Static Value Analysis of Python Programs by Abstract Interpretation

Aymeric Fromherz, Abdelraouf Ouadjaout, Antoine Miné

Static Value Analysis of Python Programs by Abstract Interpretation*

Aymeric Fromherz[1,2], Abdelraouf Ouadjaout[2], and Antoine Miné[2]

[1] Carnegie Mellon University
afromher@andrew.cmu.edu
[2] Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, F-75005 Paris, France
{abdelraouf.ouadjaout,antoine.mine}@lip6.fr

Abstract. We propose a static analysis by abstract interpretation for a significant subset of Python to infer variable values, run-time errors, and uncaught exceptions. Python is a high-level language with dynamic typing, a class-based object system, complex control structures such as generators, and a large library of builtin objects. This makes static reasoning on Python programs challenging. The control flow is highly dependent on the type of values, which we thus infer accurately.
As Python lacks a formal specification, we first present a concrete collecting semantics of reachable program states. We then propose a non-relational flow-sensitive type and value analysis based on simple abstract domains for each type, and handle non-local control such as exceptions through continuations. We show how to infer relational numeric invariants by leveraging the type information we gather. Finally, we propose a relational abstraction of generators to count the number of available elements and prove that no *StopIteration* exception is raised.
Our prototype implementation is heavily in development; it does not support some Python features, such as recursion nor the *compile* builtin, and it handles only a small part of the builtin objects and standard library. Nevertheless, we are able to present preliminary experimental results on analyzing actual, if small, Python code from a benchmarking application and a regression test suite.

1 Introduction

Sound static analyzers based on abstract interpretation [7] have been successful in formally checking correctness properties of programs. Academic and industrial successes include, for instance, Polyspace Verifier, Astrée [5], Sparrow [18], and Julia [26]. The major part of these analyzers target solely statically typed languages, such as C, Java, or C#. With the rise of web applications, the static analysis of JavaScript programs has started to gain some attention [2,14,15]. The more dynamic nature of the language makes this task challenging. In this

* This work is partially supported by the European Research Council under Consolidator Grant Agreement 681393 – MOPSA.

article, we look at another dynamic language, Python [21], that, we feel, has been largely neglected by the static analysis community.

Python is a relatively recent programming language, introduced in 1991, which has gained a lot of popularity due to its readable syntax, ease of programming, interactive toplevel, and large library support. It is used notably in education and science, for beginners and non-computer scientists, as a scripting and prototyping language. It is an interpreted language with dynamic features, including dynamic typing (variables are not typed, and can hold values of any type), a class system supporting object run-time alteration (adding fields beyond what is declared in the object class, and possibly altering the class hierarchy), overloading for methods but also builtin language operators (such as +), reflection, closures, and an *eval* keyword. While these dynamic features are a popular aspect of the language, and are effectively relied on in Python programs [1], they make reasoning on Python programs and ensuring the absence of run-time errors very difficult at compile-time. This has motivated the design of Python subsets and variants with more static typing [3], but that does not help with the large majority of existing Python code that does not obey these restrictions.

We design instead a specific analysis for Python that embraces fully the dynamic aspects of the language — we nevertheless draw the line and reject programs featuring code generation, calling *eval* or *compile* builtins, or importing modules from locations that are not statically known. Our abstract interpreter infers the possible values of program variables in a flow-sensitive way. This information allows us to derive the possible types of each variable at each program point, and hence deduce the control flow for the next instruction. Our analysis then detects soundly all possible run-time errors, that is, uncaught exceptions.

Formal semantics. Following the standard abstract interpretation road-map, we define a concrete collecting semantics, and then derive an effective analyzer by abstraction. An additional difficulty of Python is the lack of formal specification — unlike, for instance, JavaScript, that features an English specification [9] that provides a sound basis for formal specifications [6]. The Python language is defined by its reference manual [21], which leaves room for ambiguity and permits implementation freedom. We base our own semantics on earlier formalization efforts [20], on the reference manual [21], and on the CPython reference implementation. We innovate by defining the semantics as an input-output function on environments, by induction on the syntax with explicit fixpoints for loops, which lends itself well to the design of an abstract interpreter.

Value and type analysis. The core component of our analyzer employs non-relational abstractions, assigning an abstract set of values to each variable. Following the JavaScript analysis by Jensen et al. [14], each variable is given a tuple of abstract values to account for values of all possible types. We employ standard numeric domains, as well as field-sensitive representations for objects abstracted by allocation site, and simple abstractions of builtin Python types (e.g., strings are represented as finite concrete sets or $\top$; lists are represented as a summary object and a length information, etc.). Consider, for instance, that

```
1 def init(f, n=None):
2   if n is None or n <= 0 : return []
3   l = []
4   for i in range(n):
5     l.append(f(i))
6   assert len(l) == n
```

(a) Dynamic typing example.

```
1 def gen():
2   for i in range(0,10):
3     yield i
4 b = gen()
5 for j in range(0,5):
6   a = next(b) # no StopIteration
```

(b) Generator example.

Fig. 1: Python programs illustrating challenging static analysis situations.

the function `init` in Fig. 1a is called with a function object argument `f` and an optional argument `n` with a default `None` value. Our value analysis will infer that `n` is never `None` when `range(n)` is evaluated, so that no exception is raised at this point. Additionally, attribute, method, and operator resolution is handled easily by extracting type information from the value abstraction. Another complication we handle is that attributes and methods can be added dynamically to an object beyond what is statically declared in its class.

Relational numeric analysis. Additionally, we show how we can go beyond non-relational abstractions and leverage numeric relational domains, such as polyhedra [8], which are invaluable to program analysis — notably to infer non-trivial inductive loop invariants. We rely on a reduction with the non-relational domains to deduce variables that are purely numeric at each program point and can thus be fed to a relational domain. When applying our relational analysis on the previous example shown in Fig. 1a, we are able to prove the assertion at line 6, while the non-relational value analysis will raise a false alarm.

Generators. A unique characteristic of Python is the pervasive use of generators, a limited form of co-routines that permeate the standard library. The example in Fig. 1b creates a generator *gen* that returns a new value in $0, 1, \ldots, 9$ at each call to `next`. More precisely, each call to `next` resumes the execution of the iterator, until it calls `yield` and the control is returned to the caller, until the next call to `next`, etc. We develop specific abstractions to model generators, and use a continuation-based iterator to analyze complex, non-local inter-procedural control in an abstract interpreter by induction on the syntax. Combined with relational invariants, the analyzer is able to prove that there are less calls to `next` than to `yield`, so that a *StopIteration* exception is never raised.

Implementation. We have implemented a prototype analyzer and run it on a small set of Python benchmarks. The output of the analysis is a superset of all the possible variable values at each program point as well as the set of uncaught exceptions. Note that Python is a large language with many builtin types, primitives, and standard support libraries. We currently support a selected representative set of primitives, that are sufficient to analyze our benchmarks.

Focus and limitations. Although we believe that our design is sound and scalable, it currently employs some very naive abstractions with respect to the state of the art. Our almost-concrete string abstraction could be replaced with the complex abstractions designed by Amadini et al. for JavaScript [2]. Likewise, object abstractions have been studied extensively, especially for the analysis of Java, and we could replace our simple allocation-site abstraction based on recency abstraction [4] with more efficient ones, such as object-sensitive abstractions [24].

Python instructions involving dynamic code generation or retrieval, including *eval* and *compile*, are not supported — although existing work on JavaScript [13] could help. Likewise, we do not support recursive procedures, which are not much employed in Python — classic interprocedural analysis techniques [23] could also apply. Integrating and evaluating these previous works in the context of Python analysis is left as future work. We chose instead to focus our research on novel aspects of the analysis of Python: the integration of relational abstract domains, and the support for generators, which were not considered in previous works.

