Value and Allocation Sensitivity in Static Python Analyses*

Raphaël Monat
Sorbonne Université, CNRS, LIP6
F-75005 Paris, France
raphael.monat@lip6.fr

Abdelraouf Ouadjaout
Sorbonne Université, CNRS, LIP6
F-75005 Paris, France
abdelraouf.ouadjaout@lip6.fr

Antoine Miné
Sorbonne Université, CNRS, LIP6
F-75005 Paris, France
Institut Universitaire de France
F-75005 Paris, France
antoine.mine@lip6.fr

Abstract

Sound static analyses for large subsets of static programming languages such as C are now widespread. For example the Astrée static analyzer soundly overapproximates the behavior of C programs that do not contain any dynamic code loading, `longjmp` statements nor recursive functions. The sound and precise analysis of widely used dynamic programming languages like JavaScript and Python remains a challenge. This paper examines the variation of static analyses of Python – in precision, time and memory usage – by adapting three parameters: (i) the value sensitivity, (ii) the allocation sensitivity and (iii) the activation of an abstract garbage collector. It is not clear yet which level of sensitivity constitutes a sweet spot in terms of precision versus efficiency to achieve a meaningful Python analysis. We thus perform an experimental evaluation using a prototype static analyzer on benchmarks a few thousand lines long. Key findings are: the value analysis does not improve the precision over type-related alarms; the value analysis is three times costlier than the type analysis; the allocation sensitivity depends on the value sensitivity; using an abstract garbage collector lowers memory usage and running times, but does not affect precision.

CCS Concepts: • **Theory of computation** → *Program analysis*; • **Software and its engineering** → **Semantics**.

Keywords: Formal Methods, Static Analysis, Abstract Interpretation, Dynamic Programming Language, Python, Experimental Evaluation, Heap Abstraction, Recency Abstraction

1 Introduction

Dynamic programming languages, such as JavaScript and Python, have become increasingly popular over the last years. Python currently ranks as the second most used programming language on Github.[1] It is appreciated for its powerful and permissive high-level syntax, e.g., it allows programmers to redefine all operators (addition, field access, etc.) in custom classes, and comes equipped with a vast standard library. Python is a highly dynamic language, featuring dynamic typing (any variable can point to any value of any type, and this type may change during program execution), introspection (the type of a variable may be inspected at runtime and affect the control-flow) and self-modification (attribute addition/deletion at runtime).

In this work, we use a static analysis by abstract interpretation [3] defined previously [16], implemented in a modular fashion into a prototype static analyzer called Mopsa [12]. We have extended our implementation with a value analysis close to that of [7]. This analysis aims at precisely analyzing Python programs, and in particular detecting runtime errors in those programs, namely, raised exceptions escaping to the toplevel. In order to be precise, this analysis is flow-sensitive, as even variable types may be flow-sensitive. It is also context-sensitive: due to the complex semantics of Python, it would otherwise be impossible to analyze methods without any information on the types of the parameters. Our analysis strives to be sound for Python programs that do not include recursive functions, the `eval` and `super` statements, metaclasses, nor asynchronous operators. It detects all uses of these unsupported features and reports them as unsoundness alarms. In the absence of these features, it is sound by design (up to implementation errors): it overapproximates the concrete semantics of Python programs, which we modeled formally on paper after reading the reference manual

and experimenting with the reference Python interpreter. In particular, we *do* analyze without compromise important dynamic features of Python, including the redefinition of operators, the addition and removal of fields. While only field access through constant names are precisely supported, access through computed names is also soundly supported (taking all possible names into account). This work studies three research questions (RQ) by varying three parameters:

1. As Python is a dynamic programming language without explicit type information, the base of our analysis infers types. Then, we can choose to be value-sensitive by extending it to numerical invariants inference. In Python, it is possible to create programs where a variable will have different types depending on the value of another variable. **RQ1:** *Does this happen in practice? Does the value-sensitivity improve the precision over type errors? What is the cost of adding the value-sensitivity?*

