filter this set. The result of this filtering is two sets of types, one being instances of I (called t), and the other not (called f). Then, we return the type, where the type variable environments have been changed to match t and f, i.e the result is: $(\alpha, d_3[\alpha \mapsto t]), (\alpha, d_3[\alpha \mapsto f])$.

The attribute filtering is quite simple too. Most cases are straightforward. To perform the filtering of classes, we just need to look up the attribute in the class and its parents. To filter an instance $I = Instance[c, u, o]$, we consider the following cases:

1. If the attribute is present in c, we return (I, d_3) and $(\bot, d_3)$.

2. Otherwise, if the attribute is present in u, we return (I, d_3) and $(\bot, d_3)$.

3. Otherwise, if the attribute is present in o, we only know that the attribute may be part of the instance, so we have to return (I, d_3) and (I, d_3).

4. In the last case, we know for sure that the attribute is not present, so we just return $(\bot, d_3)$ and (I, d_3).

The attribute filtering of a type variable enjoys a construction similar to the instance filtering.

4.4 Abstract Semantics

We do not present the abstract semantics of the whole analysis here. It is basically the concrete semantics of Python with abstract operators replacing the concrete ones. For example, parts where the concrete union was used now use the abstract join $\cup^\sharp$, and least fixpoint computations used for loops are replaced by the computation of a limit using a widening operator. In this analysis, the interesting part is the abstract domain on which the analysis relies, rather than the abstract semantics. We will additionally see in Section 5.1 why we do not need to implement again the abstract semantics of Python.

5 Implementation Details

5.1 Integration into MOPSA

I implemented the static analysis described in the previous section into a framework called MOPSA (Modular Open Platform for Static Analysis). MOPSA is a work in progress. Its goal is to simplify the construction of static analyzers by offering a modular design. Abstract domains permitting to analyze numerical values, control-flow, pointers, etc. are written separately and are combined during the analysis given a configuration specified by the user. Using this configuration, MOPSA passes statements to each abstract domain until one is capable of analyzing the given statement. MOPSA supports static analysis of different languages which currently are subsets of C and Python. In particular, Python's value analysis [13] has been implemented into MOPSA before my internship.

Due to the modularity of the abstract domains mentioned above, and the implementation of the value analysis of Python into MOPSA, I have been able to reuse some domains dealing with Python constructs. I implemented the abstract domain of types described in the previous section, and the semantics of variable assignments, but I was able to reuse abstract domains handling if statements, while loops, ... However, the implementation was not straightforward either: some abstract domains were not modular enough, so I have had generalize them (for some), or write an alternative abstract domain compatible with the type analysis. For example, the value analysis used quite concrete addresses (the addresses were abstracted into a finite – but big – set, to be sufficiently precise and to allow a computable analysis), while addresses are abstracted into type identifiers during our analysis. Some abstract domains were not modular in the addresses, and required the addresses used by the value analysis, so I have had to adapt them.

The implementation of the type analysis into MOPSA adds around 1500 lines of OCaml code.

5.2 Implementation of the Abstract Domain

The implementation is a bit more involved than the description made in the previous section, although the differences are not of a semantical order but due to implementation and performance concerns. The domain is still a cartesian product of three finite maps, but elements of d_1 do not point to a tuple in $TypeId \times Aliasing$: if a variable $v \in \mathbf{Id}$ is aliased with a group of variables $v_1, \ldots, v_n \in \mathbf{Id}$, the variable with the least identifier (called v^*) is mapped to the type identifier, while all the other variables are mapped to v^*. Otherwise, v is mapped to the type identifier. The structure for aliases is currently similar to a union-find but not exactly one. This may be improved in the future if this structure reduces performance; we would then use a union-find structure with fast deletion [7].

To merge the partitions d_1 and d'_1 into r_1, we proceed as follow: we create a mapping $H : TypeId \times TypeId \to TypeId$. Given i, a type identifier for a partition in d_1, and i' being the same for d'_1, $H(i, i')$ represents the type identifier for the partition in r_1, being the intersection of the partitions i and i'. H is updated during the construction of the partitions, meaning that if $H(i, i')$ is needed but not bound, we create a fresh type identifier n and now use $H[(i, i') \mapsto n]$.

6 Examples and Experimental Evaluation

A preliminary experimental comparison with other static type analyzers is showed in Table 13. We comment on the results of the analysis on two examples, and delay the comparison to the next section.

```
1 class A: pass
2
3 x = A()
4 if *: y = x
5 else: y = A()
6 y.a = 2
7 z = x.a
```

Figure 11: class_attr.py

```
1 if *:
2  x = 1
3  l = [2, x]
4 else:
5  x = "a"
6  l = [x, "b"]
```

Figure 12: poly_lists.py

In this first example (Figure 11), we have a program declaring a class A, and assigning to variable x an instance of A. Then, a non-deterministic choice is done, and y is either aliased to x or assigned to a new instance of A. Afterwards, attribute a is added to the object y points at, and the value of attribute a is the integer 2. Finally, the value of $x.a$ is assigned to z. If y and x are aliased the last assignment $z = x.a$ is valid, as the attribute a has been created at line 6. If y and x are not aliased, then the last assignment will raise an `AttributeError`. When analyzing this file in MOPSA, the `AttributeError` is found. In fact, at the end of line 6, we know that: x has type $Instance[A, \emptyset, [a : Instance[int]]]$, meaning that x is an instance of A that *may* have attribute a (being an integer), and we know that y has type $Instance[A, [a : Instance[int]], \emptyset]$, meaning that y is an instance of A that *must* have attribute a (also an integer). Thus, using these type information, the analyzer is able to conclude that an `AttributeError` may be raised.

In the second example (Figure 12), a non-deterministic choice is made, and variable x is either an integer or a string, while variable l is either a list of integers or a string. At the end of the analysis, MOPSA finds that x has type $\alpha \in \{ Instance[str], Instance[int] \}$, while l has type $List[\alpha]$. We can see that variables x and l have been precisely analyzed and related using bounded parametric polymorphism. This example can be expanded: if other non-deterministic choices are made to create new possible types (for example $x = [1]$, $l = [[2, 3], x]$), then the range α would be the only thing to change (into $\{ Instance[str], Instance[int], List[Instance[int]] \}$).

7 Related Work

7.1 Typing and Abstract Interpretation

As mentioned in Section 2, [10] shows that some kinds of typing for the lambda-calculus can be seen as abstractions of a concrete denotational semantics of lambda-calculus. In that work, some lambda-terms can be untypable, as they will result in errors. In this work, we *collect* types of programs rather than discard untypable programs, due to Python's capacity to catch errors.

Gradual typing [20] mixes together explicit static typing and dynamic typing: some variables may be statically typed, and others may be typed by the unknown type (written ?), delaying their type checking at runtime (the ? type can be implicitly cast to and from any other type during static type checking). In particular, gradual typing is useful to type programs incrementally. The safety of gradual typing ensures that if a term has a given type,

it either reduces to a value of the same type, or a casting error between types has been raised. Using abstract interpretation, Garcia et al. explore a formulation of gradual typing where gradual types such as ? concretize into sets of static types [14]. The approach of [14] ensures type safety by construction. Compared to gradual typing, our approach is purely static. In the worst case, the gradual type ? will correspond to the $\top$ type of our analysis (and $\top$ may lead to a high number of false alarms). Contrary to the safety result of gradual typing, if our analysis claims that a program does not contain any type error, no cast error will happen at runtime.

7.2 Static Analysis of Other Dynamic Languages

A few static analyses for the JavaScript language have been designed. Contrary to Python, JavaScript is defined by a standard, and its semantics have been formalized in Coq [9].

In JavaScript, attributes are always accessed dynamically, so a few different abstract domains for strings where designed and evaluated in [6].

[17] defines a value analysis by abstract interpretation for JavaScript programs. [15] proposes a hybrid type inference engine, performing both a static analysis and runtime checks. In [18], Logozzo and Venter use a static type analysis to determine if some numerical variables may be cast from 64-bit floats to 32-bit integers, and optimize the JavaScript code if that is the case, yielding significantly lower running times.

7.3 General Static Analysis of Python

In previous work, Fromherz et al. [13] defined a concrete semantics of Python and a value analysis of Python programs by abstract interpretation. This analysis is relational over numerical variables. As such, the analysis developed during this internship is not an abstraction of the value analysis: the analyses are incomparable. The value analysis is more precise in some cases as it keeps tracks of values, and the type analysis can keep track of variable equalities in ways that the value analysis cannot. For example, the value analysis cannot express that variables x and y have the same type, this type being either $Instance[C]$ or $Instance[D]$. The value analysis is also more costly than the type analysis, and the latter should scale more easily. A future work after this internship is to combine these two analyses, so they can benefit each other.

7.4 Typing of Python Programs

Python Ecosystem Python developers have recently been working on adding type annotations to Python programs. Those type hints, defined in the Python Enhancement Proposal 484 [1], are purely optional and are just used for documentation and type-checking purposes. There are never used by the interpreter. These type hints are mostly an abstraction of the types used by the analysis we proposed, so we could run our analyzer and then annotate the input program with the type hints our analysis found.

Mypy [3] is a static type checker for Python programs using gradual types (without performing the dynamic type checking at runtime). As such, it does not infer types but statically checks that the type hints written in a program are not conflicting. Mypy imposes some restrictions on the input programs: for example, the type of variables cannot change during the execution. Additionally, if a variable is declared in the first branch of an if statement, the type hint should not contradict the use of the same variable in the else branch.

Typeshed [5] provides static types for a part of Python's standard library. It may be useful in the near future to automatically use those type hints to support Python's standard library, although some simplifying assumptions that are made should not be used in our analysis (for example, Python's arbitrary large `int`egers are assumed to be subtype of `float` in typeshed).

Static Analyzers There are three others static analyzers of Python programs: Typpete [16], Pytype [4] and a tool from Fritz and Hage [12]. Typpete encodes type inference of Python programs as a SMT problem and lets Z3 solve it. It supports type hints as defined in the PEP 484, so the analysis may be manually guided, contrary to ours. Pytype is a tool developed by engineers at Google, but there is no reference on how it works. Fritz and Hage use a dataflow analysis to collect types for Python. Both Typpete and Pytype are written in Python, while

Program	Fritz and Hage	Pytype	Typpete	MOPSA - types
class_attr_ok	✓	✗	✳	✓
🐇 class_pre_store	✓	✓	✓	✓
🐇 default_args_class	✓	✓	✓	✓
🐇 except_clause	✗	✎	✓	✓
fspath	✗	✗	✎	✓
🐇 magic	✓	✓	✓	✳
polyfib	✳	✗	✳	✳
poly_lists	✎	✓	✎	✓
🐇 vehicle	✓	✓	✓	✳
widening	✓	✗	✎	✓

Table 13: Comparison of type inference tools for Python

the tool from Fritz and Hage is written in Haskell. In all three cases, the static analyzers proceed by inlining when encountering a function.