Finally, the currently scarce support for Python builtins and libraries severely limits the practical usability of our prototype on realistic Python code. We are more interested at the moment in developing relational analyses that go beyond, in term of expressiveness, current analyses for dynamic languages, than supporting imprecisely the entirety of the language primitives. We also note that, to our knowledge, none of the proposed formal semantics of Python [22,11,19,20] were mature enough to analyze actual Python programs without rewriting them, while we are at least able to analyze small benchmarks and tests unmodified.

Organization. The rest of the article is organized as follows: Sect. 2 presents the syntax and concrete collecting semantics of our normalized Python subset; Sect. 3 presents a non-relational analysis based on replacing the concrete domain with abstract value domains, as well as a relational abstraction; Sect. 4 presents our generator analysis; Sect. 5 presents our implementation and experimental results. Finally, Sect. 6 discusses related work and Sect. 7 concludes.

2 The Mini-Python Language

The language we analyze is a significant subset of Python 3.6, using a simplified syntax removing redundant constructions and syntactic sugar, that we call Mini-Python. Some features that are supported by our implementation are not described here for simplicity: slices, `for` loops, and `import` directives. Some other omitted features are not supported at the moment: *eval* and *compile*, recursion, coroutines (although we do support generators).

2.1 Syntax

Following the Python language reference [21], we distinguish between expressions, that return a value, and statements, that do not.

$$
\begin{array}{lll}
\mathit{expr} ::= \texttt{True} \mid \texttt{False} \mid i \in \mathbb{Z} \mid s \in \mathit{string} & & \textit{(constants)} \\
\quad \mid \texttt{None} \mid \texttt{NotImpl} \mid \texttt{Undef} & & \textit{(singletons)} \\
\quad \mid (\mathit{expr}, \ldots, \mathit{expr}) & \textit{(tuples)} & \mid \ \mathit{expr}.\mathit{string} \quad \textit{(attributes)} \\
\quad \mid \mathit{id} & \textit{(identifier)} & \mid \ \mathit{expr} \circ \mathit{expr} \quad \textit{(binary op.)} \\
\quad \mid \diamond \mathit{expr} & \textit{(unary op.)} & \mid \ \mathit{expr}[\mathit{expr}] \quad \textit{(subscript)} \\
\quad \mid \mathit{expr}(\mathit{expr}, \ldots, \mathit{expr}) & \textit{(call)} & \mid \ \texttt{next}\ \mathit{expr} \quad \textit{(generator next)}
\end{array}
$$

Fig. 2: Mini-Python expressions.

$$
\begin{array}{lll}
\mathit{stat} ::= \mathit{expr} & \textit{(evaluation)} & \mid \ \mathit{id} \leftarrow \mathit{expr} \quad \textit{(assignment)} \\
\quad \mid \mathit{id}.\mathit{string} \leftarrow \mathit{expr} & \textit{(attribute set)} & \mid \ \texttt{return}\ \mathit{expr} \quad \textit{(return)} \\
\quad \mid \texttt{break} & \textit{(loop exit)} & \mid \ \texttt{continue} \quad \textit{(go to loop head)} \\
\quad \mid \texttt{raise}\ \mathit{expr} & \textit{(exception)} & \mid \ \texttt{yield}_i\ \mathit{expr} \quad \textit{(generator exit)} \\
\quad \mid \mathit{stat}; \mathit{stat} & \textit{(sequence)} & \mid \ \texttt{while}(\mathit{expr}, \mathit{stat}) \quad \textit{(loop)} \\
\quad \mid \texttt{if\_then\_else}(\mathit{expr}, \mathit{stat}, \mathit{stat}) & & \textit{(conditional)} \\
\quad \mid \texttt{try\_except\_else}(\mathit{stat}, (\mathit{string} \times \mathit{stat})^*, \mathit{stat}) & & \textit{(exception handling)} \\
\quad \mid \texttt{fun}(\mathit{string}, \mathit{string}^*, \mathit{stat}) & & \textit{(function declaration)} \\
\quad \mid \texttt{gen}(\mathit{string}, \mathit{string}^*, \mathit{stat}) & & \textit{(generator declaration)} \\
\quad \mid \texttt{class}(\mathit{string}, \mathit{expr}^*, \mathit{stat}) & & \textit{(class declaration)}
\end{array}
$$

Fig. 3: Mini-Python statements.

Expressions. Expressions, presented in Fig. 2, include constants of various types: integers (in $\mathbb{Z}$), booleans (`True`, `False`), strings. `None` and `NotImpl` are types with a single inhabitant each, also denoted as `None` and `NotImpl`. They represent respectively the absence of a value and of a special method (such as *__add__*, modeling +). `Undef` denotes the value of uninitialized variables. Expressions also include literal tuples $(e_1, \ldots, e_n)$, object attributes $e.\mathit{string}$, identifiers for variables, functions, and classes, an element of a collection $e_1[e_2]$ and, finally, a call $e(e_1, \ldots, e_n)$ to any callable object: function, generator, class constructor.

Statements. Fig. 3 presents the syntax of statements. Most are standard: atomic statements such as expression evaluation, assignment $e_1 \leftarrow e_2$, attribute update $e_1.\mathit{string} \leftarrow e_2$; control instructions such as `return` e, `break`, `continue`, tests `if_then_else`(c, t, e), and loops `while`(c, b). Exceptions are raised through `raise` e and caught through `try_except_else`$(e, \mathit{clauses}, \mathit{else})$, where *clauses* is a list of pairs $(\mathit{name}, \mathit{body})$ assigning a body to specific exception classes, and *else* to execute when no exception is raised. Generators generate a value using $\texttt{yield}_i\ e$, while the next element of a generator is queried with `next` o, passing the generator object o as argument — $o.\_\_\mathit{next}\_\_()$ in Python. Each $\texttt{yield}_i\ e$ statement is subscripted with a unique syntactic token $i \in \mathbb{N}$, used later in the semantic state of generator instances to remember to which `yield` instruction we should jump back when `next` o is called. Finally, `fun`$(\mathit{name}, \mathit{args}, \mathit{body})$ declares a function with a name, a list of formal arguments, and a body; `gen`$(\mathit{name}, \mathit{args}, \mathit{body})$ declares a generator similarly; and `class`$(\mathit{name}, \mathit{bases}, \mathit{body})$ declares a new class with the given name, inheriting from a list of base classes, and with the given body. Such declarations can appear in any statement, possibly nested in conditionals, loops, or other declarations. Definitions occur at run-time: the act of executing a definition statement creates a new binding in the environment.

There is a unified namespace for variable names, function names, and class names and we assume that all identifiers in the program are unique. We also restrict the language to recursion-free programs. We will be able to encode environments as maps from names to values without ambiguity. Python features unintuitive scoping rules: due to the lack of variable declarations, any assigned variable automatically gets function scope, even if it is used before it is first assigned. We hoist declarations at the function scope level, explicitly assigning them to `Undef`. The analysis is then able to detect `UnboundLocalError` exceptions due to using uninitialized variables, which is a major issue in Python.

2.2 Concrete Collecting Semantics

We define a concrete collecting semantics by induction on the syntax, as a function mapping sets of environments to sets of environments. To handle non-local control flow, such as `break` and `return`, we add a continuation layer to environments. The case of generators is more involved; its description is deferred to Sect. 4. As Python is a large language, we only present here the semantics of a selection of statements that we feel illustrate the specific difficulties of Python semantics and our solutions.