2. Python is also object-oriented, so our analysis uses a heap abstraction to handle objects and aliasing soundly. We use the recency abstraction [1], and we can choose different allocation sensitivities. In particular, we present a variable-policy recency abstraction, allowing different abstract allocation precisions for different objects. **RQ2:** *Which policy offers the best performance-precision ratio? Do these policies depend on the value-sensitivity?*

3. We can also decide to activate an abstract garbage collector [5, 15], which removes unused abstract addresses. As the AGC reduces the size of the abstract states, it may improve the performance of the analyses. **RQ3:** *Is the gain provided significant enough to offset the cost of the AGC?*

As a theoretical evaluation is intractable, we perform an experimental evaluation, and measure the impact of the parameters over the precision as well as the resource consumption of the analysis. We use benchmarks from CPython (the official interpreter), as well as a small real-world utility called PathPicker, which are a few thousand lines long. These benchmarks are fairly small, as the sound static analysis of Python is not as mature as for other languages. Other static analyzers for Python exist: Fritz and Hage developed a dataflow type analysis [6], Google has its own type analyzer called Pytype [13], and Typpete [9] performs type inference and checking through MaxSMT queries. The forementioned parameters cannot be adjusted in the other analyzers, which are thus not included in our experimental evaluation.

We present each parameter in Sections 2, 3 and 4, before presenting the experimental results in Section 5. Section 6 discusses related works and Section 7 concludes.

2 Type and Value Analyses of Python

This section describes the behavior of the implemented type analysis (described in more details in [16]). It then shows the precision gain obtained using values.

Type Analysis. On the example provided in Fig. 1, the analysis infers that l is a list of instances of the Task class.