We compare the behaviour of these analyzers on a few chosen examples in Table 13. Five examples are taken from Typpete's tests (those are prefixed with 🐇 in the Program column), and five were handwritten to compare the precision of the four analyzers. ✓ means that the analyzer is sound on this example, ✎ means that the analyzer is sound, but too unprecise (for example, an integer is declared as being an object), ✳ means that the analysis is sound, but a false alarm is generated and ✗ means that the analyzer is unsound on this example. False alarms may be due to the analyzer not supporting a feature. For example, Typpete assumes that all attributes are declared statically in the class declaration, and creates a false alarm while analyzing `class_attr_ok`. The files mentioned in Table 13 are given in Appendix D.

As we can see, the tool from Fritz and Hage and Pytype are unsound on a few programs. Typpete appears to be sound on every example, but it is sometimes unprecise. We can notice that none of the analyzers is capable of analyzing program `polyfib` correctly (this is due to two facts: the + operator is highly polymorphic and difficult to analyze precisely, and the program calls a recursive function). The false alarms created by our analysis are mainly due to statements that are not supported yet (for example, recursion, iterations on lists, and metaclasses).

8 Conclusion

We have designed a static type analysis based on abstract interpretation. The abstract domain underlying the analysis is relational, and supports bounded parametric polymorphism. A concretization function explicitly defines what the elements of the abstract domain represent in concrete terms. This type analysis has been partially implemented in the MOPSA static analyzer. Preliminary experimental evaluation shows that this analysis is at least as precise as the state of the art. Future work includes finishing the implementation, supporting a wide range of Python's standard library, and combining this analysis with the value analysis of Fromherz et al.

References

[1] Python Enhancement Proposal 484, about Type Hints. `https://www.python.org/dev/peps/pep-0484/`. Accessed: 2018-07-23.

[2] Implementation of os.fspath. `https://github.com/python/cpython/blob/374c6e178a7599aae46c857b17c6c8bc19dfe4c2/Lib/os.py#L1031`. Accessed: 2018-07-10.

[3] Mypy. `http://mypy-lang.org/`. Accessed: 2018-07-22.

[4] Pytype. `https://github.com/google/pytype`. Accessed: 2018-07-17.

[5] Typeshed. `https://github.com/python/typeshed/`. Accessed: 2018-07-22.

[6] 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 *Lecture Notes in Computer Science*, pages 41–57, 2017. doi:10.1007/978-3-662-54577-5_3.

[7] Amir M. Ben-Amram and Simon Yoffe. A simple and efficient Union-Find-Delete algorithm. *Theor. Comput. Sci.*, 412(4-5):487–492, 2011. doi:10.1016/j.tcs.2010.11.005.

[8] 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. doi:10.1561/2500000002.

[9] 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. doi:10.1145/2535838.2535876.

[10] Patrick Cousot. Types as Abstract Interpretations. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 316–331. ACM Press, 1997. doi:10.1145/263699.263744.

[11] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.

[12] Levin Fritz and Jurriaan Hage. Cost versus precision for approximate typing for Python. In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2017, Paris, France, January 18-20, 2017*, pages 89–98. ACM, 2017. doi:10.1145/3018882.3018888.

[13] Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. Static Value Analysis of Python Programs by Abstract Interpretation. In *NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*, volume 10811 of *Lecture Notes in Computer Science*, pages 185–202. Springer, 2018. doi:10.1007/978-3-319-77935-5_14.

[14] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 429–442. ACM, 2016. doi:10.1145/2837614.2837670.

[15] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *PLDI*, pages 239–250. ACM, 2012. doi:10.1145/2254064.2254094.

[16] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. MaxSMT-Based Type Inference for Python 3. In *CAV (2)*, volume 10982 of *Lecture Notes in Computer Science*, pages 12–19. Springer, 2018. doi:10.1007/978-3-319-96142-2_2.

[17] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *SAS*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2009. doi:10.1145/2535838.2535876.

[18] Francesco Logozzo and Herman Venter. RATA: Rapid Atomic Type Analysis by Abstract Interpretation - Application to JavaScript Optimization. In *CC*, volume 6011 of *Lecture Notes in Computer Science*, pages 66–83. Springer, 2010. doi:10.1007/978-3-642-11970-5_5.

[19] 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. doi:10.1145/2254064.2254092.

[20] Jeremy G. Siek and Walid Taha. Gradual Typing for Objects. In *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2007. doi:10.1007/978-3-540-73589-2_2.

[21] Fausto Spoto. The Julia Static Analyzer for Java. In *SAS*, volume 9837 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2016. doi:10.1007/978-3-662-53413-7_3.

A Missing Proofs

A.1 Soundness of Join over Polymorphic Types

Proof. We want to prove that $\cup^v$ is an overapproximation of $\cup$, i.e:

$$\forall t, t', d_3, d_3', \gamma_p(t, d_3) \cup \gamma_p(t, d_3') \sqsubseteq_p \gamma_p((t, d_3) \cup^v (t', d_3'))$$

We prove only some cases:

- $(\bot, \boldsymbol{d_3}) \cup^v (\boldsymbol{u}, \boldsymbol{d_3'})$:

$$\gamma_p(\bot, d_3 \cup^v u, d_3') = \gamma_p(u, d_3'|_{Vars(u)}) = \gamma_p(u, d_3') = \gamma_p(u, d_3') \cup \gamma_p(\bot, d_3)$$

- $(\boldsymbol{Class\ cl}, \boldsymbol{d_3}) \cup^v (\boldsymbol{Class\ cl'}, \boldsymbol{d_3'})$: Assuming $cl \cup^c cl' \neq object$, we want to show that:

$$\gamma_p(Class\ cl, d_3) \cup \gamma_p(Class\ cl', d_3') \subseteq \gamma_p(Class\ (cl \cup^c cl'), \emptyset)$$

  Let $d \in \{ cl, cl' \}$, we just need to show that $\gamma_v(Class\ d) \subseteq \gamma_v(Class\ (cl \cup^c cl'))$. Given the concretization of types, we need to show that $d \subseteq^c cl \cup^c cl'$, which holds by definition of $\cup^c$.

- **Fallback case:** By definition of γ_p, we get:

$$\begin{aligned}\gamma_p((u, d_3) \cup^v (v, d_3')) &= \gamma_p(\alpha, \alpha \mapsto \gamma_p(u, d_3) \cup \gamma_p(v, d_3')) \\ &= \gamma_p(u, d_3) \cup \gamma_p(v, d_3')\end{aligned}$$

□

A.2 Soundness of Join

We now prove the soundness of the join operator on the whole abstract domain:

Proof. Let $(d_1, d_2, d_3), (d_1', d_2', d_3')$ be two abstract domains. In addition to assuming the invariant in Eq. (7) holds, we assume that $d_2 = d_2'$ and $d_3 = d_3'$ (this can be easily done by transposing type identifiers and type variables (and substituting accordingly), as both $TypeId$ and V_Σ are infinite sets). Let (r_1, d_2, d_3) be the result of the join between (d_1, d_2, d_3) and (d_1', d_2', d_3'). Let ϕ be a bijection between initial type identifiers and resulting type identifiers: $\forall i \in \mathbf{Id}, \phi(d_1(i), d_1'(i)) = r_1(i)$. Without loss of generality, we show that $\gamma(d_1, d_2, d_3) \subseteq \gamma(r_1, d_2, d_3)$. Let $P \in \gamma_{part}(d_2, d_3)$, and $(e, h) \in \mathcal{E} \times \mathcal{H}$ such that:

$$\forall i \in \mathbf{Id}, d_1(i) = (tid, \_) \implies (e(i), h) \in \gamma_v(P(tid)) \tag{8}$$

$$\forall i, j \in \mathbf{Id}, d_1(i) = d_1(j) \wedge \mathrm{snd}\, d_1(i) \neq \bot \implies e(i) = e(j) \tag{9}$$

We show that $(e, h) \in \gamma(r_1, d_2, d_3)$:

1. Let $i \in \mathbf{Id}$. Let $r_1(i) = (rid, \_)$. Let $d_1(i) = (tid, \_)$. Using Eq. (8), we get: $(e(i), h) \in \gamma_v(P(tid))$. We want to show that $(e(i), h) \in \gamma_v(P(rid))$. Let $(\tau_i \in d_3(\alpha_i))_{\alpha_i \in V_\Sigma}$, unfolding the definition of P yields: $P = \lambda t.d_2(t)[\alpha_i \mapsto \tau_i]$. Thus, $P(rid) = d_2(rid)[\alpha_i \mapsto \tau_i]$. As $d_2(rid), d_3 \supseteq (d_2(tid), d_3) \cup^v (d_2(d_1'(i)), d_3)$, we get that $P(rid) \supseteq P(tid)$, so that $(e(i), h) \in \gamma_v(P(rid))$ (as γ_v is monotone).

2. Let $i, j \in \mathbf{Id}$, such that $r_1(i) = r_1(j)$ and $\mathrm{snd}\, r_1(i) \neq \bot$. Let a, a', b, b' such that $r_1(i) = \phi(a, b)$, $r_1(j) = \phi(a', b')$. As $r_1(i) = r_1(j)$, we get $\phi(a, b) = \phi(a', b')$, and by injectivity of ϕ we can conclude that $a = a'$, so that $d_1(i) = d_1(j)$. Using Eq. (9), we get that $e(i) = e(j)$.

□

B Concrete Semantics of Python

As mentioned in Section 3, the concrete semantics of Python described here is a slight evolution from the semantics proposed by Fromherz et al. in [13]. *This part is a copy of [13]'s annex, containing minor updates due to the changes on the environment, and some small update on the semantics (especially, on the semantics of attribute accesses).* The changes are described in the first paragraph of Section 3.

B.1 Domain

$$\mathcal{E} \stackrel{\text{def}}{=} \mathbf{Id} \rightharpoonup \mathbf{Addr} \cup \mathtt{Undef} \qquad \mathbf{Id} \subseteq \mathit{string}$$
$$\mathcal{H} \stackrel{\text{def}}{=} \mathbf{Addr} \rightharpoonup \mathbf{Obj} \times \mathbf{Val} \qquad \mathbf{Obj} \stackrel{\text{def}}{=} \mathit{string} \rightharpoonup \mathbf{Addr}$$
$$\mathcal{F} \stackrel{\text{def}}{=} \{\, \mathit{cur}, \mathit{ret}, \mathit{brk}, \mathit{cont}, \mathit{exn} \,\} \qquad \mathbf{Val} \stackrel{\text{def}}{=} \mathbb{Z} \cup \mathit{string} \cup \{\mathtt{True}, \mathtt{False}, \mathtt{None}, \mathtt{NotImpl}\}$$

Figure 14: Environment of Python programs

The memory environment consists in two parts: the environment $\mathcal{E}$ and the heap $\mathcal{H}$, formally defined in Figure 14. The environment $\mathcal{E}$ is a finite map from variable identifiers **Id** to addresses **Addr**. Due to the scope of variables in Python, variable identifiers may also be locally undefined `Undef`. The heap $\mathcal{H}$ maps addresses to tuples of objects **Obj** and primitive values **Val**. Although everything is object in Python, we need to keep track of primitive values and store them in the heap. Primitive values are either integers, or strings, or booleans, the `None` value, or the `NotImplemented` value. Objects are finite maps from strings (corresponding to attributes and methods) to addresses. To keep track of the complex, non-local control flow of Python programs, while defining semantics by induction on the syntax, we use continuations: in addition to keeping the current state of the analysis, states found before a jump point in the control flow will be kept for later use (for example, the state when an exception is `raise`d will be kept for analysis of later `except` statements). To store those continuations in the program states, we label states using *flow tokens* (elements of $\mathcal{F}$), so the states we consider are elements of $\mathcal{P}(\mathcal{F} \times \mathcal{E} \times H)$ (see the semantics of raise and try/except in Figure 24 and Figure 25) Flow token *cur* represents the current flow on which most instructions operate. *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.