Program environments and values. We denote as **Id** the (finite) set of identifiers used in the program and as **Addr** an infinite set of memory addresses. As usual, a memory state is a pair $m = (\epsilon, \Sigma) \in \mathcal{E} \times \mathcal{H}$, where the environment $\epsilon \in \mathcal{E}$ is a partial function assigning a value to existing variables, while the heap $\Sigma \in \mathcal{H}$ maps currently allocated addresses to objects. Values, in **Val**, can be atomic, such as integers, strings or constants, or addresses of objects, which live in **Obj**:

$$\begin{array}{lcl} \mathcal{E} & \stackrel{\text{def}}{=} & \mathbf{Id} \rightharpoonup \mathbf{Val} \\ \mathcal{H} & \stackrel{\text{def}}{=} & \mathbf{Addr} \rightharpoonup \mathbf{Obj} \\ \mathbf{Obj} & \stackrel{\text{def}}{=} & \mathit{string} \rightharpoonup \mathbf{Val} \\ \mathbf{Val} & \stackrel{\text{def}}{=} & \mathbb{Z} \cup \mathit{string} \cup \{\texttt{True}, \texttt{False}, \texttt{None}, \texttt{NotImpl}, \texttt{Undef}\} \cup \mathbf{Addr} \end{array}$$

Objects map (finitely many) attributes to values. Following Python, we model complex values, including lists, functions, generators, classes, and methods, as objects: $\mathbf{List}, \mathbf{Fun}, \mathbf{Gen}, \mathbf{Class}, \mathbf{Method} \subseteq \mathbf{Obj}$. Their special semantic properties are derived from the presence of some attributes. For instance: a list $l \in \mathbf{List}$ has a length $l.\mathit{length} \in \mathbb{Z}$. We assume that identifiers are strings, $\mathbf{Id} \subseteq \mathit{string}$, which can be exploited to reify environments $\epsilon \in \mathcal{E}$ as objects: $\epsilon \in \mathbf{Obj}$.

States and continuations. To implement non-local control-flow in our input-output semantic, we employ continuations: a semantic state contains not only the current memory state (ϵ, Σ), but also memory states at previously encountered jump points, that are meant to flow into the current state when encountering the corresponding jump target. This technique has been used, for instance, in Astrée [5], to model `break` and `return` in C. For Python, we consider the following flow tokens: $\mathcal{F} \stackrel{\text{def}}{=} \{\, \mathit{cur}, \mathit{ret}, \mathit{brk}, \mathit{cont}, \mathit{exn} \,\}$, where *cur* is the current flow, on which

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

$$\texttt{NameError}(f,\epsilon,\Sigma) \stackrel{\text{def}}{=}$$
$$\quad \text{let } (f_1,\epsilon_1,\Sigma_1,v_1) = \mathbb{E}[\![\, \texttt{NameError}() \,]\!]\,(f,\epsilon,\Sigma) \text{ in}$$
$$\quad (\mathit{exn},\epsilon_1[\mathit{exn\_var} \mapsto v_1],\Sigma_1,\texttt{None})$$
(and similarly for `UnboundLocalError` and `TypeError`)

$$\mathbb{E}[\![\, e_1 + e_2 \,]\!](f,\epsilon,\Sigma) \stackrel{\text{def}}{=}$$
$$\quad \text{if } f \neq \mathit{cur} \text{ then } (f,\epsilon,\Sigma,\texttt{None}) \text{ else}$$
$$\quad \text{let } (f_1,\epsilon_1,\Sigma_1,v_1) = \mathbb{E}[\![\, e_1 \,]\!]\,(f,\epsilon,\Sigma) \text{ in}$$
$$\quad \text{if } f_1 \neq \mathit{cur} \text{ then } (f_1,\epsilon_1,\Sigma_1,v_1) \text{ else}$$
$$\quad \text{let } (f_2,\epsilon_2,\Sigma_2,v_2) = \mathbb{E}[\![\, e_2 \,]\!]\,(f_1,\epsilon_1,\Sigma_1) \text{ in}$$
$$\quad \text{if } f_2 \neq \mathit{cur} \text{ then } (f_2,\epsilon_2,\Sigma_2,v_2) \text{ else}$$
$$\quad \text{if } \mathit{has\_field}(v_1,\mathit{\_\_add\_\_},\Sigma_2) \text{ then}$$
$$\qquad \text{let } (f_3,\epsilon_3,\Sigma_3,v_3) = \mathbb{E}[\![\, v_1.\mathit{\_\_add\_\_}(v_2) \,]\!]\,(f_2,\epsilon_2,\Sigma_2) \text{ in}$$
$$\qquad \text{if } f_3 \neq \mathit{cur} \text{ then } (f_3,\epsilon_3,\Sigma_3,v_3) \text{ else}$$
$$\qquad \text{if } v_3 = \texttt{NotImpl} \wedge \mathit{typeof}(v_1) \neq \mathit{typeof}(v_2) \text{ then}$$
$$\qquad\quad \text{if } \mathit{has\_field}(v_2,\mathit{\_\_radd\_\_},\Sigma_3) \text{ then}$$
$$\qquad\qquad \text{let } (f_4,\epsilon_4,\Sigma_4,v_4) = \mathbb{E}[\![\, v_2.\mathit{\_\_radd\_\_}(v_1) \,]\!]\,(f_3,\epsilon_3,\Sigma_3) \text{ in}$$
$$\qquad\qquad \text{if } f_4 \neq \mathit{cur} \text{ then } (f_4,\epsilon_4,\Sigma_4,v_4) \text{ else}$$
$$\qquad\qquad \text{if } v_4 = \texttt{NotImpl} \text{ then } \texttt{TypeError}(f_4,\epsilon_4,\Sigma_4) \text{ else } (f_4,\epsilon_4,\Sigma_4,v_4)$$
$$\qquad\quad \text{else } \texttt{TypeError}(f_3,\epsilon_3,\Sigma_3)$$
$$\qquad \text{else if } v_3 = \texttt{NotImpl} \text{ then } \texttt{TypeError}(f_3,\epsilon_3,\Sigma_3) \text{ else } (f_3,\epsilon_3,\Sigma_3,v_3)$$
$$\quad \text{else if } \mathit{has\_field}(v_2,\mathit{\_\_radd\_\_},\Sigma_2) \wedge \mathit{typeof}(v_1) \neq \mathit{typeof}(v_2) \text{ then}$$
$$\qquad \text{let } (f_3,\epsilon_3,\Sigma_3,v_3) = \mathbb{E}[\![\, v_2.\mathit{\_\_radd\_\_}(v_1) \,]\!]\,(f_2,\epsilon_2,\Sigma_2) \text{ in}$$
$$\qquad \text{if } f_3 \neq \mathit{cur} \text{ then } (f_3,\epsilon_3,\Sigma_3,v_3) \text{ else}$$
$$\qquad \text{if } v_3 = \texttt{NotImpl} \text{ then } \texttt{TypeError}(f_3,\epsilon_3,\Sigma_3) \text{ else } (f_3,\epsilon_3,\Sigma_3,v_3)$$
$$\quad \text{else } \texttt{TypeError}(f_2,\epsilon_2,\Sigma_2)$$

Fig. 4: Semantics of a few Mini-Python expressions.

most instructions operate; *ret*, *brk*, *cont*, *exn* collect the set of states jumping respectively from a `return`, a `break`, a `continue`, or a `raise` statement to the end of, respectively the current function, loop, loop iteration, or `try` statement. Our semantics manipulates collections of memory states attached to flow tokens. The concrete domain is thus $\mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{F} \times \mathcal{E} \times \mathcal{H})$.