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(3), Task(5)]
m = 0
for i in range(len(l)):
    m = m + l[i].weight
m = m // (i + 1)
```

Figure 1. Value-sensitivity Example

Lists are currently summarized [8] into one weak content variable (per list) by the analysis. The type analysis infers that weight is an integer, but without a numeric domain, it has to assume it can take possible non-positive integer values. It will thus raise false ValueError alarms during each instantiation of Task. It then infers that m is an integer at line 7. The type analysis does not maintain numeric length information for lists, and thus creates an out-of-bound access alarm (an IndexError exception in Python) at each list access to ensure that all reachable exceptions handled are included in the analysis. As l is a list of Task instances and provided that l[i] is a valid list access, the attribute weight exists for l[i] and is an integer. Knowing that both m and l[i].weight are integers, the type analysis infers that the + operator is resolved as a call to the __add__ method of m, which returns an integer. The type analysis is unable to know for sure that loop body is executed, and in particular that variable i is defined at line 10. It will thus create a NameError false alarm, and the division also raises a ZeroDivisionError. In the end, the analysis inferred precise type information for l and m. It infers that if i exists, then it is an integer, and that seven false alarms can be raised: four ValueErrors, one IndexError, one NameError and one ZeroDivisionError.

Value Analysis. The value analysis is a refinement of the type analysis. For example, during the comparison at line 3, the type analysis infers that weight is an integer, and that the comparison calls int.__lt__(weight, 0). The type analysis infers that this call will return a boolean, and the value analysis refines the result: the return will be True, as the weight is positive. The value analysis infers that l is a list of size four, and that the weight of each Task is between one and five. It finds that the for loop is executed, and that all list accesses are valid, avoiding all seven false alarms generated by the type analysis.

3 Variable-Policy Recency Abstraction

The recency abstraction [1] is used to abstract the heap into a finite number of abstract addresses. In its canonical form, the recency abstraction abstracts addresses by a twofold partitioning. It first splits addresses by their allocation site $l \in \mathbb{L}$, and then through a recency criterion, discriminates the most recent allocation (l, r), where strong updates are

possible, from the older addresses (l, o), which are summarized into a weak address. Given an address to allocate at l, the recency abstraction will return the recent address (l, r). Before that, if the recent address (l, r) was already allocated, the recency abstraction merges the previous recent address into the matching old address (l, o). The recency criterion was introduced to keep precision during the initialization of a structure, which usually happens just after its allocation. Without the recency abstraction, all assignments are weak updates, including the ones at initialization time, and it is not possible to express that an attribute is always initialized, resulting in pervasive `AttributeError` false alarms.

This classic partitioning criterion of heap addresses may be inadequate for our type analysis. Objects of the same type allocated at different locations are often indistinguishable for the type analysis. It is unnecessary, in these cases, to discern between objects of the same type at different allocation sites. For this reason, we assess the impact of changing the first partitioning criterion with a type-based partitioning. The abstract addresses are now of the shape (t, m), where $t \in \mathbb{T}$ is a Python type, and m is the recency criterion $m \in \{r, o\}$.

In the example of Fig. 1, the usual allocation-site-based recency abstraction would yield four strong abstract addresses: one for each of the instantiated `Task`s. With the new partitioning, the evaluation of `Task(2)` line 6 creates the address (Task, r). This address is recent, allowing for strong updates. In particular, the attribute `weight` is added to the instance line 4 for address (Task, r). In the case of the value analysis, the abstract state contains $(\text{Task}, r) \cdot \text{weight} \mapsto [2, 2]$. Then `Task(1)` is evaluated: upon the allocation of a `Task` instance, the recency abstraction detects that (Task, r) is already in use, and moves it to the old part (Task, o) (the state is now $(\text{Task}, o) \cdot \text{weight} \mapsto [2, 2]$). When the `weight` assignment line 4 is analyzed, `self` is still the recent address, allowing for a strong update, and yielding the following state: $(\text{Task}, o) \cdot \text{weight} \mapsto [2, 2], (\text{Task}, r) \cdot \text{weight} \mapsto [1, 1]$. After the analysis of `Task(3)`, the abstract state is $(\text{Task}, o) \cdot \text{weight} \mapsto \mathbf{[1, 2]}, (\text{Task}, r) \cdot \text{weight} \mapsto [3, 3]$ ($[1, 1]$ and $[2, 2]$ are joined in the old address). Note that by using the recency criterion, the abstraction is able to perform strong updates. This way we ensure that the `weight` attribute exists.

We noticed, however, that this partitioning is too coarse in some cases. For example, the lists allocated at lines 1 and 2 in Fig. 2 would be summarized in the same abstract address (list, o) after the allocation at line 3. The variables summarizing list contents are defined using the abstract address of the list. The list contents of `qty` and `els` would thus be summarized in the same variable, meaning that all list accesses to `qty` or `els` would have to return both integers and strings, which lacks precision. In this case, it would be beneficial to partition abstract addresses by allocation site. In some cases, we can even go a step further and partition addresses by (partial) callstacks. This shows that the usual homogeneous partitioning could benefit from different allocation precision