We denote by $\mathbb{E}[\![\, e \,]\!]$ the semantics of expression e. This semantics has the following signature: $\mathbb{E}[\![\, e \,]\!] : (\mathcal{F} \times \mathcal{E} \times \mathcal{H}) \to (\mathcal{F} \times \mathcal{E} \times \mathcal{H} \times \mathbf{Addr})$, so $\mathbb{E}[\![\, e \,]\!]$ returns the address where e has been allocated. The semantics of statements is written $\mathbb{S}[\![\, s \,]\!]$ and has for signature $\mathcal{F} \times \mathcal{E} \times \mathcal{H} \to \mathcal{F} \times \mathcal{E} \times \mathcal{H}$. Note that in the following, both semantics may be implicitly lifted to the powerset.

B.2 Constants

The semantics of constants (Figure 15) returns the address where the constant has been allocated for the current flow, and the address of `None` for all other flows. We assume that booleans, `None` and `NotImpl` are statically

allocated, while integers and strings are not (in CPython, things are a bit more complex as integers between -127 and 128 are allocated statically). Allocation of a new address is performed using fa, which returns a new, yet unused address, i.e., in $\mathbf{Addr} \setminus dom(\sigma)$ (as $\mathbf{Addr}$ is infinite).

$$
\begin{array}{ll}
\mathbb{E}[\![\, \texttt{True} \,]\!]\,(f,\epsilon,\sigma) & \stackrel{\text{def}}{=} \text{if } f \neq cur \text{ then } (f,\epsilon,\sigma,addr_{\texttt{None}}) \text{ else } (f,\epsilon,\sigma,addr_{\texttt{True}}) \\
\mathbb{E}[\![\, \texttt{False} \,]\!]\,(f,\epsilon,\sigma) & \stackrel{\text{def}}{=} \text{if } f \neq cur \text{ then } (f,\epsilon,\sigma,addr_{\texttt{None}}) \text{ else } (f,\epsilon,\sigma,addr_{\texttt{False}}) \\
\mathbb{E}[\![\, i \in \mathbb{Z} \,]\!]\,(f,\epsilon,\sigma) & \stackrel{\text{def}}{=} \text{if } f \neq cur \text{ then } (f,\epsilon,\sigma,addr_{\texttt{None}}) \text{ else } (f,\epsilon,\sigma[fa \mapsto (int,i)],fa) \\
\mathbb{E}[\![\, s \in string \,]\!]\,(f,\epsilon,\sigma) & \stackrel{\text{def}}{=} \text{if } f \neq cur \text{ then } (f,\epsilon,\sigma,addr_{\texttt{None}}) \text{ else } (f,\epsilon,\sigma[fa \mapsto (str,s)],fa) \\
\mathbb{E}[\![\, \texttt{None} \,]\!]\,(f,\epsilon,\sigma) & \stackrel{\text{def}}{=} \text{if } f \neq cur \text{ then } (f,\epsilon,\sigma,addr_{\texttt{None}}) \text{ else } (f,\epsilon,\sigma,addr_{\texttt{None}}) \\
\mathbb{E}[\![\, \texttt{NotImpl} \,]\!]\,(f,\epsilon,\sigma) & \stackrel{\text{def}}{=} \text{if } f \neq cur \text{ then } (f,\epsilon,\sigma,addr_{\texttt{None}}) \text{ else } (f,\epsilon,\sigma,addr_{\texttt{NotImpl}})
\end{array}
$$

Figure 15: Concrete semantics of constants

B.3 Expressions

In the rest of the definition of the concrete semantics, we use the following notation: "letif $(f,\epsilon,\sigma,a) = \ldots$ in ", which unfolds into "let $(f,\epsilon,\sigma,a) = \ldots$ in if $f \neq cur$ then (f,ϵ,σ,a) else ".

Figure 16 presents the semantics of tuples. To evaluate a tuple, we evaluate each element from left to right. If an evaluation does not return the current flow (for example, if an exception has been raised), the evaluation is stopped. If all elements have been evaluated successfully, we allocate a new tuple on the heap.

$$
\begin{array}{l}
\mathbb{E}[\![\, (e_1,\ldots,e_n) \,]\!]\,(f,\epsilon,\sigma) \stackrel{\text{def}}{=} \\
\quad \text{if } f \neq cur \text{ then } (f,\epsilon,\sigma,addr_{\texttt{None}}) \text{ else} \\
\quad \text{letif } (f_1,\epsilon_1,\sigma_1,a_1) = \mathbb{E}[\![\, e_1 \,]\!]\,(f,\epsilon,\sigma) \text{ in} \\
\quad \ldots \\
\quad \text{letif } (f_n,\epsilon_n,\sigma_n,a_n) = \mathbb{E}[\![\, e_n \,]\!]\,(f_{n-1},\epsilon_{n-1},\sigma_{n-1}) \text{ in} \\
\quad \text{let } \sigma' = \sigma_n[fa] \leftarrow (Tuple(v_1,\ldots,v_n),\emptyset) \text{ in} \\
\quad (f_n,\epsilon_n,\sigma',fa)
\end{array}
$$

Figure 16: Concrete semantics of tuples.

Figure 17 presents the access to a variable id. If the key id is not present in the map ϵ, a `NameError` exception is constructed and returned, which corresponds to accessing a non-existent variable. If it maps to `Undef`, then an `UnboundLocalError` exception is returned instead, which denotes access to a local variable before it has been initialized.

$$\mathbb{E}[\![\, id \,]\!]\,(f,\epsilon,\sigma) \stackrel{\text{def}}{=}$$
$$\quad \text{if } f \neq cur \text{ then } (f,\epsilon,\sigma,addr_{\texttt{None}}) \text{ else}$$
$$\quad \text{if } id \notin dom\,\epsilon \text{ then } \texttt{NameError}(f,\epsilon,\sigma) \text{ else}$$
$$\quad \text{if } \epsilon(id) = \texttt{Undef} \text{ then } \texttt{UnboundLocalError}(f,\epsilon,\sigma) \text{ else}$$
$$\quad (f,\epsilon,\sigma,\epsilon(id))$$

$$\texttt{NameError}(f,\epsilon,\sigma) \stackrel{\text{def}}{=}$$
$$\quad \text{let } (f_1,\epsilon_1,\sigma_1,a_1) = \mathbb{E}[\![\, \texttt{NameError}() \,]\!]\,(f,\epsilon,\sigma) \text{ in}$$
$$\quad (exn,\epsilon_1[exn\_var \mapsto a_1],\sigma_1,addr_{\texttt{None}})$$
(and similarly for `UnboundLocalError` and `TypeError`)

Figure 17: Concrete semantics of identifier accesses.

An object has several special attributes: a *class* attribute that points to its class, which is also an object, as well as a *mro* attribute — for *method resolution order* — pointing to a list of classes it inherits from. The *mro* field is used to look up attributes missing in the class — such as inherited methods. We will denote as *contains_field* the test of whether an object contains a field, without searching in its class and superclasses, and as *has_field* the test considering the class and superclasses as well. We also recall that lists, functions, generators, classes, and methods are objects; we denote their sets, respectively, as $\mathbf{List}, \mathbf{Fun}, \mathbf{Gen}, \mathbf{Class}, \mathbf{Method} \subseteq \mathbf{Obj}$. In the following, we denote as $\mathbf{Class}?\ v$, $\mathbf{Fun}?\ v$, etc., an operation that tests whether an address $v \in \mathbf{Addr}$ corresponds to a class, a function, etc.

Figure 18 presents the semantics of accessing an object attribute. We first evaluate the element we are trying to access. If the evaluation is successful, we look for *__getattribute__* in the MRO, and try to evaluate it (we know that at least the `object` class has a *__getattribute__* field). If that *__getattribute__* fails with an `AttributeError`, we search for a *__getattr__* method in the MRO. If that method does exist we call it; otherwise we just raise an `AttributeError`. The description of object's *__getattribute__* is also given. In that case, we first evaluate the object, and then the attribute. We check that `attr` is a string, called s. We search for s in the MRO. If the result is a data descriptor (i.e, it has both a *__get__* and a *__set__*), we return that. Otherwise, we search for s in the instance dictionary, and return otherwise the result found during the MRO search.