Semantics. Expressions return a value, but can also have side effects — including changing the control flow in case an exception is raised. The semantics $\mathbb{E}[\![\, e \,]\!] : \mathcal{P}(\mathcal{F} \times \mathcal{E} \times \mathcal{H}) \to \mathcal{P}(\mathcal{F} \times \mathcal{E} \times \mathcal{H} \times \mathbf{Val})$ of an expression e in some states thus returns a set of states with a value attached. Many expressions map each state to a single state and value, in which case we define $\mathbb{E}[\![\, e \,]\!]$ as a function $(\mathcal{F} \times \mathcal{E} \times \mathcal{H}) \to (\mathcal{F} \times \mathcal{E} \times \mathcal{H} \times \mathbf{Val})$ and leave implicit the lifting to sets of states. Fig. 4 gives a representative sample of such expression semantics. Expressions are only evaluated for the current flow *cur*, while states attached to other flows "pass

$$\mathbb{S}[\![\,\texttt{if\_then\_else}(e, s_1, s_2)\,]\!]S \stackrel{\text{def}}{=} \mathbb{S}[\![\,s_1\,]\!](\mathit{filter}(e)(S))\,\mathbb{S}[\![\,s_2\,]\!](\mathit{filter}(\neg e)(S))$$

$$\mathbb{S}[\![\,\texttt{break}\,]\!]S \stackrel{\text{def}}{=} \{\,(f,\epsilon,\Sigma) \in S \mid f \neq \mathit{cur}\,\} \cup \{\,(\mathit{brk},\epsilon,\Sigma) \mid (\mathit{cur},\epsilon,\Sigma) \in S\,\}$$

$$\mathit{filter}(e)(S) \stackrel{\text{def}}{=} \cup_{(f,\epsilon,\Sigma)\in S}\ \mathit{filter}(e)(f,\epsilon,\Sigma)$$

$$\mathit{filter}(e)(f,\epsilon,\Sigma) \stackrel{\text{def}}{=} \text{let } (f_1,\epsilon_1,\Sigma_1,v_1) = \mathbb{E}[\![\,e\,]\!](f,\epsilon,\Sigma) \text{ in } \text{if } f_1 \neq \mathit{cur} \text{ then } \{(f_1,\epsilon_1,\Sigma_1)\} \text{ else } \text{let } (f_2,\epsilon_2,\Sigma_2,v_2) = \mathit{is\_true}(f_1,v_1,\epsilon_1,\Sigma_1) \text{ in } \text{if } f_2 \neq \mathit{cur} \text{ then } \{(f_2,\epsilon_2,\Sigma_2)\} \text{ else } \text{if } v_2 = \texttt{True} \text{ then } \{(f_2,\epsilon_2,\Sigma_2)\} \text{ else } \emptyset$$

$$\mathbb{S}[\![\,\texttt{while}(e, c)\,]\!]S \stackrel{\text{def}}{=} \text{let } S_0 = \{\,(f,\epsilon,\Sigma) \in S \mid f \neq \mathit{brk}, \mathit{cont}\,\} \text{ in } \text{let } S_1 = \mathit{filter}(\neg e)(\mathrm{lfp}\ \mathit{post}) \text{ in } \{\,(f,\epsilon,\Sigma) \in S \mid f \in \{\mathit{brk}, \mathit{cont}\}\,\} \cup \{\,(f,\epsilon,\Sigma) \in S_1 \mid f \neq \mathit{brk}, \mathit{cont}\,\} \cup \{\,(\mathit{cur},\epsilon,\Sigma) \mid (\mathit{brk},\epsilon,\Sigma) \in S_1\,\}$$

$$\mathit{post}(T) \stackrel{\text{def}}{=} \text{let } T_0 = \mathbb{S}[\![\,c\,]\!](\mathit{filter}(e)(T)) \text{ in } S_0 \cup T_0 \cup \{\,(\mathit{cur},\epsilon,\Sigma) \mid (\mathit{cont},\epsilon,\Sigma) \in T_0\,\}$$

Fig. 5: Semantics of a few Mini-Python statements.

through" the evaluation unchanged — they return a `None` value which is not used. The case of identifiers $\mathbb{E}[\![\,\mathit{id}\,]\!]$ illustrates the generation of an exception when the variable has not been found or not been initialized: `NameError`() is a constructor that allocates a new object of class `NameError` and returns its address, while the helper function `NameError`(f,ϵ,Σ) binds this new object to the special global variable *exn_var* denoting the currently raised exception, and shifts the flow token to *exn* to instruct the semantics to ignore the effect of instructions on this environment until an `except` statement is encountered; `UnboundLocalError` and `TypeError` behave similarly. The case of the addition + is far more complex, and a good illustration of the complexity of the language — most operators are as complex, and yet sufficiently different from one another to defeat attempts to factor their definitions. We start by evaluating the arguments from left to right. We then execute the *__add__* method from the left argument, if it exists — which is detected using $\mathit{has\_field}(v, \mathit{attr}, \Sigma)$. If it does not exist, or if it returns `NotImpl`, we call the *__radd__* method from the right argument. Note the systematic check that the flow token is still *cur*: a change of flow token denotes an exception that causes the operator to abort while returning the latest environment.

We denote as $\mathbb{S}[\![\,s\,]\!] : \mathcal{P}(\mathcal{F}\times\mathcal{E}\times\mathcal{H}) \to \mathcal{P}(\mathcal{F}\times\mathcal{E}\times\mathcal{H})$ the semantics of a statement s. Using sets of environments allows us to easily chain statements, so that we define $\mathbb{S}[\![\,s_1; s_2\,]\!] \stackrel{\text{def}}{=} \mathbb{S}[\![\,s_2\,]\!] \circ \mathbb{S}[\![\,s_1\,]\!]$. Fig. 5 gives the semantics of a few statements that illustrate the use of non-local control flow. As usual, a test filters its environments to keep only those satisfying the condition, or its negation, to execute the respective branch, and merges them with a union. Filters use *is_true* (omitted for concision) to compute truth values; similarly to +, it successively tries to call the special methods *__bool__* and *__len__*, and returns `True` if none of these methods are implemented. The semantics of loops computes, as usual, a least fixpoint. Its definition is complicated by non-local control: a `break` instruction shifts the current environment into a *brk* continuation, which is consumed by the loop semantics to compute the actual exiting environments. The case of `continue` and `return`, as well as exception handling, is similar.

3 Value Abstraction

We now present our static analysis of Python. Following the Abstract Interpretation framework, it is designed by abstraction of the concrete semantics from the previous section. The result is an interpreter by induction on the syntax following closely the concrete semantics, using standard non-relational and relational domains, and modeling control flow through partitioning by flow tokens.

3.1 Non-Relational Abstraction

We first consider non-relational abstractions: each variable is assigned an abstract value in $\mathbf{Val}^\sharp$ representing a set of possible concrete values. Following [14], we abstract separately each type of values in its abstract domain, while their product $\mathbf{Val}^\sharp$ can represent sets of heterogeneous values:

$$\mathbf{Val}^\sharp \stackrel{\text{def}}{=} \texttt{Undef}^\sharp \times \texttt{None}^\sharp \times \texttt{NotImpl}^\sharp \times \texttt{Bool}^\sharp \times \texttt{Num}^\sharp \times \texttt{String}^\sharp \times \mathcal{P}(\mathbf{Addr}^\sharp)$$

For finite types, each domain tracks the presence of each possible value. For instance, $\texttt{Undef}^\sharp \stackrel{\text{def}}{=} \{\bot, \texttt{Undef}\}$, where $\bot$ denotes the definite absence of `Undef`, while `Undef` denotes the possible presence of `Undef`; $\texttt{None}^\sharp$ and $\texttt{NotImpl}^\sharp$ are similar, while $\texttt{Bool}^\sharp \stackrel{\text{def}}{=} \{\bot, \texttt{True}, \texttt{False}, \top\}$. Our string domain is simply the finite sets of strings, plus a $\top$ element to denote any string: $\texttt{String}^\sharp \stackrel{\text{def}}{=} \mathcal{P}_{\mathit{finite}}(\mathit{string}) \cup \{\top\}$. More clever abstractions, such as [2], will be considered in future work. We can use any non-relational domain for $\texttt{Num}^\sharp$, and our implementation uses integer and float intervals. To finitely represent the heap, we use a classic allocation-site abstraction of $\mathbf{Addr}$ into a finite set $\mathbf{Addr}^\sharp$ of abstract addresses — our implementation actually uses recency abstraction [4], which we omit in our formalization for simplicity. An abstract tuple $V = (V_{\mathit{Undef}}, \ldots, V_{\mathit{String}}, V_{\mathbf{Addr}}) \in \mathbf{Val}^\sharp$ then represents the join of elements from the type-based abstractions:

$$\gamma_{\mathbf{Val}}(V) \stackrel{\text{def}}{=} \gamma_{\mathit{Undef}}(V_{\mathit{Undef}}) \cup \cdots \cup \gamma_{\mathit{String}}(V_{\mathit{String}}) \cup (\cup_{a \in V_{\mathbf{Addr}}} \gamma_{\mathbf{Addr}}(a))$$

The definition of the join $\cup^\sharp_{\mathbf{Val}}$, subset $\subseteq^\sharp_{\mathbf{Val}}$, and widening $\triangledown_{\mathbf{Val}}$ operators on this abstract domain is pointwise and straightforward.