```python
1  qty = [3, 1, 0]
2  els = ['choc', 'flour', 'egg']
3  exp = []
4  for i in range(len(qty)):
5    for j in range(qty[i]):
6      exp.append(els[i])
```

Figure 2. Allocation-sensitivity Example

$$p_{\text{all}} = \lambda(t, l, c, m). \ (t, \top, \top, m)$$

$$p_{\text{loc}} = \lambda(t, l, c, m). \ (t, l, \top, m)$$

$$p_{\text{all}}^{\text{types}}(t, l, c, m) = \begin{cases} p_{\text{loc}}(t, l, c, m) & \text{if } t \in \{ \text{list, tuple, dict} \} \\ p_{\text{all}}(t, l, c, m) & \text{otherwise} \end{cases}$$

$$p_{\text{all}}^{\text{values}}(t, l, c, m) = \begin{cases} p_{\text{loc}}(t, l, c, m) & \text{if } t \in \{ \text{range, slice} \} \\ p_{\text{all}}^{\text{types}}(t, l, c, m) & \text{otherwise} \end{cases}$$

Figure 3. Allocation policies

depending on the type of the address abstracted. This alternative allocation mechanism is called the variable-policy recency abstraction. Abstract addresses are now elements of $\mathbf{Addr}^{\sharp} = \mathbb{T} \times \mathbb{L}^{\top} \times \mathbb{C}^{\top} \times \{r, o\}$, where $\mathbb{T}$ is the set of types, $\mathbb{L}^{\top}$ is the set of program locations extended with top and $\mathbb{C}^{\top}$ is the set of callstacks extended with top. The allocation policy is a function $p : \mathbb{T} \times \mathbb{L} \times \mathbb{C} \times \{r, o\} \rightarrow \mathbf{Addr}^{\sharp}$ which given a type decides which partitioning is applied. This policy is defined by the user at the beginning of the analysis, and used by the recency abstraction as the address constructor.

The policies used in the benchmarks are defined in Fig. 3. The default policy, using only the type-based partitioning is p_{all}, while the one with location sensitivity is p_{loc}. The policy used by the type analysis is $p_{\text{all}}^{\text{types}}$. It uses the type sensitivity by default. Exceptions are made for containers such as lists, tuples and dictionaries where partitioning by location is applied. Type mixes as instanced in Fig. 2 are thus avoided. In order to be sufficiently precise on the numerical values that `range` and `slice` iterators hold, these objects are partitioned by allocation site in the value analysis $p_{\text{all}}^{\text{values}}$. There is however no need for the same precision in the type analysis, as all `range` and `slice` objects have the same integer attributes, only differing in values to which the type analysis is not sensitive. Therefore we find for our **RQ2** that these policies depend on the value-sensitivity.

4 Abstract Garbage Collection

Our first analysis implementation did not include an abstract garbage collector, meaning that all allocated abstract addresses were kept in the abstract environment until the end of the analysis. However, we noticed that up to two thirds of the allocated addresses were unreachable – most objects are stored into local variables, which are removed after function returns. We decided, therefore, to implement an abstract garbage collector (AGC), which detects and removes unreachable abstract addresses and works as a tracing