$$
\begin{array}{l}
\mathbb{E}[\![\, e.s \,]\!]\,(f,\epsilon,\sigma) \stackrel{\text{def}}{=} \\
\quad \text{if } f \neq \mathit{cur} \text{ then } (f,\epsilon,\sigma,\texttt{None}) \text{ else} \\
\quad \text{letif } (f_1,\epsilon_1,\sigma_1,v_1) = \mathbb{E}[\![\, e \,]\!]\,(f,\epsilon,\sigma) \text{ in} \\
\quad \text{let } c_i = \mathit{mrosearch}(v_1, \_\_\mathit{getattribute}\_\_) \text{ in} \\
\quad \text{let } (f_2,\epsilon_2,\sigma_2,v_2) = \mathbb{E}[\![\, c_i.\_\_\mathit{getattribute}\_\_(v_1,s) \,]\!] \text{ in} \\
\quad \text{if } f_2 = \mathit{exn} \wedge \epsilon_2[\mathit{exn\_var}] = \texttt{AttributeError} \text{ then} \\
\qquad \text{let } d_i = \mathit{mrosearch}(v_1, \_\_\mathit{getattr}\_\_) \text{ in} \\
\qquad \text{if } \textbf{Class}?d_i \text{ then} \\
\qquad\quad \text{let } (f_3,\epsilon_3,\sigma_3,v_3) = \mathbb{E}[\![\, d_i.\_\_\mathit{getattr}\_\_(v_1,s) \,]\!] \text{ in} \\
\qquad\quad \text{if } f_3 = \mathit{exn} \wedge \epsilon_3[\mathit{exn\_var}] = \texttt{AttributeError} \text{ then } \texttt{AttributeError}(f_3,\epsilon_3,\sigma_3) \\
\qquad\quad \text{else } f_3,\epsilon_3,\sigma_3,v_3 \\
\qquad \text{else } \texttt{AttributeError}(f_2,\epsilon_2,\sigma_2) \\
\quad \text{else } f_2,\epsilon_2,\sigma_2,v_2
\end{array}
$$

$$
\begin{array}{l}
\mathbb{E}[\![\, \mathit{object}.\_\_\mathit{getattribute}\_\_(\mathit{obj},\mathit{attr}) \,]\!]\,(f,\epsilon,\sigma) \stackrel{\text{def}}{=} \\
\quad \text{if } f \neq \mathit{cur} \text{ then } (f,\epsilon,\sigma,\texttt{None}) \text{ else} \\
\quad \text{letif } (f_1,\epsilon_1,\sigma_1,v_1) = \mathbb{E}[\![\, \mathit{obj} \,]\!]\,(f,\epsilon,\sigma) \text{ in} \\
\quad \text{letif } (f_2,\epsilon_2,\sigma_2,s) = \mathbb{E}[\![\, \mathit{attr} \,]\!]\,(f_1,\epsilon_1,\sigma_1) \text{ in} \\
\quad \text{if } \textbf{str}?s \text{ then} \\
\qquad \text{let } c_i = \mathit{mrosearch}(v_1,s) \text{ in} \\
\qquad \text{if } \textbf{Class}?c_i \text{ then} \\
\qquad\quad \text{if } \mathit{contains\_field}(c_i.s, \_\_\mathit{get}\_\_) \wedge \mathit{contains\_field}(c_i.s, \_\_\mathit{set}\_\_) \text{ then} \\
\qquad\qquad \mathbb{E}[\![\, c_i.s.\_\_\mathit{get}\_\_() \,]\!]\,(f_2,\epsilon_2,\sigma_2) \\
\qquad\quad \text{else} \\
\qquad\qquad \text{let } (f_3,\epsilon_3,\sigma_3,v_3) = \mathbb{E}[\![\, v_1.\_\_\mathit{dict}\_\_[s] \,]\!] \text{ in} \\
\qquad\qquad \text{if } f_3 = \mathit{exn} \wedge \epsilon_3[\mathit{exn\_var}] = \texttt{KeyError} \text{ then } f_3,\epsilon_3,\sigma_3,c_i.s \\
\qquad\qquad \text{else } f_3,\epsilon_3,\sigma_3,v_3 \\
\qquad \text{else} \\
\qquad\quad \text{let } (f_3,\epsilon_3,\sigma_3,v_3) = \mathbb{E}[\![\, v_1.\_\_\mathit{dict}\_\_[s] \,]\!] \text{ in} \\
\qquad\quad \text{if } f_3 = \mathit{exn} \wedge \epsilon_3[\mathit{exn\_var}] = \texttt{KeyError} \text{ then } \texttt{AttributeError}(f_3,\epsilon_3,\sigma_3) \\
\qquad\quad \text{else } f_3,\epsilon_3,\sigma_3,v_3 \\
\quad \text{else } \mathit{TypeError}(f_2,\epsilon_2,\sigma_2)
\end{array}
$$

Figure 18: Concrete semantics of attribute access.

Figure 19 presents the semantics of addition and of negation. An addition is implemented by the special method $\_\_add\_\_$. When evaluating $e_1 + e_2$, if e_1 does not support the addition, that is, it does not contain a method $\_\_add\_\_$ or the method returns `NotImpl`, and the operands are of different types, we call the reflected method $\_\_radd\_\_$. If this method is not supported either, the addition is not supported for these objects and we return an exception. Other binary arithmetic operators are similar, replacing respectively $\_\_add\_\_$ and $\_\_radd\_\_$ with

the corresponding special methods. The negation is slightly simpler, as it only checks for the $\_\_neg\_\_$ special method. The evaluation of other unary operators is similar, replacing $\_\_neg\_\_$ with the corresponding special method.

$$
\begin{array}{l}
\mathbb{E}[\![\, e_1 + e_2 \,]\!]\,(f,\epsilon,\sigma) \stackrel{\text{def}}{=} \\
\quad \text{if } f \neq \mathit{cur} \text{ then } (f,\epsilon,\sigma,\mathit{addr}_{\texttt{None}}) \text{ else} \\
\quad \text{letif } (f_1,\epsilon_1,\sigma_1,a_1) = \mathbb{E}[\![\, e_1 \,]\!]\,(f,\epsilon,\sigma) \text{ in} \\
\quad \text{letif } (f_2,\epsilon_2,\sigma_2,a_2) = \mathbb{E}[\![\, e_2 \,]\!]\,(f_1,\epsilon_1,\sigma_1) \text{ in} \\
\quad \text{if } \mathit{has\_field}(a_1, \_\_add\_\_, \sigma_2) \text{ then} \\
\qquad \text{letif } (f_3,\epsilon_3,\sigma_3,a_3) = \mathbb{E}[\![\, a_1.\_\_add\_\_(a_2) \,]\!]\,(f_2,\epsilon_2,\sigma_2) \text{ in} \\
\qquad \text{if } \sigma_3(a_3) = (\_, \texttt{NotImpl}) \text{ then} \\
\qquad\quad \text{if } \mathit{has\_field}(a_2, \_\_radd\_\_, \sigma_3) \wedge \mathit{typeof}(a_1) \neq \mathit{typeof}(a_2) \text{ then} \\
\qquad\qquad \text{letif } (f_4,\epsilon_4,\sigma_4,a_4) = \mathbb{E}[\![\, a_2.\_\_radd\_\_(a_1) \,]\!]\,(f_3,\epsilon_3,\sigma_3) \text{ in} \\
\qquad\qquad \text{if } \sigma_4(a_4) = (\_, \texttt{NotImpl}) \text{ then } \texttt{TypeError}(f_4,\epsilon_4,\sigma_4) \\
\qquad\qquad \text{else } (f_4,\epsilon_4,\sigma_4,a_4) \\
\qquad\quad \text{else } \texttt{TypeError}(f_3,\epsilon_3,\sigma_3) \\
\qquad \text{else } f_3,\epsilon_3,\sigma_3,a_3 \\
\quad \text{else if } \mathit{has\_field}(a_2, \_\_radd\_\_, \sigma_2) \wedge \mathit{typeof}\, a_1 \neq \mathit{typeof}\, a_2 \text{ then} \\
\qquad \text{letif } (f_3,\epsilon_3,\sigma_3,a_3) = \mathbb{E}[\![\, a_2.\_\_radd\_\_(a_1) \,]\!]\,(f_2,\epsilon_2,\sigma_2) \text{ in} \\
\qquad \text{if } \sigma_3(a_3) = (\_, \texttt{NotImpl}) \text{ then } \texttt{TypeError}(f_3,\epsilon_3,\sigma_3) \\
\qquad \text{else } (f_3,\epsilon_3,\sigma_3,a_3) \\
\quad \text{else } \texttt{TypeError}(f_2,\epsilon_2,\sigma_2)
\end{array}
$$

$$
\begin{array}{l}
\mathbb{E}[\![\, -e \,]\!]\,(f,\epsilon,\sigma) \stackrel{\text{def}}{=} \\
\quad \text{if } f \neq \mathit{cur} \text{ then } (f,\epsilon,\sigma,\mathit{addr}_{\texttt{None}}) \text{ else} \\
\quad \text{letif } (f_1,\epsilon_1,\sigma_1,a) = \mathbb{E}[\![\, e \,]\!]\,(f,\epsilon,\sigma) \text{ in} \\
\quad \text{if } \mathit{has\_field}(a, \_\_neg\_\_, \sigma_1) \text{ then } \mathbb{E}[\![\, a.\_\_neg\_\_() \,]\!]\,(f_1,\epsilon_1,\sigma_1) \text{ else} \\
\quad \texttt{TypeError}(f_1,\epsilon_1,\sigma_1)
\end{array}
$$

Figure 19: Concrete semantics of addition and negation.

Figure 20 presents the semantics of equality ==. The other comparison operators are similar. The evaluation is similar to the addition: If an operation is not supported, we try the reflected operand ($\_\_eq\_\_$ for $\_\_eq\_\_$, and $\_\_lt\_\_$ for $\_\_gt\_\_$ for instance). If the operands do not have the same type and the right operand's type is a subclass of the left operand's type, we first try the reflected method.