Given abstract values $\mathbf{Val}^\sharp$ and addresses $\mathbf{Addr}^\sharp$, environments $\epsilon^\sharp$ map variables to values, and stores $\Sigma^\sharp$ map addresses to objects, as in the concrete:

$$\begin{aligned} \epsilon^\sharp \in \mathcal{E}^\sharp &\stackrel{\text{def}}{=} \mathbf{Id} \rightharpoonup \mathbf{Val}^\sharp \\ \Sigma^\sharp \in \mathcal{H}^\sharp &\stackrel{\text{def}}{=} \mathbf{Addr}^\sharp \rightharpoonup \mathbf{Obj}^\sharp \\ \text{where} \quad \mathbf{Obj}^\sharp &\stackrel{\text{def}}{=} (\mathit{string} \rightharpoonup \mathbf{Val}^\sharp) \times \mathcal{P}(\mathit{string}) \end{aligned}$$

Due to address abstraction, an abstract object may represent a set of concrete objects with different attributes. Abstract objects are pairs $(\mathit{attr}, \mathit{must}) \in \mathbf{Obj}^\sharp$, where attr maps all possible object attributes to their values, while must is the subset of attributes from $\mathit{dom}(\mathit{attr})$ that are guaranteed to exist in all objects:

$$\gamma_{\mathbf{Obj}}(\mathit{attr}, \mathit{must}) \stackrel{\text{def}}{=} \{\, o \in \mathbf{Obj} \mid \begin{array}{l} \mathit{must} \subseteq \mathit{dom}(o) \subseteq \mathit{dom}(\mathit{attr}) \wedge \\ \forall i \in \mathit{dom}(o) : o(i) \in \gamma_{\mathbf{Val}}(\mathit{attr}(i)) \,\} \end{array}$$

The must information is important to precisely rule out `AttributeError` exceptions.

$$\mathbb{S}^\sharp[\![\,\texttt{break}\,]\!]S^\sharp \stackrel{\text{def}}{=} S^\sharp[cur \mapsto \bot_{\mathcal{M}}, brk \mapsto S^\sharp(cur) \cup^\sharp_{\mathcal{M}} S^\sharp(brk)]$$

$$\begin{array}{l}\mathbb{S}^\sharp[\![\,\texttt{while}(e,c)\,]\!]S^\sharp \stackrel{\text{def}}{=} \\ \quad \text{let } S_0^\sharp = S^\sharp[brk, cont \mapsto \bot_{\mathcal{M}}] \text{ in} \\ \quad \text{let } S_1^\sharp = \mathit{filter}^\sharp(\neg e)(\text{lim } \lambda T^\sharp.\, T^\sharp \triangledown (S_0^\sharp \cup^\sharp \mathit{post}^\sharp(T^\sharp))) \text{ in} \\ \quad S_1^\sharp[cur \mapsto S_1^\sharp(cur) \cup^\sharp S_1^\sharp(brk), brk \mapsto S^\sharp(brk), cont \mapsto S^\sharp(cont)] \\ \quad \text{where } \mathit{post}^\sharp(T^\sharp) \stackrel{\text{def}}{=} (\mathbb{S}^\sharp[\![\,c\,]\!] \circ \mathit{filter}^\sharp(e))(T^\sharp[cur \mapsto T^\sharp(cur) \cup^\sharp T^\sharp(cont)])\end{array}$$

Fig. 6: Abstract semantics of a few Mini-Python constructions.

Finally, we partition abstract states with respect to flow tokens in $\mathcal{F}$. Hence, an abstract element lives in $\mathcal{D}^\sharp \stackrel{\text{def}}{=} \mathcal{F} \to (\mathcal{E}^\sharp \times \mathcal{H}^\sharp)$, with concretization:

$$\begin{array}{l}\gamma(X^\sharp) \stackrel{\text{def}}{=} \{\,(f,\epsilon,\Sigma) \mid (\epsilon,\Sigma) \in \gamma_{\mathcal{M}}(X^\sharp(f))\,\} \\ \text{where } \gamma_{\mathcal{M}}(\epsilon^\sharp,\Sigma^\sharp) \stackrel{\text{def}}{=} \{\,(\epsilon,\Sigma) \mid\ dom(\Sigma) \subseteq (\cup_{a^\sharp \in dom(\Sigma^\sharp)} \gamma_{\mathbf{Addr}}(a^\sharp)) \wedge \\ \qquad\qquad\qquad\qquad \forall i:\ \epsilon(i) \in \gamma_{\mathbf{Val}}(\epsilon^\sharp(i)) \wedge \\ \qquad\qquad\qquad\qquad a \in \gamma_{\mathbf{Addr}}(a^\sharp) \implies \Sigma(a) \in \gamma_{\mathbf{Obj}}(\Sigma^\sharp(a^\sharp))\,\}\end{array}$$

The join $\cup^\sharp$ on abstract states is pointwise. Note that it joins the *must* attribute information for objects with an intersection $\cap$:

$$\begin{array}{l}X_1^\sharp \cup^\sharp X_2^\sharp \stackrel{\text{def}}{=} \lambda F \in \mathcal{F}.\, X_1^\sharp(F) \cup^\sharp_{\mathcal{M}} X_2^\sharp(F) \\ \text{where } (\epsilon_1^\sharp,\Sigma_1^\sharp) \cup^\sharp_{\mathcal{M}} (\epsilon_2^\sharp,\Sigma_2^\sharp) \stackrel{\text{def}}{=} (\lambda i.\, \epsilon_1^\sharp(i) \cup^\sharp_{\mathbf{Val}} \epsilon_2^\sharp(i), \lambda a^\sharp.\, \Sigma_1^\sharp(a^\sharp) \cup^\sharp_{\mathbf{Obj}} \Sigma_2^\sharp(a^\sharp)) \\ \text{and } (a_1,m_1) \cup^\sharp_{\mathbf{Obj}} (a_2,m_2) \stackrel{\text{def}}{=} (\lambda s.\, a_1(s) \cup^\sharp_{\mathbf{Val}} a_2(s), m_1 \cap m_2)\end{array}$$

Fig. 6 gives the abstract semantics for a few Mini-Python constructions. It is similar to the concrete one, up to the partitioning with respect to flow tokens. For instance, a `break` statement merges with a join $\cup^\sharp_{\mathcal{M}}$ the current flow with that of the accumulated break flows, and empties the current flow. Similarly, the loop incorporates back the continue flow at the loop head, and the break flow at its end, after which the continue and break flow from any enclosing loop is restored. Additionally, it replaces the least-fixpoint with a limit lim of the iteration accelerated with a widening $\triangledown$, which applies $\triangledown_{\mathbf{Val}}$ pointwise.