```
1   class A:
2       def __init__(self, v):
3           self.v = v
4
5   def f(i):
6       b = A(i)
7       c = A(i+1)
8       return b.v
9
10  r1 = f(0)
11  r2 = f(100)
```

Figure 4. Precision Gain Example with the AGC garbage collector. Its roots are all the variables bound in the abstract environment. As the AGC reduces the size of the abstract states, it may improve the performance of the analyses. This, however, may be offset by the cost of the AGC. An experimental evaluation is to be found in the next section.

Note that the AGC may also improve the precision due to its interaction with the recency abstraction (as found in [15]). The AGC may remove addresses that will allow the recency abstraction to be more precise – either by considering previously old addresses recent, or by having fewer objects summarized in an old address. Fig. 4 provides an example for such precision improvements. When performing a value analysis with the allocation policy $p_{\text{all}}^{\text{values}}$, the first call to f(0) line 10 allocates two addresses on the abstract heap, written $(A, o), (A, r)$. In the interval domain, two variables corresponding to the attribute v of each address, exist: $(A, o) \cdot v \mapsto [0, 0]; (A, r) \cdot v \mapsto [1, 1]$. Without the AGC, when analyzing the call to f(100), the first allocation of A triggers the renaming of the recent address into the old one, and binds the value of the recent address to 100, resulting in state ① in the table below. The second allocation triggers another renaming, giving state ②. In the end, the analyzer finds that $r_2 \in [0, 100]$, which is sound but imprecise. However, if the AGC is called between two calls to f at the end of line 10, it is able to detect that both addresses $(A, r), (A, o)$ are dead, and it removes them. This means that during the analysis of the second call, we get state ③ after the first instantiation, and state ④ in the end, so we can infer that $(A, o) \cdot v \mapsto [100, 100]$ after line 7, which is precise.

	l. 6, during f(100)	l. 7, during f(100)
No AGC	$(A, o) \cdot v \mapsto [0, 1]$ ①	$(A, o) \cdot v \mapsto [0, 100]$ ②
	$(A, r) \cdot v \mapsto [100, 100]$	$(A, r) \cdot v \mapsto [101, 101]$
AGC	**(A, o) cleared by AGC**	**$(A, o) \cdot v \mapsto [100, 100]$**
	$(A, r) \cdot v \mapsto [100, 100]$ ③	$(A, r) \cdot v \mapsto [101, 101]$ ④

5 Experimental Results

We empirically study the parameters mentioned in the three previous sections, using a prototype analyzer called Mopsa [12], written in OCaml. Mopsa aims at easing the development of new analyses by splitting them into reusable, loosely-coupled modules. Tables 1, 2 and 3 show the results with two fixed parameters and only one varying. For the sake of readability and concision, we have decided to show only one instantiation of these fixed parameters per table, but the results are similar in the other cases.

We took 12 benchmarks from Python's reference interpreter (prefixed with 🐍). Out of the 44 benchmarks currently available, we chose 12 with no external dependencies and few standard library module dependencies, in order to have less libraries to support. We also analyze the two main parts (processInput.py, choose.py) of a real-world command-line utility from Facebook, called PathPicker (prefixed with ⓕ; the LOC for these files consists in the size of the file and all the PathPicker files imported by this one).

Value Sensitivity. The results of the type and value analyses are to be read in Table 1. Both analyses were performed with no allocation sensitivity (they respectively used $p_{\text{all}}^{\text{type}}$ and $p_{\text{all}}^{\text{values}}$ as allocation policies), and the AGC active. The memory is measured through OCaml's garbage collector statistics, using the maximum size reached by the major heap. Reductions in the number of detected expections are printed in bold. The exceptions detected are split into different categories: • type errors: TypeError, AttributeError exceptions • index errors: out-of-bound list accesses. • key errors: key not found during dictionary access, • math errors: overflows, divisions by zero, • value errors: when an iterable is unpacked in too many values, and • all other errors, including user-defined exceptions. Some exceptions