$$
\begin{array}{l}
\mathbb{E}[\![\, e_1 == e_2 \,]\!]\,(f,\epsilon,\sigma) \stackrel{\text{def}}{=} \\
\quad \text{if } f \neq \mathit{cur} \text{ then } (f,\epsilon,\sigma,\mathit{addr}_{\texttt{None}}) \text{ else} \\
\quad \text{letif } (f_1,\epsilon_1,\sigma_1,a_1) = \mathbb{E}[\![\, e_1 \,]\!]\,(f,\epsilon,\sigma) \text{ in} \\
\quad \text{letif } (f_2,\epsilon_2,\sigma_2,a_2) = \mathbb{E}[\![\, e_2 \,]\!]\,(f_1,\epsilon_1,\sigma_1) \text{ in} \\
\quad \text{if } \mathit{typeof}(a_1) \neq \mathit{typeof}(a_2) \wedge \mathit{typeof}(a_2)\ \mathit{subtypeof}\ \mathit{typeof}(a_1) \text{ then} \\
\qquad \text{if } \mathit{has\_field}(a_2, \_\_\mathit{eq}\_\_, \sigma_2) \text{ then} \\
\qquad\quad \text{letif } (f_3,\epsilon_3,\sigma_3,a_3) = \mathbb{E}[\![\, a_2.\_\_\mathit{eq}\_\_(a_1) \,]\!]\,(f_2,\epsilon_2,\sigma_2) \text{ in} \\
\qquad\quad \text{if } \sigma_3(a_3) = (\_, \texttt{NotImpl}) \text{ then} \\
\qquad\qquad \text{if } \mathit{has\_field}(a_1, \_\_\mathit{eq}\_\_, \sigma_3) \text{ then} \\
\qquad\qquad\quad \text{letif } (f_4,\epsilon_4,\sigma_4,a_4) = \mathbb{E}[\![\, a_1.\_\_\mathit{eq}\_\_(a_2) \,]\!]\,(f_3,\epsilon_3,\sigma_3) \text{ in} \\
\qquad\qquad\quad \text{if } \sigma_4(a_4) = (\_, \texttt{NotImpl}) \text{ then } (f_4,\epsilon_4,\sigma_4,\mathit{addr}_{a_1 =_{addr} a_2}) \\
\qquad\qquad\quad \text{else } (f_4,\epsilon_4,\sigma_4,a_4) \\
\qquad\qquad \text{else } (f_3,\epsilon_3,\sigma_3,\mathit{addr}_{a_1 =_{addr} a_2}) \\
\qquad\quad \text{else } f_3,\epsilon_3,\sigma_3,a_3 \\
\qquad \text{else if } \mathit{has\_field}(a_1, \_\_\mathit{eq}\_\_, \sigma_2) \text{ then} \\
\qquad\quad \text{letif } (f_3,\epsilon_3,\sigma_3,a_3) = \mathbb{E}[\![\, a_1.\_\_\mathit{eq}\_\_(a_2) \,]\!]\,(f_2,\epsilon_2,\sigma_2) \text{ in} \\
\qquad\quad \text{if } \sigma_3(a_3) = (\_, \texttt{NotImpl}) \text{ then } (f_3,\epsilon_3,\sigma_3,\mathit{addr}_{a_1 =_{addr} a_2}) \\
\qquad\quad \text{else } (f_3,\epsilon_3,\sigma_3,a_3) \\
\qquad \text{else } (f_2,\epsilon_2,\sigma_2,\mathit{addr}_{a_1 =_{addr} a_2}) \\
\quad \text{else} \\
\qquad \text{if } \mathit{has\_field}(a_1, \_\_\mathit{eq}\_\_, \sigma_2) \text{ then} \\
\qquad\quad \text{letif } (f_3,\epsilon_3,\sigma_3,a_3) = \mathbb{E}[\![\, a_1.\_\_\mathit{eq}\_\_(a_2) \,]\!]\,(f_2,\epsilon_2,\sigma_2) \text{ in} \\
\qquad\quad \text{if } \sigma(a_3) = (\_, \texttt{NotImpl}) \text{ then} \\
\qquad\qquad \text{if } \mathit{has\_field}(a_2, \_\_\mathit{eq}\_\_, \sigma_3) \text{ then} \\
\qquad\qquad\quad \text{letif } (f_4,\epsilon_4,\sigma_4,a_4) = \mathbb{E}[\![\, a_2.\_\_\mathit{eq}\_\_(a_1) \,]\!]\,(f_3,\epsilon_3,\sigma_3) \text{ in} \\
\qquad\qquad\quad \text{if } \sigma_4(a_4) = (\_, \texttt{NotImpl}) \text{ then } (f_4,\epsilon_4,\sigma_4,\mathit{addr}_{a_1 =_{addr} a_2}) \\
\qquad\qquad\quad \text{else } (f_4,\epsilon_4,\sigma_4,a_4) \\
\qquad\qquad \text{else } (f_3,\epsilon_3,\sigma_3,\mathit{addr}_{a_1 =_{addr} a_2}) \\
\qquad\quad \text{else } f_3,\epsilon_3,\sigma_3,a_3 \\
\qquad \text{else if } \mathit{has\_field}(a_2, \_\_\mathit{eq}\_\_, \sigma_2) \text{ then} \\
\qquad\quad \text{letif } (f_3,\epsilon_3,\sigma_3,a_3) = \mathbb{E}[\![\, a_2.\_\_\mathit{eq}\_\_(a_1) \,]\!]\,(f_2,\epsilon_2,\sigma_2) \text{ in} \\
\qquad\quad \text{if } \sigma_3(a_3) = (\_, \texttt{NotImpl}) \text{ then } (f_3,\epsilon_3,\sigma_3,\mathit{addr}_{a_1 =_{addr} a_2}) \\
\qquad\quad \text{else } (f_3,\epsilon_3,\sigma_3,a_3) \\
\qquad \text{else } (f_2,\epsilon_2,\sigma_2,\mathit{addr}_{a_1 =_{addr} a_2})
\end{array}
$$

Figure 20: Concrete semantics of comparisons.

Figure 21 presents a list access, which is done by calling the special method $\_\_\mathit{getitem}\_\_$. We first evaluate the list object, then the index object. If both evaluations are successful, we proceed with the call to the method, if it exists. If not, a `TypeError` is returned since this operation is not supported.

$$
\begin{array}{l}
\mathbb{E}[\![\, e_1[e_2] \,]\!]\,(f,\epsilon,\sigma) \stackrel{\text{def}}{=} \\
\quad \text{if } f \neq \mathit{cur} \text{ then } (f,\epsilon,\sigma,\mathit{addr}_{\texttt{None}}) \text{ else} \\
\quad \text{letif } (f_1,\epsilon_1,\sigma_1,a_1) = \mathbb{E}[\![\, e_1 \,]\!]\,(f,\epsilon,\sigma) \text{ in} \\
\quad \text{letif } (f_2,\epsilon_2,\sigma_2,a_2) = \mathbb{E}[\![\, e_2 \,]\!]\,(f_1,\epsilon_1,\sigma_1) \text{ in} \\
\quad \text{if } \mathit{has\_field}(a_1, \texttt{\_\_}\mathit{getitem}\texttt{\_\_}) \text{ then } \mathbb{E}[\![\, a_1.\texttt{\_\_}\mathit{getitem}\texttt{\_\_}(a_2) \,]\!]\,(f_2,\epsilon_2,\sigma_2) \\
\quad \text{else } \texttt{TypeError}(f_2,\epsilon_2,\sigma_2)
\end{array}
$$

Figure 21: Concrete semantics of list access.

Figure 22 presents the semantics of function calls. We first evaluate the called object, and all arguments passed. If these evaluations are successful, the call can now have different effects. If the called object is a function, or a method, we ensure that the number of arguments corresponds to the expected number. If not, a `TypeError` is raised. If the function called is actually a generator, we create a new generator object on the heap, and set the generator counter to *start* to indicate that the generator has not been called yet. If not, we execute the body of the function, returning `None` if no return value was provided. If the called object is a class, we create a new instance of this class, by calling the special method `__`*new*`__`. If `__`*new*`__` returns an instance of the class, the `__`*init*`__` method of this new instance is invoked. If `__`*init*`__` is invoked and does not return a `None` value, a `TypeError` is raised. Finally, if none of the above cases apply and the called object emulates a callable object through a `__`*call*`__` method, we call this method. If not, a `TypeError` is raised.

$$
\begin{array}{l}
\mathbb{E}[\![\, e(e_1,\ldots,e_n)\,]\!]\,(f,\epsilon,\sigma) \stackrel{\text{def}}{=} \\
\quad \text{if } f \neq \mathit{cur} \text{ then } (f,\epsilon,\sigma,\mathit{addr}_{\texttt{None}}) \text{ else} \\
\quad \text{letif } (f_0,\epsilon_0,\sigma_0,v) = \mathbb{E}[\![\, e\,]\!]\,(f,\epsilon,\sigma) \text{ in} \\
\quad \text{letif } (f_1,\epsilon_1,\sigma_1,v_1) = \mathbb{E}[\![\, e_1\,]\!]\,(f_0,\epsilon_0,\sigma_0) \text{ in} \\
\quad \ldots \\
\quad \text{letif } (f_n,\epsilon_n,\sigma_n,v_n) = \mathbb{E}[\![\, e_n\,]\!]\,(f_{n-1},\epsilon_{n-1},\sigma_{n-1}) \text{ in} \\
\quad \text{if } \mathrm{fst}\,\sigma_n[v] = \mathbf{Fun}((a_1,\ldots,a_m),\mathit{body},\mathit{is\_gen}) \text{ then} \\
\qquad \text{if } m \neq n \text{ then } \texttt{TypeError}(f_n,\epsilon_n,\sigma_n) \text{ else} \\
\qquad \text{let } \epsilon' = \epsilon_n[a_1,\ldots,a_m] \leftarrow [v_1,\ldots,v_n] \text{ in} \\
\qquad \text{if } \mathit{is\_gen} \text{ then } (\mathit{cur},\epsilon',\sigma_n[\mathit{fa}] \leftarrow \\
\qquad\quad (\mathbf{Gen}((\mathit{start},[a_1 \to v_1,\ldots,a_n \to v_n]),\mathit{body}),\emptyset),\mathit{fa}) \\
\qquad \text{else let } (f_e,\epsilon_e,\sigma_e) = \mathbb{S}[\![\, \mathit{body}\,]\!]\,(\mathit{cur},\epsilon',\sigma_n) \text{ in} \\
\qquad \text{if } f_e \neq \mathit{cur},\mathit{ret} \text{ then } (f_e,\epsilon_e,\sigma_e,\mathit{addr}_{\texttt{None}}) \text{ else} \\
\qquad \text{if } \epsilon_e[\mathit{return\_var}] = \texttt{Undef} \text{ then } (\mathit{cur},\epsilon_e,\sigma_e,\mathit{addr}_{\texttt{None}}) \\
\qquad \text{else } (\mathit{cur},\epsilon_e,\sigma_e,\epsilon_e[\mathit{return\_var}]) \\
\quad \text{else if } \mathrm{fst}\,\sigma_n[v] = \mathbf{Method}(\mathit{obj},\mathbf{Fun}((a_1,\ldots,a_m),\mathit{body},\mathit{is\_gen})) \text{ then} \\
\qquad \text{if } m \neq n+1 \text{ then } \texttt{TypeError}(f_n,\epsilon_n,\sigma_n) \text{ else} \\
\qquad \text{let } \epsilon' = \epsilon_n[a_1,\ldots,a_m] \leftarrow [\mathit{obj},v_1,\ldots,v_n] \text{ in} \\
\qquad \text{if } \mathit{is\_gen} \text{ then } (\mathit{cur},\epsilon',\sigma_n[\mathit{fa}] \leftarrow \\
\qquad\quad (\mathbf{Gen}((\mathit{start},[a_1 \to \mathit{obj},\ldots,a_m \to v_n]),\mathit{body}),\emptyset),\mathit{fa}) \\
\qquad \text{else let } (f_e,\epsilon_e,\sigma_e) = \mathbb{S}[\![\, \mathit{body}\,]\!]\,(\mathit{cur},\epsilon',\sigma_n) \text{ in} \\
\qquad \text{if } f_e \neq \mathit{cur},\mathit{ret} \text{ then } (f_e,\epsilon_e,\sigma_e,\mathit{addr}_{\texttt{None}}) \text{ else} \\
\qquad \text{if } \epsilon_e[\mathit{return\_var}] = \texttt{Undef} \text{ then } (\mathit{cur},\epsilon_e,\sigma_e,\mathit{addr}_{\texttt{None}}) \\
\qquad \text{else } (\mathit{cur},\epsilon_e,\sigma_e,\epsilon_e[\mathit{return\_var}]) \\
\quad \text{else if } \mathrm{fst}\,\sigma_n[v] = \mathbf{Class}(c) \text{ then} \\
\qquad \text{let } (f_{new},\epsilon_{new},\sigma_{new},v_{new}) = \mathbb{E}[\![\, c.\_\_new\_\_(c,v_1,\ldots,v_n)\,]\!]\,(f_n,\epsilon_n,\sigma_n) \text{ in} \\
\qquad \text{if } f_{new} \neq \mathit{cur},\mathit{ret} \text{ then } (f_{new},\epsilon_{new},\sigma_{new},v_{new}) \\
\qquad \text{else if } \mathit{typeof}(v_{new}) \neq c \text{ then } (\mathit{cur},\epsilon_{new},\sigma_{new},v_{new}) \\
\qquad \text{else let } (f_{init},\epsilon_{init},\sigma_{init},v_{init}) = \\
\qquad\quad \mathbb{E}[\![\, c.\_\_init\_\_(v_{new},v_1,\ldots,v_n)\,]\!]\,(\mathit{cur},\epsilon_{new},\sigma_{new}) \text{ in} \\
\qquad \text{if } f_{init} \neq \mathit{cur},\mathit{ret} \text{ then } (f_{init},\epsilon_{init},\sigma_{init},v_{init}) \text{ else} \\
\qquad \text{if } v_{init} \neq \mathit{addr}_{\texttt{None}} \text{ then } \mathit{TypeError}(f_{init},\epsilon_{init},\sigma_{init}) \text{ else } (\mathit{cur},\epsilon_{init},\sigma_{init},v_{new}) \\
\quad \text{else if } \mathit{contains\_field}(v,\_\_call\_\_,\sigma_n) \text{ then } \mathbb{E}[\![\, v.\_\_call\_\_(v_1,\ldots,v_n)\,]\!]\,(f_n,\epsilon_n,\sigma_n) \\
\quad \text{else } \texttt{TypeError}(f_n,\epsilon_n,\sigma_n)
\end{array}
$$

Figure 22: Concrete semantics of a function call.