3.2 Relational Abstraction

We now present how we leverage relational numeric domains in a dynamically typed language to improve the analysis precision. The intuition is to maintain relations among pure numeric variables only. We exploit, in a reduced product, the type information provided by $\mathcal{D}^\sharp$ to update the relational invariant dynamically when the type of a variable changes. Let us assume that we are given a numeric abstract domain $\mathcal{N}^\sharp$, such as octagons [17] or polyhedra [8], provided with classic operators, such as a concretization $\gamma_{\mathcal{N}} \in \mathcal{N}^\sharp \to \mathcal{P}(\mathbf{Id} \rightharpoonup \mathbb{Z})$, transfer functions $\mathbb{S}^\sharp_{\mathcal{N}}[\![\,\mathit{stmt}\,]\!] \in \mathcal{N}^\sharp \to \mathcal{N}^\sharp$, and condition filters $\mathit{filter}^\sharp_{\mathcal{N}}(e) \in \mathcal{N}^\sharp \to \mathcal{N}^\sharp$. We define our relation-aware domain as $\mathcal{D}^\sharp_{\mathcal{R}} \stackrel{\text{def}}{=} \mathcal{F} \to \big((\mathcal{E}^\sharp \times \mathcal{H}^\sharp) \times \mathcal{N}^\sharp\big)$ with the following concretization:

$$\begin{array}{l}\gamma_{\mathcal{R}}(X^\sharp) \stackrel{\text{def}}{=} \{\,(f,\epsilon,\Sigma) \mid \text{let } ((\epsilon^\sharp,\Sigma^\sharp),\nu^\sharp) = X^\sharp(f) \text{ in } (\epsilon,\Sigma) \in \gamma_{\mathcal{M}}(\epsilon^\sharp,\Sigma^\sharp) \wedge \\ \qquad\qquad\qquad\qquad \exists \nu \in \gamma_{\mathcal{N}}(\nu^\sharp), \forall id \in dom(\nu) : \epsilon(id) = \nu(id)\,\}\end{array}$$

$\mathbb{S}^\sharp_{\mathcal{R}}[\![\, id \leftarrow e \,]\!] S^\sharp \stackrel{\text{def}}{=}$
$\quad \text{let } S_0^\sharp = \text{merge}(S^\sharp, \mathbb{S}^\sharp[\![\, id \leftarrow e \,]\!](\text{extract}(S^\sharp))) \text{ in}$
$\quad \text{let } ((\epsilon_0^\sharp, \Sigma_0^\sharp), \nu_0^\sharp) = S_0^\sharp(cur) \text{ in}$
$\quad \text{let } \nu_1^\sharp = \text{if } \text{is\_num}(id)(\epsilon_0^\sharp) \text{ then } \mathbb{S}^\sharp_{\mathcal{N}}[\![\, id \leftarrow e \,]\!]\, \nu_0^\sharp \text{ else } \mathbb{S}^\sharp_{\mathcal{N}}[\![\, id \leftarrow \top \,]\!]\, \nu_0^\sharp \text{ in}$
$\quad S_0^\sharp[cur \mapsto ((\epsilon_0^\sharp, \Sigma_0^\sharp), \nu_1^\sharp)]$

$filter^\sharp_{\mathcal{R}}(e) S^\sharp \stackrel{\text{def}}{=}$
$\quad \text{let } S_0^\sharp = \text{merge}(S^\sharp, filter^\sharp(e)(\text{extract}(S^\sharp))) \text{ in}$
$\quad \text{let } ((\epsilon_0^\sharp, \Sigma_0^\sharp), \nu_0^\sharp) = S_0^\sharp(cur) \text{ in}$
$\quad \text{let } \nu_1^\sharp = \cap^\sharp_{v \in \text{vars}(e)} filter^\sharp_{\mathcal{N}}(e \wedge \inf(v)(\epsilon_0^\sharp) \leq v \leq \sup(v)(\epsilon_0^\sharp))(\nu_0^\sharp) \text{ in}$
$\quad S_0^\sharp[cur \mapsto ((\epsilon_0^\sharp, \Sigma_0^\sharp), \nu_1^\sharp)]$

where:
$\text{extract}(S^\sharp) \stackrel{\text{def}}{=} \lambda f.\, \text{let } ((\epsilon^\sharp, \Sigma^\sharp), \_) = S^\sharp(f) \text{ in } (\epsilon^\sharp, \Sigma^\sharp)$
$\text{merge}(S_r^\sharp, S_{nr}^\sharp) \stackrel{\text{def}}{=} \lambda f.\, \text{let } (\_, \nu^\sharp) = S_r^\sharp(f) \text{ in } (S_{nr}^\sharp(f), \nu^\sharp)$
$\text{is\_num}(id)(\epsilon^\sharp) \stackrel{\text{def}}{=} \epsilon^\sharp(id) \subseteq^\sharp_{\mathbf{Val}} (\bot, \bot, \bot, \bot, \top_{num}, \bot, \emptyset)$
$\inf(id)(\epsilon^\sharp) \stackrel{\text{def}}{=} \text{if } \text{is\_num}(id)(\epsilon^\sharp) \text{ then } inf(\epsilon^\sharp(id).num) \text{ else } -\infty$
$\sup(id)(\epsilon^\sharp) \stackrel{\text{def}}{=} \text{if } \text{is\_num}(id)(\epsilon^\sharp) \text{ then } sup(\epsilon^\sharp(id).num) \text{ else } +\infty$

Fig. 7: Abstract relational semantics of atomic statements.

The concretization performs a reduction between the relational and non-relational environments. The reduction with the heap objects is similar but it is omitted here for simplicity.

Some transfer functions in $\mathcal{D}^\sharp_{\mathcal{R}}$ are given in Fig. 7. They show the interaction between $\mathcal{D}^\sharp$ and $\mathcal{N}^\sharp$. After an assignment, the type of the *lhs* variable is checked by the non-relational domain. If its value is necessarily numeric, the statement is also applied in the numeric environment $\nu^\sharp$; otherwise the variable is removed from $\nu^\sharp$. When applying a filter, mixed-type variables can be constrained to become pure numeric variables. The pre-condition numeric environment $\nu_0^\sharp$ has no information on them, and they are thus created and initialized with interval information extracted from the non-relational environment $\epsilon_0^\sharp$.

We illustrate these abstractions through our motivating example from Fig. 1a. Assume that the function `init` is called with the abstract environment $\{(cur, \epsilon^\sharp = \langle \texttt{n} \mapsto (\texttt{None} \vee [10, 100]), \dots \rangle), \nu^\sharp = \top_{\mathcal{N}}, \Sigma^\sharp)\}$. In the `else` branch at line 3, $\epsilon^\sharp$ is filtered and `n` becomes numeric, which allows $\nu^\sharp$ to be refined with the invariant $10 \leq \texttt{n} \leq 100$. An expressive enough domain can then prove $0 \leq i = \texttt{len(l)} < \texttt{n}$ at line 5 inside the loop, so that the `assert` statement at line 6 is satisfied.

4 Generator Analysis

Generators allow a called function to suspend itself with a `yield` statement, storing its state into an object, and resume its execution later with a `next`. We now show how to leverage our continuation-based semantics to analyze them.