are systematically raised by the type analysis (such as index errors during list accesses). We included these as alarms in the table to show how many potential errors are eliminated by the value analysis. To answer **RQ1**, the precision gained with the value analysis does not remove any type-related error in the programs we analyze. We find that the value analysis is in average 3.25 times slower than the type analysis and similarly, it needs 3 times more memory. In some rare cases, the value analysis is able to detect dead code that the type analysis considers reachable. The value sensitivity does not reduce the key errors (the abstraction is too coarse). For all other exception categories, an improvement in precision is witnessed, e.g. in the case of the index errors, the number of raised alarms is divided by 10.

Allocation-site Sensitivity. We perform an allocation-site sensitivity comparison in Table 2, using the value analysis (the allocation policies are $p_{\text{all}}^{\text{values}}$ and $p_{\text{loc}}^{\text{values}}$) and the AGC. For each allocation policy, the time spent, the memory used, and the number of exceptions detected are indicated. The best results for each policy are printed in bold. Replying to **RQ2**, we find that performing a type-based partitioning at allocation is more efficient, although it may raise a few more alarms – less than 9% more in total. Cases such as chaos and regex illustrate this observation. The analysis of richards reveals that the addition of the location sensitivity may cost an order of magnitude more in resources. This costly increase is a consequence of objects of the same type being allocated at different program locations, resulting in more cases to

Table 1. Value-sensitivity Comparison (no allocation sensitivity, with AGC)

Name	LOC	Type Analysis (policy: $p_{\text{all}}^{\text{types}}$)								Value Analysis (policy: $p_{\text{all}}^{\text{values}}$)							
		Time	Mem.	Exceptions detected						Time	Mem.	Exceptions detected					
				Type	Index	Key	Math	Value	Other			Type	Index	Key	Math	Value	Other
🐍 fannk	59	0.32s	3MB	0	9	0	3	0	0	0.63s	3MB	0	**4**	0	**0**	0	0
🐍 float	63	0.19s	3MB	0	2	0	8	0	0	0.32s	3MB	0	**0**	0	3	0	0
🐍 spectral	74	0.70s	6MB	0	0	0	9	0	1	1.7s	15MB	0	0	0	3	0	**0**
🐍 nbody	157	1.5s	3MB	0	22	1	11	5	1	5.7s	9MB	0	**1**	1	**1**	0	0
🐍 chaos	324	7.4s	42MB	0	28	0	54	10	0	30s	197MB	0	**18**	0	**4**	8	0
🐍 rayt	411	14s	74MB	5	0	0	43	1	1	27s	171MB	5	0	0	**22**	1	**0**
🐍 scimark	416	1.4s	12MB	1	1	0	23	0	0	3.4s	27MB	1	**0**	0	3	0	0
🐍 richards	426	13s	112MB	1	4	0	2	1	1	17s	149MB	1	**2**	0	**0**	1	1
🐍 unpack	458	8.3s	7MB	0	0	0	0	400	0	9.4s	6MB	0	0	0	0	**0**	0
🐍 go	461	27s	345MB	33	20	0	11	0	0	2.0m	1.4GB	33	20	0	**4**	0	0
🐍 hexiom	674	1.1m	525MB	0	46	3	0	2	3	4.7m	3.2GB	0	**21**	3	0	1	2
🐍 regex	1792	23s	18MB	0	2053	0	0	0	0	1.3m	56MB	0	**145**	0	0	0	0
🌐 process	1417	10s	64MB	7	7	1	2	1	2	12s	85MB	7	**4**	1	**0**	1	2
🌐 choose	2562	1.1m	1.6GB	12	22	7	19	18	7	2.9m	3.7GB	12	**13**	7	**11**	18	7
Total	9294	4.0m	2.8GB	59	2214	12	185	438	16	13m	9.1GB	59	228	12	51	30	12

Table 2. Allocation-site Comparison (value analysis & AGC)

Name	No loc. sensitivity: $p_{\text{all}}^{\text{values}}$			Loc. sensitivity: $p_{\text{loc}}^{\text{values}}$		
	Time	Mem.	⚠	Time	Mem.	⚠
🐍 fannk	0.63s	3MB	4	0.63s	3MB	4
🐍 float	**0.32s**	3MB	3	0.39s	3MB	3
🐍 spectral	1.7s	15MB	3	1.7s	15MB	3
🐍 nbody	5.7s	9MB	3	**5.0s**	**9MB**	3
🐍 chaos	**30s**	**197MB**	30	2.4m	1.2GB	**15**
🐍 rayt	27s	171MB	28	**4.5s**	**74MB**	**7**
🐍 scimark	3.4s	27MB	4	**3.0s**	27MB	**3**
🐍 richards	**17s**	**149MB**	5	69m	15GB	5
🐍 unpack	**9.4s**	6MB	0	9.6s	6MB	0
🐍 go	2.0m	1.4GB	57	**1.7m**	**1.2GB**	57
🐍 hexiom	4.7m	3.2GB	27	**4.2m**	3.2GB	27
🐍 regex	**1.3m**	**56MB**	145	3.6m	85MB	145
🌐 process	12s	85MB	15	**11s**	**74MB**	**13**
🌐 choose	**2.9m**	**3.7GB**	68	3.1m	4.3GB	**63**
Total	13m	9.1GB	392	87m	25GB	359

Table 3. AGC Comparison (type analysis, with $p_{\text{all}}^{\text{types}}$)

Name	Without AGC		With AGC		Rel. Impr.	
	Time	Mem.	Time	Mem.	Time	Mem.
🐍 fannk	0.32s	3MB	0.32s	3MB	0%	0%
🐍 float	0.22s	3MB	0.19s	3MB	16%	0%
🐍 spectral	0.72s	6MB	0.70s	6MB	3%	0%
🐍 nbody	1.7s	4MB	1.5s	3MB	10%	25%
🐍 chaos	10s	64MB	7.4s	42MB	28%	34%
🐍 rayt	17s	74MB	14s	74MB	16%	0%
🐍 scimark	1.5s	13MB	1.4s	12MB	5%	8%
🐍 richards	16s	227MB	13s	112MB	21%	51%