B.4 Statements

The semantics of statements have signature $\mathbb{S}[\![\, s\,]\!] : \mathcal{P}(\mathcal{F}\times\mathcal{E}\times\mathcal{H}) \to \mathcal{P}(\mathcal{F}\times\mathcal{E}\times\mathcal{H})$.

Figure 23 presents the semantics of atomic statements that map a state in the current continuation flow to

another state in the current continuation flow. We denote as $read\_only(v)$ the function that tests whether the address v corresponds to a builtin object that is not modifiable, such as a frozen set, in order to raise an exception in case of assignment.

$$\mathbb{S}[\![\,\texttt{pass}\,]\!]\,S \stackrel{\text{def}}{=} S$$

$$\mathbb{S}[\![\,e\,]\!]\,S \stackrel{\text{def}}{=}$$
$$\{\,(f,\epsilon,\sigma) \mid (f,\epsilon,\sigma,v) \in \mathbb{E}[\![\,e\,]\!]S\,\}$$

$$\mathbb{S}[\![\,id = e\,]\!]\,S \stackrel{\text{def}}{=}$$
$$\text{let } S_e, A_e = \mathbb{E}[\![\,e\,]\!]\,S \text{ in}$$
$$\{\,(f,\epsilon,\sigma) \mid (f,\epsilon,\sigma) \in S_e \wedge f \neq cur\,\} \cup$$
$$\{\,(cur, \epsilon[id] \leftarrow a, \sigma) \mid (cur,\epsilon,\sigma,a) \in (S_e, A_e)\,\}$$

$$\mathbb{S}[\![\,id.s = e\,]\!]\,S \stackrel{\text{def}}{=}$$
$$\text{let } S_e, A_e = \mathbb{E}[\![\,e\,]\!]\,S \text{ in}$$
$$\{\,(f,\epsilon,\sigma) \mid (f,\epsilon,\sigma) \in S_e \wedge f \neq cur\,\} \cup$$
$$\{\,(exn, \epsilon[exn\_var] \leftarrow \texttt{NameError}, \sigma) \mid (cur,\epsilon,\sigma) \in S_e \wedge id \not\in dom\,\epsilon\,\} \cup$$
$$\{\,(exn, \epsilon[exn\_var] \leftarrow \texttt{UnboundLocalError}, \sigma) \mid (cur,\epsilon,\sigma) \in S_e \wedge \epsilon[id] = \texttt{Undef}\,\} \cup$$
$$\{\,(exn, \epsilon[exn\_var] \leftarrow \texttt{AttributeError}, \sigma) \mid (cur,\epsilon,\sigma) \in S_e \wedge read\_only(\epsilon[id]) \wedge$$
$$\epsilon[id] \neq \texttt{Undef}\,\} \cup$$
$$\{\,(f_1,\epsilon_1,\sigma_1) \mid (cur,\epsilon,\sigma) \in S_e \wedge id \notin dom\,\epsilon \wedge \epsilon[id] \neq \texttt{Undef} \wedge \neg read\_only(\epsilon[id]) \wedge$$
$$(f_1,\epsilon_1,\sigma_1,\_) \in \mathbb{E}[\![\,id.\_\_setattr\_\_(s,e)\,]\!](f,\epsilon,\sigma)\,\}$$

$$\mathbb{S}[\![\,object.\_\_setattr\_\_(obj, attr, expr)\,]\!]\,S \stackrel{\text{def}}{=}$$
$$\text{if } f \neq cur \text{ then } (f,\epsilon,\sigma,\texttt{None}) \text{ else}$$
$$\text{letif } (f_1,\epsilon_1,\sigma_1,o) = \mathbb{E}[\![\,obj\,]\!]\,(f,\epsilon,\sigma) \text{ in}$$
$$\text{letif } (f_2,\epsilon_2,\sigma_2,a) = \mathbb{E}[\![\,attr\,]\!]\,(f_1,\epsilon_1,\sigma_1) \text{ in}$$
$$\text{letif } (f_3,\epsilon_3,\sigma_3,e) = \mathbb{E}[\![\,expr\,]\!]\,(f_2,\epsilon_2,\sigma_2) \text{ in}$$
$$\text{if } \textbf{str}?a \text{ then}$$
$$\quad \text{let } c_i = mrosearch(o,a) \text{ in}$$
$$\quad \text{if } \textbf{Class}?c_i \text{ then}$$
$$\quad\quad \text{if } contains\_field(c_i.a, \_\_get\_\_) \wedge contains\_field(c_i.a, \_\_set\_\_) \text{ then}$$
$$\quad\quad\quad \mathbb{E}[\![\,c_i.a.\_\_set\_\_(o,e)\,]\!]$$
$$\quad\quad \text{else}$$
$$\quad\quad\quad \mathbb{E}[\![\,o.\_\_dict\_\_[a] = e\,]\!]$$
$$\quad \text{else}$$
$$\quad\quad \mathbb{E}[\![\,o.\_\_dict\_\_[a] = e\,]\!]$$
$$\text{else } TypeError(f_3,\epsilon_3,\sigma_3)$$

Figure 23: Concrete semantics of atomic statements.

Figure 24 presents the semantics of statements that store the environment from the current flow into another continuation: `return`, `break`, `continue`, but also exception handing (`raise`) and generator exit (`yield`). We denote as $inherits\_from(v_1, v_2)$ the function that tests if v_1 is a subclass of v_2, or if v_1's type is a subclass of v_2. When raising an expression e, we set the special variable exn_var to the value of e to propagate the exception to its handler. If the value raised does not inherit from `BaseException`, a `TypeError` is raised.

$$
\begin{array}{l}
\mathbb{S}[\![\,\texttt{return}\ e\,]\!]\ S \stackrel{\text{def}}{=} \\
\quad \text{let } S_1 = \mathbb{S}[\![\, return\_var \leftarrow e\,]\!]\ S \text{ in} \\
\quad \{\,(f,\epsilon,\sigma) \in S_1 \mid f \neq cur\,\} \cup \\
\quad \{\,(ret,\epsilon,\sigma) \mid (cur,\epsilon,\sigma) \in S_1\,\}
\end{array}
$$

$$
\begin{array}{l}
\mathbb{S}[\![\,\texttt{break}\,]\!]\ S \stackrel{\text{def}}{=} \\
\quad \{\,(f,\epsilon,\sigma) \in S \mid f \neq cur\,\} \cup \\
\quad \{\,(brk,\epsilon,\sigma) \mid (cur,\epsilon,\sigma) \in S\,\}
\end{array}
$$

$$
\begin{array}{l}
\mathbb{S}[\![\,\texttt{continue}\,]\!]\ S \stackrel{\text{def}}{=} \\
\quad \{\,(f,\epsilon,\sigma) \in S \mid f \neq cur\,\} \cup \\
\quad \{\,(cont,\epsilon,\sigma) \mid (cur,\epsilon,\sigma) \in S\,\}
\end{array}
$$

$$
\begin{array}{l}
\mathbb{S}[\![\,\texttt{raise}\ e\,]\!]\ S \stackrel{\text{def}}{=} \\
\quad \text{let } S_1 = \mathbb{S}[\![\, exn\_var = e\,]\!]\ S \text{ in} \\
\quad \{\,(f,\epsilon,\sigma) \in S_1 \mid f \neq cur\,\} \cup \\
\quad \{\,(exn, \epsilon[exn\_var] \leftarrow \texttt{TypeError}, \sigma) \mid (cur,\epsilon,\sigma) \in S_1 \wedge \\
\qquad \neg inherits\_from(\epsilon[exn\_var], \texttt{BaseException})\,\} \cup \\
\quad \{\,(exn,\epsilon,\sigma) \mid (cur,\epsilon,\sigma) \in S_1 \wedge \\
\qquad inherits\_from(\epsilon[exn\_var], \texttt{BaseException})\,\}
\end{array}
$$

$$
\begin{array}{l}
\mathbb{S}[\![\,\texttt{yield}_i\ e\,]\!]\ S \stackrel{\text{def}}{=} \\
\quad \text{let } S_1 = \mathbb{S}[\![\, yield\_var \leftarrow e\,]\!]\ S \text{ in} \\
\quad \{\,(f,\epsilon,\sigma) \in S_1 \mid f \neq cur, next\,\} \cup \\
\quad \{\,(yield(i),\epsilon,\sigma) \mid (cur,\epsilon,\sigma) \in S_1\,\} \cup \\
\quad \{\,(cur,\epsilon,\sigma) \mid (next(i),\epsilon,\sigma) \in S_1\,\}
\end{array}
$$

$$
\mathbb{S}[\![\, s_1; s_2\,]\!] \stackrel{\text{def}}{=} \mathbb{S}[\![\, s_2\,]\!] \circ \mathbb{S}[\![\, s_1\,]\!]
$$

Figure 24: Concrete semantics of flow statements.