$$
\begin{array}{l}
\mathbb{E}[\![\,\texttt{next}\ e\,]\!](f,\epsilon,\Sigma) \stackrel{\text{def}}{=} \\
\quad \text{if } f \neq \mathit{cur} \text{ then } (f,\epsilon,\Sigma,\texttt{None}) \text{ else} \\
\quad \text{let } (f_1,\epsilon_1,\Sigma_1,v) = \mathbb{E}[\![\,e\,]\!](f,\epsilon,\Sigma) \text{ in} \\
\quad \text{if } f_1 \neq \mathit{cur} \text{ then } (f_1,\epsilon_1,\Sigma_1,v) \text{ else} \\
\quad \text{if } \Sigma_1(v) = \mathbf{Gen}(\mathit{cont},\mathit{frame},\mathit{body},\mathit{vars}) \text{ then} \\
\qquad \text{if } \mathit{cont} = \mathit{end} \text{ then } \texttt{StopIteration}(f_1,\epsilon_1,\Sigma_1) \text{ else} \\
\qquad \text{let } S_1 = \text{if } \mathit{cont} = \mathit{start} \text{ then } \mathbb{S}[\![\,\mathit{body}\,]\!](\mathit{cur},\epsilon_1 \cup \mathit{frame},\Sigma_1) \\
\qquad\qquad\qquad\qquad \text{else } \mathbb{S}[\![\,\mathit{body}\,]\!](\mathit{next}(i),\epsilon_1 \cup \mathit{frame},\Sigma_1) \text{ in} \\
\qquad \{\,(\mathit{cur},\epsilon_{|_{\mathbf{Id}\setminus \mathit{var}}},\Sigma[v \mapsto \mathbf{Gen}(j,\epsilon_{|_{\mathit{var}}},\mathit{body},\mathit{vars})],\epsilon(\mathit{yield\_var})) \mid \\
\qquad\quad (\mathit{yield}(j),\epsilon,\Sigma) \in S_1,\ j \in \mathbb{N}\,\} \cup \\
\qquad \{\,(\mathit{exn},\epsilon,\Sigma[v \mapsto \mathbf{Gen}(\mathit{end},[],\mathit{body},\mathit{vars})],\texttt{None}) \mid (\mathit{exn},\epsilon,\Sigma) \in S_1\,\} \cup \\
\qquad \{\,\texttt{StopIteration}(\mathit{cur},\epsilon,\Sigma[v \mapsto \mathbf{Gen}(\mathit{end},[],\mathit{body},\mathit{vars})]) \mid \\
\qquad\quad (f,\epsilon,\Sigma) \in S_1 \wedge f \neq \mathit{exn},\mathit{yield}\,\} \\
\quad \text{else } \mathbb{E}[\![\,v.\_\_\mathit{next}\_\_()\,]\!](f_1,\epsilon_1,\Sigma_1)
\end{array}
$$

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

Fig. 8: Concrete semantics of generators in Mini-Python.

4.1 Concrete Semantics

We extend flow tokens to represent continuations able to jump between `next` and `yield` instructions: $\mathcal{F}_g \stackrel{\text{def}}{=} \mathcal{F} \cup \{\,\mathit{next}(i),\mathit{yield}(i) \mid i \in \mathbb{N}\,\}$, where i represents a syntactic label to identify statements. A generator is an object $g = \mathbf{Gen}(\mathit{cont},\mathit{frame},\mathit{body},\mathit{vars}) \in \mathbf{Gen} \subseteq \mathbf{Obj}$ which is given, upon creation, a body to execute and its set $\mathit{vars} \subseteq \mathbf{Id}$ of local variables. It also maintains some state information: a map $\mathit{frame} \in \mathit{vars} \to \mathbf{Val}$ from local variables to values, stored at `yield` statements and restored at the following `next`, and the location $\mathit{cont} \in \mathcal{C}$ where to resume execution upon the following `next`, where $\mathcal{C} \stackrel{\text{def}}{=} \mathbb{N} \cup \{\mathit{start},\mathit{end}\}$ denotes either the beginning (start) of the generator before the first `next`, or a $\texttt{yield}_i$ statement ($i \in \mathbb{N}$), or the end (end) of the generator — meaning that a call to `next` raises a `StopIteration` exception.

The concrete semantics of `next` and `yield` is given in Fig. 8. Each call to `next` executes the generator body from the beginning but sets the flow token to $\mathit{next}(i)$: it instructs the interpreter to ignore the effect of statements until reaching the corresponding $\texttt{yield}_i$, effectively modeling a jump to the correct location. The $\texttt{yield}_i\ e$ statement uses a $\mathit{yield}(i)$ flow token to skip remaining statements and return to the calling `next`. A global $\mathit{yield\_var}$ variable is used to transfer the value of e to `next`, while $\epsilon_{|_{\mathit{var}}}$ and $\epsilon_{|_{\mathbf{Id}\setminus \mathit{var}}}$ extract the values of the local variables at `yield` and freeze them into the frame attribute of the generator. They are restored into the environment at the following `next`.

4.2 Abstractions

Value abstraction. The concrete modeling of generators can be reduced to simple kinds of operations: flow token updates, and copies between local variables and

$$\mathbb{S}^\sharp[\![\,\texttt{yield}_i\ e\,]\!]S^\sharp \stackrel{\text{def}}{=}$$
$$\text{let } S_1^\sharp = \mathbb{S}^\sharp[\![\,yield\_var \leftarrow e\,]\!]\ S^\sharp \text{ in}$$
$$S_1^\sharp[cur \mapsto S_1^\sharp(next(i)), yield(i) \mapsto S_1^\sharp(cur) \cup^\sharp_{\mathcal{M}} S_1^\sharp(yield(i))]$$

Fig. 9: Abstract semantics of `yield`.

```
1 def gen():
2   for i in range(0,10):
3     yield i
4 b = gen()
5 for j in range(0,5):
6   a = next(b) # no StopIteration
```

(a) Original generator example

```
1 def gen():
2   for i in range(0,10):
3     yield i
4 b = gen()
5 %counter = 0
6 for j in range(0,5):
7   %counter += 1
8   a = next(b) # no StopIteration
```

(b) Instrumented generator

Fig. 10: Generator example (a) and its instrumented version with counters (b).

entries in the generator *frame*, which can be seen as object attributes. Section 3 showed how to abstract these operators, and it is thus easy to enrich it to support generators without the need to enrich the abstract domains at all. This is illustrated in Fig. 9 for the `yield` statement — the case of `next` is similar.

This abstraction is sufficient to infer valuable information on the type, value, and even numeric relations between the local variables of a generator. However, it does not always precisely match the flow of control between `yield` and `next` instructions, leading to spurious exceptions. Consider the example in Fig. 10a. An interval analysis with widening will correctly infer that $i \in [0, 9]$ and $j \in [0, 4]$. However, the abstraction states that all calls to `next` can jump back to the `yield` statement, whatever the number of iterations of the loop indexed by `i`. In particular, at iteration 10, the generator exits the loop, causing a (spurious) *StopIteration* exception.

Counting abstraction. We solve this precision issue by automatically instrumenting programs to keep track of the number of calls to `next` and `yield` through a counter. We show in Fig. 10b a version with this counter explicit.[3] We maintain the counter in both the frame of the caller and the frame of the generator — which is stored in its *frame* attribute. Using a relational domain, such as octagons, allows the analysis to establish both equalities $\%\texttt{counter} = i + 1$ at line 3, and $\%\texttt{counter} = j + 1$ at line 7. As $j \in [0, 3]$, we deduce that $i \in [0, 3]$ as well, i.e., we never actually exit the loop at line 2 and never raise a *StopIteration* exception. Through the counter, relations between a generator and its caller can be established.

[3] A global variable is used for illustration purposes. In practice, a counter is an attribute attached to the generator instance.

Program	Lines	Analysis time	Tests	✓	✗	?	✱	Coverage
`test_augassign`	273	645ms	7	4	0	2	1	85.71%
`test_baseexception`	141	20ms	10	6	0	0	4	60.00%
`test_bool`	294	47ms	26	12	0	0	14	46.15%
`test_builtin`	454	360ms	21	3	0	0	18	14.29%
`test_contains`	77	418ms	4	1	0	0	3	25.00%
`test_int_literal`	91	29ms	6	6	0	0	0	100.00%
`test_int`	218	88ms	8	3	0	0	5	37.50%
`test_list`	106	88ms	9	3	0	0	6	33.33%
`test_unary`	39	11ms	6	2	0	0	4	33.33%

Table 1: Experimental results on regression tests. Result categories: ✓ test passed with no false alarm, ✗ test failed with no false alarm, ? test failed with false alarms, ✱ test containing unsupported builtins.