🐍 unpack	10s	9MB	8.3s	7MB	19%	22%
🐍 go	38s	604MB	27s	345MB	31%	43%
🐍 hexiom	2.2m	1.1GB	1.1m	525MB	49%	50%
🐍 regex	30s	24MB	23s	18MB	23%	25%
🌐 process	14s	85MB	10s	64MB	28%	25%
🌐 choose	2.0m	3.2GB	1.1m	1.6GB	43%	50%
Total	6.5m	5.4GB	4.0m	2.8GB	38%	47%

be analyzed. In some cases (go, hexiom), the location sensitivity yields quicker analyses, because the analysis without location sensitivity performs more weak updates (when allocating an address that already exists through the recency), which are costlier than having multiple recent addresses in these cases. We tested adding the callstack-sensitivity to the location-sensitivity. In our benchmarks, it achieved the same results in precision, and similar analysis times.

Abstract Garbage Collection. We show the effect of the AGC in Table 3. The AGC does not change the number of exceptions detected in the benchmarks. We believe the situation presented in the previous section where a weak object is accessed, does not happen enough to affect the number of exceptions detected. We show the time spent and the memory used during each analysis. The last column gives the relative change in time spent and memory used when the AGC is enabled. Answering to **RQ3**, the activation of the AGC is extremely beneficial: it almost halves the global memory usage,

and brings a 38% analysis time improvement. In addition, the most significant speedups are observed on the largest files, showing the scalability of the approach. The AGC represents less than 6% of the analysis time for all benchmarks except the last, where it takes 30% of the analysis time. In the results, the AGC is called after each assignment where the right-handside is a (function, method or object) call. We have tested running the AGC at only a fraction of those assignments, but the results were not as satisfying.

6 Related Work

Static Analysis of Dynamic Programming Languages. JavaScript is the most popular dynamic language studied by the static analysis community. One of the first static analyses for Javascript is "Type Analysis for JavaScript" (TAJS) [11]. Contrary to JavaScript, Python uses a class-based inheritance system, operations rely less on strings and there is no implicit type conversion. A few other static analyses for Python

exist: Fritz and Hage propose a type-based dataflow analysis [6], while Pytype [13] performs a type inference that is used by Google to check their code and Typpete [9] encodes type inference rules into a MaxSMT problem. To the best of our knowledge, these analyzers cannot infer numeric invariants, and they do not have an abstract heap or an abstract garbage collector. This is why we could not perform our experiments on them. This work is based on the implementation of the modular type analysis we recently developed [16]. We then refined this type analysis with a value analysis close to the one described in previous work [7]. In that work, Fromherz et al. were able to analyze small programs with relational numeric abstract domains, and the scalability was limited by the standard library support. Here, the scalability is limited by the lack of efficient packing techniques [2]. Without packing, we are able to analyze fannk in 14 seconds with the polyhedra domain, and the number of out-of-bound false alarms is halved. We are also able to analyze float in 1.4s, but then spectral already takes 8.7 minutes.

Heap Abstractions. [14] finds that the recency abstraction [1] leads to high precision in 75% percent of their cases. It also studies the precision improvement when more recent blocks are created per objects. It thus seems natural that most previous analyses of dynamic programming languages taking soundly into account object mutation and aliasing used the recency abstraction [7, 11]. Specialized heap abstractions for dynamic programming languages exist, such as the heap with open objects [4], aiming at precisely analyzing code where object attributes are unknown.