Figure 25 gives the definition of compound statements composed of sub-statements: tests, loops, and exception handlers. For exception handling, the try block is first evaluated. If an exception is raised, the first except block is executed. If x_1 is not an exception, i.e., it does not derive from `BaseException`, this is an error. If it does, we check whether the exception raised by the try block derives from x_1. If it does, we catch the exception, and evaluate the corresponding block s_1. If not, we iterate over the other handlers. If the try block was completely

executed without interruption, the else block is normally executed.

$$
\begin{array}{l}
\mathbb{S}[\![\, \texttt{if\_then\_else}(e, s_1, s_2) \,]\!]\, S \stackrel{\text{def}}{=} \\
\quad \mathbb{S}[\![\, s_1 \,]\!]\, (\mathit{filter}(e)(S)) \cup \mathbb{S}[\![\, s_2 \,]\!](\mathit{filter}(\neg e)(S)) \\
\text{where:} \\
\mathit{filter}(e)(S) \stackrel{\text{def}}{=} \cup_{(f,\epsilon,\sigma)\in S}\ \mathit{filter}(e)(f, \epsilon, \sigma) \\
\mathit{filter}(e)(f, \epsilon, \sigma) \stackrel{\text{def}}{=} \\
\quad \text{letif } (f_1, \epsilon_1, \sigma_1, v_1) = \mathbb{E}[\![\, e \,]\!]\, (f, \epsilon, \sigma) \text{ in} \\
\quad \text{letif } (f_2, \epsilon_2, \sigma_2, v_2) = \mathit{is\_true}(f_1, v_1, \epsilon_1, \sigma_1) \text{ in} \\
\quad \text{if } v_2 = \texttt{True} \text{ then } \{(f_2, \epsilon_2, \sigma_2)\} \text{ else } \emptyset
\end{array}
$$

$$
\begin{array}{l}
\mathbb{S}[\![\, \texttt{while}(e, c) \,]\!]\, S \stackrel{\text{def}}{=} \\
\quad \text{let } S_0 = \{\, (f, \epsilon, \sigma) \in S \mid f \notin \{\mathit{brk}, \mathit{cont}\} \,\} \text{ in} \\
\quad \text{let } S_1 = \mathit{filter}(\neg e)(\text{lfp } f) \text{ in} \\
\qquad \{\, (f, \epsilon, \sigma) \in S_1 \mid f \notin \{\mathit{brk}, \mathit{cont}\} \,\} \cup \\
\qquad \{\, (\mathit{cur}, \epsilon, \sigma) \mid (\mathit{brk}, \epsilon, \sigma) \in S_1 \,\} \cup \\
\qquad \{\, (f, \epsilon, \sigma) \in S_0 \mid f \in \{\mathit{brk}, \mathit{cont}\} \,\} \\
\text{where:} f(S) \stackrel{\text{def}}{=} \text{let } S_1 = \mathbb{S}[\![\, c \,]\!]\, (\mathit{filter}(e)(S)) \text{ in } S_1 \cup \{\, (\mathit{cur}, \epsilon, \sigma) \mid (\mathit{cont}, \epsilon, \sigma) \in S_1 \,\}
\end{array}
$$

$$
\begin{array}{l}
\mathbb{S}[\![\, \texttt{try\_except\_else}(s_{try}, [(x_1, s_1), \ldots, (x_n, s_n)], s_{else}) \,]\!]\, S \stackrel{\text{def}}{=} \\
\quad \text{let } S_0 = \mathbb{S}[\![\, s_{try} \,]\!]\, S \text{ in} \\
\quad \text{let } S_{exc1} = \{\, (f, \epsilon, \sigma) \in S_0 \mid f = \mathit{exn} \,\} \text{ in} \\
\quad \text{let } S_{err1} = \{\, (\mathit{exn}, \epsilon, \sigma) \in S_{exc1} \mid \neg \mathit{inherits\_from}(\epsilon[x_1], \texttt{BaseException}) \,\} \text{ in} \\
\quad \text{let } S'_1 = \{\, (\mathit{exn}, \epsilon, \sigma) \in (S_{exc1} \setminus S_{err1}) \mid \mathit{inherits\_from}(\epsilon[\mathit{exn\_var}], \epsilon[x_1]) \,\} \text{ in} \\
\quad \text{let } S_1 = \mathbb{S}[\![\, s_1 \,]\!]\, \{\, (\mathit{cur}, \epsilon, \sigma) \mid (\mathit{exn}, \epsilon, \sigma) \in S'_1 \,\} \text{ in} \\
\quad \text{let } S_{exc2} = S_{exc1} \setminus (S'_1 \cup S_{err_1}) \text{ in} \\
\quad \ldots \\
\quad \text{let } S_{err_n} = \{\, (\mathit{exn}, \epsilon, \sigma) \in S_{exc_n} \mid \neg \mathit{inherits\_from}(\epsilon[x_n], \texttt{BaseException}) \,\} \text{ in} \\
\quad \text{let } S'_n = \{\, (\mathit{exn}, \epsilon, \sigma) \in (S_{exc_n} \setminus S_{err_n}) \mid \mathit{inherits\_from}(\epsilon[\mathit{exn\_var}], \epsilon[x_n]) \,\} \text{ in} \\
\quad \text{let } S_n = \mathbb{S}[\![\, s_n \,]\!]\, \{\, (\mathit{cur}, \epsilon, \sigma) \mid (\mathit{exn}, \epsilon, \sigma) \in S'_n \,\} \text{ in} \\
\quad \text{let } S_{exc_f} = S_{exc_n} \setminus (S_{err_n} \cup S'_n) \text{ in} \\
\quad \text{let } S_{else} = \mathbb{S}[\![\, s_{else} \,]\!]\, (\{\, (f, \epsilon, \sigma) \in S_0 \mid f = \mathit{cur} \,\} \text{ in} \\
\quad S_{else} \cup (\bigcup_{i=1}^{i=n} S_i \cup \{\, (f, \epsilon[\mathit{exn\_var}] \leftarrow \texttt{TypeError}, \sigma \mid (f, \epsilon, \sigma) \in S_{err_i} \,\}) \cup \\
\quad S_{exc_f} \cup \{\, (f, \epsilon, \sigma) \in S_0 \mid f \neq \mathit{cur}, \mathit{exn} \,\}
\end{array}
$$

Figure 25: Concrete semantics of composed statements.

Finally, Figure 26 presents the semantics of declarations, for functions, generators, and classes. The declaration of both a function and a generator consists in the creation of a new **Fun** object on the heap. If we declare a

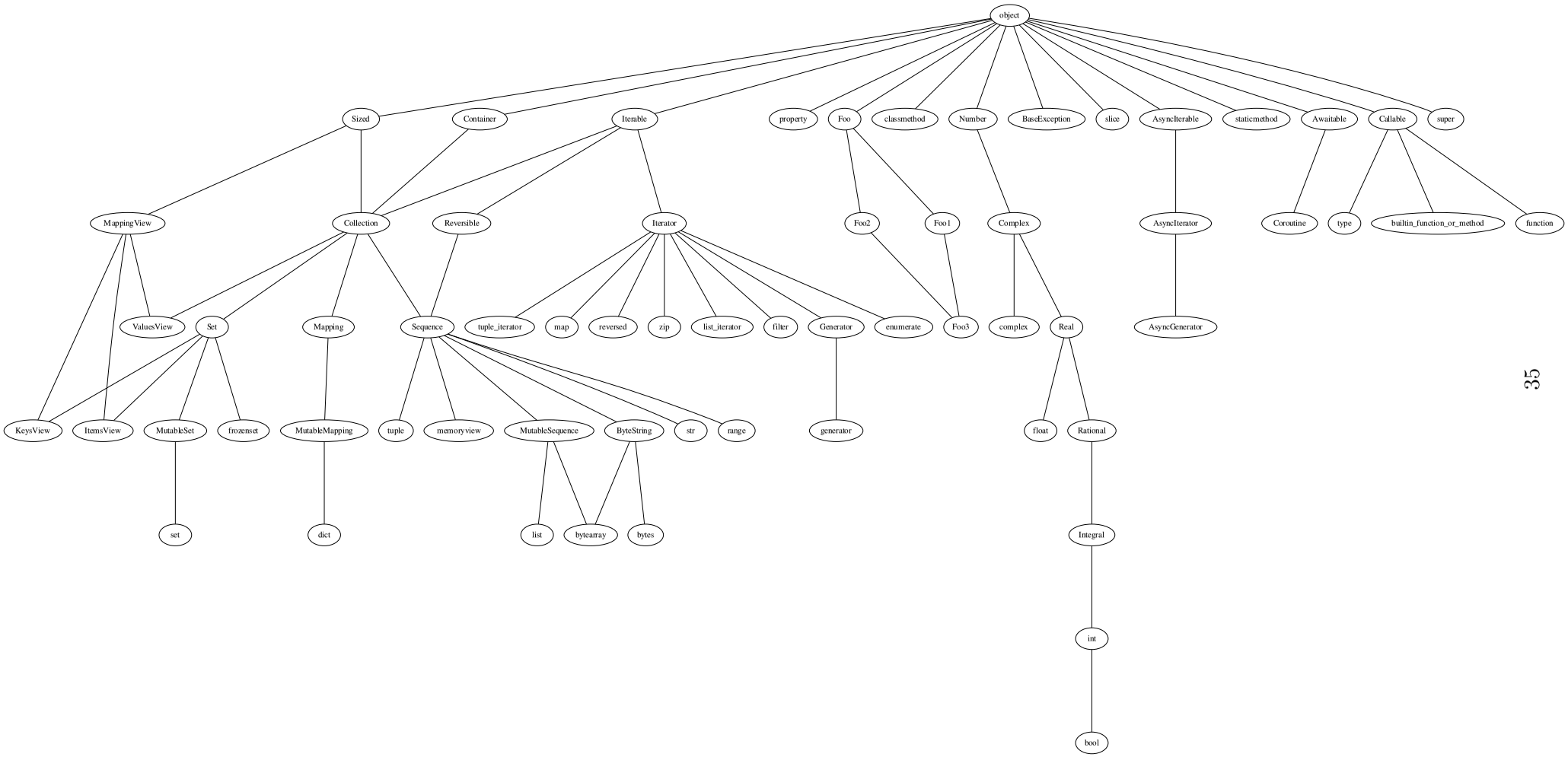
generator, we set the *is_gen* attribute in this object to true. The declaration of a new class *name*, inheriting from expressions *supers* is done through a call to the builtin method *type*. If this call is successful, we set the variable *name* to the newly created class.