5 Experimental Evaluation

We have implemented our method in a prototype static analyzer in OCaml and tested it on two categories of benchmarks. Firstly, regression tests from the official Python 3.6.3 distribution were used to assess the correctness of the implementation. Secondly, to evaluate precision and efficiency, we have considered programs from the Python Performance Benchmark Suite,[4] which employ more realistic and challenging constructions. Our analyzer reports all uncaught exceptions: type errors, name errors, unbound locals, stop iterations, failed assertions.

Regression tests. The official regression tests suite consists in a large number of test programs (nearly 500) covering the builtins of the language and the standard libraries. Since our prototype supports only a subset of Python builtins, we have selected only the handful of tests that target the implemented features. The results of the analysis are presented in Table 1.

For each program, we compute the analysis time (in milliseconds) and the number of unit test methods that (i) were proven correct, (ii) raised exceptions and assertion violations, (iii) were not completely analyzed due to the presence of unsupported language features. We investigated the failed tests to check whether the alarms are real or spurious. The obtained outcomes for each regression test are shown in columns 5 to 8. The last column gives the percentage of test methods that we were able to analyze completely. No alarm was detected, which argues in favor of our analyzer faithfully modeling the language semantics, as the tests do not raise errors when executed by the Python interpreter either, and they generally test a single execution. Also, the precision of our prototype analyzer is reflected by the low false alarm rate: only 2 unit tests among 97 resulted in spurious violations of `assert` statements. Finally, due to the incomplete support for builtins, the analyzer was unable to analyze some methods, resulting in low coverage ratios in many cases. However, the analyzer is still under development and features a modular architecture that allows adding missing builtins abstractions easily, without requiring the modification of existing code.

[4] `https://github.com/python/performance`

Relational tests. We have analyzed three programs from the Python Performance Benchmark Suite: `float`, `fannkuch`, and `nbody` (around 270 lines of Python in total) and we varied the underlying numeric domains to show the impact of relational information on the analysis. Firstly, the analysis of these programs using the interval domain terminated in less than 3s: `float` was proven correct but `fannkuch` and `nbody` resulted in a total of 5 false alarms. Using octagons, the analysis time increased to 10mn10s, but the number of false alarms was reduced to 3. Finally, an analysis with the polyhedra domain was able to prove the correctness of both `float` and `fannkuch` under 5s, but the analysis of `nbody` did not terminate before a timeout of 30mn. The scalability is limited because each relational abstract element currently contains all variables and object attributes; classic packing techniques [5] would help us improve this situation.

6 Related Work

Several works aim at restricting Python towards more static typing, as Mypy or RPython [3], to ease program verification. While this would help design future programs, a static analyzer for existing code is still invaluable. Note also that Python features static analyzer tools, such as Pylint that, while helping the user, are not based on a formal semantics and do not attempt to be sound.

While [22] proposed a semantics for an object-free Python, the first realistic formal semantics of Python was proposed by Smeding [25] in 2009 for Python 2.5 in Haskell, followed by [20] for Python 3.2 in Racket, and [11] for Python 3.3 in K, inspired from related work on JavaScript [10]. They present small-step executable operational semantics that aim at being tested against CPython's own regression tests, although experiments were limited by the lack of support for advanced language features and libraries used by the tests — an issue from which we also suffer. Poli [19] presents the first attempt at deriving an abstract semantics, but remains uninstantiated as no abstract value domains are provided. Hassan [12] proposes a static typing using SMT solvers, but require variables to have a single type in the program, a limitation that we overcome. We provide the first complete and implemented abstract interpreter for (a subset of) Python. Unlike previous works, we opted for a big-step semantics, which maps conveniently to an abstract interpreter by induction on the syntax. Continuations have been employed before in abstract interpreters to model control flow, as in Astrée for C [5]; we go one step further by handling exceptions and generators.

We find the works closest to ours in the abstract interpretation of JavaScript. Our non-relational abstraction resembles that of Jensen et al. [14] and later [15]. We go one step further by leveraging relational abstractions as well, which were absent in previous works up to our knowledge. Certain non-relational domains differ due to the different nature of the language and properties we seek, notably our need to under-approximate sets of strings to precisely detect *AttributeError* exceptions. Nevertheless, we could benefit from more advanced string domains as proposed in [16]. Likewise, the analysis of practical uses of *eval* in JavaScript [13] could be the basis to support the equivalent construction in Python.

7 Conclusion

We have presented the first static analysis for a realistic subset of Python, able to infer the types and values of variables, and the exceptions that can be raised. In addition to its novel language target, its main characteristics are the ability to infer numeric relations despite dynamic typing, and the support for generators. Our implementation is currently limited to a small subset of Python builtins and standard libraries; nevertheless, it is sufficient to analyze a few small Python programs from actual tests and benchmarking suites, without modification.

Our prototype is a work in progress. Planned work include completing the support for builtins and libraries to be able to analyze Python applications. We also wish to enrich the abstractions used in our analyzer, targeting in particular abstractions proposed for JavaScript [2,13,14] and Java [24]. The static analysis of dynamic languages, and in particular of Python, is still a new field. There is much to do to raise its effectiveness to that of the analysis of static languages, such as C.

References

1. B. Åkerblom, J. Stendahl, M. Tumlin, and T. Wrigstad. Tracing dynamic features in Python programs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 292–295. ACM, 2014.
2. R. Amadini, A. Jordan, G. Gange, F. Gauthier, P. Schachte, H. Søndergaard, P. Stuckey, and C. Zhang. Combining string abstract domains for JavaScript analysis: An evaluation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017), Uppsala, Sweden*, pages 41–57. Springer, 2017.
3. D. Ancona, M. Ancona, A. Cuni, and N. Matsakis. RPython: A step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, pages 53–64. ACM, 2007.
4. G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *Static Analysis: 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006. Proceedings*, pages 221–239. Springer, 2006.
5. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace*, number 2010-3385 in AIAA, pages 1–38. AIAA (American Institute of Aeronautics and Astronautics), Apr. 2010.
6. M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised JavaScript specification. *SIGPLAN Not.*, 49(1):87–100, January 2014.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM Symp. on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, Jan. 1977.
8. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conf. Rec. of the 5th Annual ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM, 1978.

9. Standard ECMA-262. *ECMAScript 2017 Language Specification, 8th edition*, June 2017.
10. A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 126–150. Springer-Verlag, 2010.
11. D. Guth. A formal semantics of Python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, Jul. 2013.
12. M. Hassan. SMT-based static type inference for Python 3. Bachelor thesis, ETH Zürich, Department of Computer Science, 2017.
13. S.-H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 34–44. ACM, 2012.
14. S.-H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255. Springer-Verlag, 2009.
15. V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: A static analysis platform for JavaScript. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 121–132. ACM, 2014.
16. M. Madsen and E. Andreasen. String analysis for dynamic field access. In *Compiler Construction: 23rd International Conference (CC 2014)*, pages 197–217. Springer, 2014.
17. A. Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19(1):31–100, March 2006.
18. H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. *SIGPLAN Not.*, 47(6):229–238, June 2012.
19. F. Poli. A small step abstract interpreter for (desugared) Python. Master's thesis, Università degli Studi di Padova, Dipartimento di Matematica, 2016.
20. J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: The full monty. *SIGPLAN Not.*, 48(10):217–232, October 2013.
21. Python Software Foundation. *The Python language reference*, 3.6 edition, 2017. `https://docs.python.org/3.6/reference`.
22. J. F. Ranson, H. J. Hamilton, and P. W. L. Fong. A semantics of Python in Isabelle/HOL. Technical report, Department of Computer Science, University of Regina, Dec. 2008.
23. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice-Hall, 1981.
24. Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. *SIGPLAN Not.*, 46(1):17–30, January 2011.
25. G. J. Smeding. An executable operational semantics for Python. Master's thesis, Universiteit Utrecht, 2009.
26. F. Spoto. Julia: A generic static analyser for the Java bytecode. In *Proc. of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP'2005)*, page 17, July 2005.