Abstract Garbage Collection. Tracing AGCs were first mentioned in [10], then described by Might and Shivers in [15]. More recently, an AGC based on reference counting was presented [5]. Using an AGC has been found to reduce the analysis time by an order of magnitude, and sometimes to improve the precision. All the authors aimed at analyzing higher-order languages. In this paper, which focuses on dynamic object-oriented programming languages, our results are less impressive, as the relative time improvement is of 38% when using the AGC. Our benchmarks do not show any alarm reduction when the AGC is active, even though we do have examples where the AGC improves the precision of the analysis. Such lack of improvement might be caused by the imprecision of the analysis, such as the analysis of containers. In TAJS [11], the authors find that their AGC reduces their memory consumption, but has little impact on the precision or the analysis time.

7 Conclusion

Although the value analysis reduces the global number of false alarms in our benchmarks, the number of type-related alarms is not reduced, compared to the type analysis. We have found that the value analysis is more costly than the type analysis, but only by a factor 3. The value-sensitivity should thus be chosen with regards to the kind of errors

we are interested in. Our notion of variable-policy recency abstraction, allows for a more flexible partitioning of the abstract addresses, which can be chosen by the user before running the analysis. In order to have the best precision, this configuration depends on the chosen value-sensitivity. In addition to this, being less allocation-sensitive has been shown to improve the analysis times with only a few more false alarms. Though, in our benchmarks, the AGC does not increase the precision of the analysis, it does reduce significantly the running times and the memory used.

Future work includes: (i) making it possible to change the allocation-sensitivity during the analysis, (ii) developing a more modular function analysis, in order to lower the analysis times, (iii) having a relational value analysis that scales, (iv) finding more precise dictionary abstractions, (v) validating these experiments on more realistic Python applications.

References
[1] G. Balakrishnan and T. W. Reps. 2006. Recency-Abstraction for Heap-Allocated Storage. In *SAS (LNCS)*, Vol. 4134. Springer, 221–239.

[2] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. 2010. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace (I@A 2010)*.

[3] P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252.

[4] A. Cox, B.-Y. Evan Chang, and X. Rival. 2014. Automatic Analysis of Open Objects in Dynamic Language Programs. In *SAS (LNCS)*, Vol. 8723. Springer, 134–150.

[5] N. Van Es, Q. Stiévenart, and C. De Roover. 2019. Garbage-Free Abstract Interpretation Through Abstract Reference Counting. In *ECOOP (LIPIcs)*, Vol. 134. 10:1–10:33.

[6] L. Fritz and J. Hage. 2017. Cost versus precision for approximate typing for Python. In *PEPM*. ACM, 89–98.

[7] A. Fromherz, A. Ouadjaout, and A. Miné. 2018. Static Value Analysis of Python Programs by Abstract Interpretation. In *NFM (LNCS)*, Vol. 10811. Springer, 185–202.

[8] D. Gopan, F. DiMaio, N. Dor, T. W. Reps, and S. Sagiv. 2004. Numeric Domains with Summarized Dimensions. In *TACAS (LNCS)*, Vol. 2988. Springer, 512–529.

[9] M. Hassan, C. Urban, M. Eilers, and P. Müller. 2018. MaxSMT-Based Type Inference for Python 3. In *CAV (LNCS)*, Vol. 10982. Springer, 12–19.

[10] S. Jagannathan, P. Thiemann, S. Weeks, and A. K. Wright. 1998. Single and Loving It: Must-Alias Analysis for Higher-Order Languages. In *POPL*. ACM, 329–341.

[11] S. H. Jensen, A. Møller, and P. Thiemann. 2009. Type Analysis for JavaScript. In *SAS (LNCS)*, Vol. 5673. Springer, 238–255.

[12] M. Journault, A. Miné, R. Monat, and A. Ouadjaout. 2019. Combinations of reusable abstract domains for a multilingual static analyzer. In *Proc. of VSTTE19*. 1–17.

[13] M. Kramm, R. Chen, T. Sudol, M. Demello, A. Caceres, D. Baum, A. Peters, P. Ludemann, P. Swartz, N. Batchelder, A. Kaptur, and L. Lindzey. 2019. Pytype. https://github.com/google/pytype.

[14] P. Liang, O. Tripp, M. Naik, and M. Sagiv. 2010. A dynamic evaluation of the precision of static heap abstractions. In *OOPSLA*. ACM, 411–427.

[15] M. Might and O. Shivers. 2006. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *ICFP*. ACM, 13–25.

[16] R. Monat, A. Ouadjaout, and A. Miné. 2020. Static Type Analysis by Abstract Interpretation of Python Programs. In *ECOOP (LIPIcs)*. To appear.