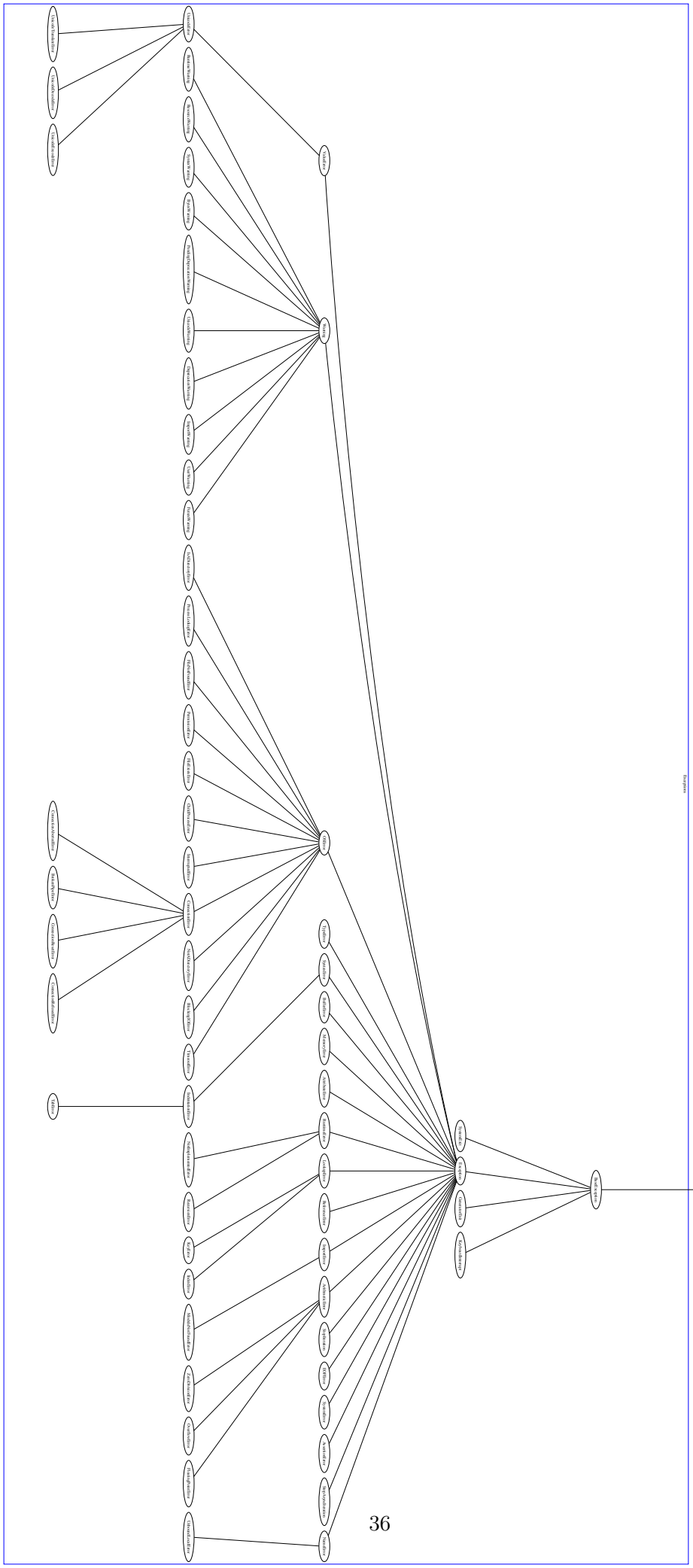
$$
\begin{array}{l}
\mathbb{S}[\![\, \texttt{fun}(name, args, body) \,]\!]\, S \stackrel{\text{def}}{=} \\
\quad \{\, (f, \epsilon, \sigma) \in S \mid f \neq cur \,\} \cup \\
\quad \{\, (cur, \epsilon[name] \leftarrow fa, \sigma[fa] \leftarrow (\mathbf{Fun}(args, body, false), \emptyset)) \mid (cur, \epsilon, \sigma) \in S \,\} \\
\\
\mathbb{S}[\![\, \texttt{gen}(name, args, body) \,]\!]\, S \stackrel{\text{def}}{=} \\
\quad \{\, (f, \epsilon, \sigma) \in S \mid f \neq cur \,\} \cup \\
\quad \{\, (cur, \epsilon[name] \leftarrow fa, \sigma[fa] \leftarrow (\mathbf{Fun}(args, body, true), \emptyset))) \mid (cur, \epsilon, \sigma) \in S \,\} \\
\\
\mathbb{S}[\![\, \texttt{class}(name, supers, body) \,]\!]\, S \stackrel{\text{def}}{=} \\
\quad \text{let } S_1 = \mathbb{E}[\![\, type(name, (\texttt{object}, supers), body) \,]\!]\, (S) \text{ in} \\
\quad \{\, (f, \epsilon, \sigma) \mid (f, \epsilon, \sigma, v) \in S_1 \wedge f \neq cur \,\} \cup \\
\quad \{\, cur, \epsilon[name] \leftarrow v, \sigma) \mid (cur, \epsilon, \sigma, v) \in S_1 \,\}
\end{array}
$$

Figure 26: Concrete semantics of declarations.

C Inheritance graph for built-in python classes

The first graph shows built-in python classes, without any exception. The second shows only the exceptions.


object
Sized
Container
Iterable
property
Foo
classmethod
Number
BaseException
slice
AsyncIterable
staticmethod
Awaitable
Callable
super
MappingView
Collection
Reversible
Iterator
Foo2
Foo1
Complex
AsyncIterator
Coroutine
type
builtin_function_or_method
function
ValuesView
Set
Mapping
Sequence
tuple_iterator
map
reversed
zip
list_iterator
filter
Generator
enumerate
Foo3
complex
Real
AsyncGenerator
KeysView
ItemsView
MutableSet
frozenset
MutableMapping
tuple
memoryview
MutableSequence
ByteString
str
range
generator
float
Rational
set
dict
list
bytearray
bytes
Integral
int
bool


functions

D Files Used in the Comparison with Pytype, Typpete and the Tool from Fritz and Hage

In the following, the boolean conditions written `if *` represent a non-deterministic choice. If the analyzer does not evaluate trivial conditions, an expression such as $1 \geq 0$ is sufficient to represent this non-determinism.

Listing 1: `class_attr_ok.py`

```
class A: pass

x = A()
if *:
    y = x
else:
    y = A()
y.a = 2
```

Listing 2: `class_pre_store.py`

```
# Coming from Typpete
class A:
    def f(self):
        return B()

    def g(self):
        return self.f().g()


class B(A):
    def g(self):
        return "string"


x = A()
y = x.g()
```

Listing 3: `default_args_class`

```
# Coming from Typpete
class A:
    def __init__(self, x=1):
        pass


def f(A):
    x = A()
    y = A(1)

f(A)

x = A()
y = A(1)
```

Listing 4: `except_clause`

```
# Coming from Typpete
class MyException(Exception):
    def __init__(self):
        self.val = 15
```

```
try:
    a = 23
    raise MyException
except MyException as e:
    b = e.val
    a = b + 2
```

Listing 5: `fspath.py`

```
def fspath(p):
    # a simplified version of os.fspath
    # same example with **custom** classes would not be analyzable
    if isinstance(p, str) or isinstance(p, bytes):
        return p
    elif hasattr(p, "__fspath__"):
        res = p.__fspath__()
        if isinstance(res, str) or isinstance(res, bytes):
            return res
        else:
            raise TypeError("__fspath__ should return str or bytes")
    else:
        raise TypeError("input should have type str, bytes or attribute __fspath__, found type %s
     instead" % type(p).__name__)

class FSPath1: pass
class FSPath2:
    def __fspath__(self):
        return 42
class FSPath3:
    def __fspath__(self):
        return "bli"


f1 = fspath("bla")
f2 = fspath(b'bla')
f5 = fspath(FSPath3())
f3 = fspath(FSPath1())
f4 = fspath(FSPath2())
```

Listing 6: `magic.py`

```
# Coming from Typpete
T = 3

for x in [1, 2, 3, T + 1]:

    l1 = 2
    for i in [1, l1]:
        pass
    L1 = [9, 10, 11, 12]
    for i in [l1 + 1, 4]:
        pass

    l2 = 3
    for i in [1, 2, l2]:
        pass
    L2 = [9, 10, 7, 12]
    for i in [l2 + 1]:
        pass

    z = 0
    n = 0
```

```
    for i in [0, 1, 2, 3]:
        for j in [0, 1, 2, 3]:
            if L1[i] == L2[j]:
                z = z + 1
                n = L1[i]

    if z == 1:
        res = ""
    else:
        if z == 0:
            res = ""
        else:
            res = ""
```

Listing 7: `polyfib.py`

```
def fib(n, v0, v1):
    if n <= 0:
        return v0
    elif n == 1:
        return v1
    else:
        return fib(n-1, v0, v1) + fib(n-2, v0, v1)

class AddAbsorb:
    def __add__(self, other):
        return AddAbsorb()

if __name__ == "__main__":
    a = fib(6, 0, 1)
    b = fib(6, "", " ")
    c = fib(6, [1], [[1]])
    d = fib(10, AddAbsorb(), [["a"]])
```

Listing 8: `poly_lists.py`

```
if *:
    x = 1
    l = [2, x]
elif *:
    x = "a"
    l = [x, "bla"]
else:
    x = [1]
    l = [x, [2, 3]]


if isinstance(x, int):
    y = True
elif isinstance(x, str):
    y = "True"
else:
    y = x

t = y
```

Listing 9: `vehicle.py`

```
# Coming from Typpete
from abc import ABCMeta, abstractmethod

class Vehicle(metaclass=ABCMeta):
```

```
    """A vehicle for sale by Jeffco Car Dealership.

    Attributes:
        miles: The integral number of miles driven on the vehicle.
        make: The make of the vehicle as a string.
        model: The model of the vehicle as a string.
        year: The integral year the vehicle was built.
        sold_on: The date the vehicle was sold.
    """

    base_sale_price = 0
    wheels = 0

    def __init__(self, miles, make, model, year, sold_on):
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on

    def sale_price(self):
        """Return the sale price for this vehicle as a float amount."""
        if self.sold_on is not None:
            return 0.0  # Already sold
        return 5000.0 * self.wheels

    def purchase_price(self):
        """Return the price for which we would pay to purchase the vehicle."""
        if self.sold_on is None:
            return 0.0  # Not yet sold
        return self.base_sale_price - (.10 * self.miles)

    @abstractmethod
    def vehicle_type(self):
        """Return a string representing the type of vehicle this is."""
        pass


class Car(Vehicle):
    """A car for sale by Jeffco Car Dealership."""

    base_sale_price = 8000
    wheels = 4

    def vehicle_type(self):
        """Return a string representing the type of vehicle this is."""
        return 'car'


class Truck(Vehicle):
    """A truck for sale by Jeffco Car Dealership."""

    base_sale_price = 10000
    wheels = 4

    def vehicle_type(self):
        """Return a string representing the type of vehicle this is."""
        return 'truck'


class Motorcycle(Vehicle):
    """A motorcycle for sale by Jeffco Car Dealership."""
```

```
    base_sale_price = 4000
    wheels = 2

    def vehicle_type(self):
        """Return a string representing the type of vehicle this is."""
        return 'motorcycle'


car = Car(0, "", "", 2004, 2004)
cp = car.sale_price()
truck = Truck(1000, "", "", 2009, 2011)
tp = truck.purchase_price()
motorcycle = Motorcycle(5000, "", "", 2016, 2017)
mt = motorcycle.vehicle_type()
```

Listing 10: `widening.py`

```
x = 1
i = 0
while i < 100:
    x = [x]
    i = i + 1
y = x
```