Static Analysis of Python Programs[⋆]

Raphaël Monat[0000−0001−8487−0326]

LIP6, Sorbonne Université
raphael.monat@lip6.fr

Abstract. Python is an increasingly popular dynamic programming language, which is particularly used in the scientific community, and well-known for its powerful and permissive high-level syntax. The goal of this PhD is to develop static analyses for Python, i.e. to detect automatically uncaught exceptions in programs without running them. The current focus is to detect type and attribute errors using a type analysis.

Keywords: Formal Methods · Static Analysis · Abstract Interpretation · Python · Dynamic Programming Language

1 Introduction

Python is a popular programming language: it ranks second on StackOverflow and third on GitHub[1]. Major software projects written in Python include the Django web framework, the SageMath computer algebra system and the TensorFlow machine-learning framework. Python is an object-oriented, interpreted, dynamic programming language, meaning for example that variables are not statically typechecked and metaprogramming features such as introspection are widely used. In particular, errors such as undeclared variables and type incompatibilities are detected at runtime by the interpreter and raised as exceptions, which can be caught by the program itself.

Our goal is to develop static analyses of Python programs through the framework of Abstract Interpretation[7]: we want to write analyzers that read the source code of a Python program and automatically report potential uncaught exceptions to the user. We also want our analyzers to be sound: if no exception is found in a program, then the program is guaranteed to not have uncaught exceptions. Mature static analyzers for statically typed programming languages such as C and Java are common (e.g. Astrée[3], Frama-C[15] and Infer[6]), but only a few analyses exist for dynamically typed languages: some exist for JavaScript [11,14,16] and less for Python [9,12]. Analyzing Python programs present new challenges: the semantics of Python is not well-known and not clearly defined, as it is only specified by the reference interpreter called CPython. The semantics is

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

1 https://insights.stackoverflow.com/survey/2019#technology-_-programming-scripting-and-markup-languages https://githut.info/

also much more complex: for example, the addition operator has 7 different return cases depending on the runtime types of the addition parameters, and may call up to 7 different builtin functions. The object-orientation of Python entails a lot of user-defined function calls in a program, so our analyses need to scale to handle analysis of thousands of function calls. The language is by design quite permissive: variables may be declared only in some parts of the control-flow of a program, and they do not have to be of a constant type.

Another goal of this PhD is to implement and validate theoretically designed analyses within a static analysis framework called MOPSA. The goal of MOPSA is to make the implementation of new static analyses easier, through the modularity of abstract domains.

2 Current work

Our current work aims at developing a type analysis for Python programs, in order to detect uncaught type errors. For each program point and each variable, our analysis infers the set of types matching each potential value of the variable. This is already a challenging task, as we need to take into account object mutability, two notions of types (nominal typing, corresponding to the class from which an object is instantiated, and duck typing, based on attributes) – as well as the complex semantics of the language and the size of the standard library. Our analysis has the following characteristics: it supports aliasing, standard python containers and is flow-sensitive. It also performs polymorphic type inference and partially modular interprocedural analysis. We now describe these features.

Aliasing. We separate our analysis into two parts: a heap abstraction[2] maps variables to abstract addresses they may point to. This heap abstraction is independent from other parts of the analysis and can be changed to achieve different precision levels. The second part is a type domain, mapping abstract addresses to types, where the type consists in the class from which the object is instantiated as well as the attributes that may have been added to this object. This separation lets us analyze aliasing and object mutations: we are able to infer in Fig. 1 that the update of y updates x as well.

```
class A:
  def __init__(self):
    self.val = 0
  def update(self, x):
    self.val = x

x = A()
c = x.val
y = x
y.update('a')
z = x.val
```

Fig. 1. Aliasing example

Python containers. We have added a smashing abstraction[4] for Python containers such as lists. In the case of lists, this smashing means that its contents is summarized into one weak variable for each allocation site. This abstraction is implemented independently from the basic analysis in MOPSA: it could be used as is in other analyses (such as a value analysis). This smashing abstraction could also be swapped with another abstraction (such as expansion), without having to modify the type analysis.

```
def dint(x):
  if isinstance(x, int): return x*2
  else: raise TypeError

try: z2 = f('a')
except TypeError: z2 = None
```

Fig. 2. Control-flow example

Flow-sensitivity. Our analysis is flow-sensitive: variables may have different types in different branches of a conditional. Moreover, our domain is able to take into account calls to `isinstance` or `hasattr` filtering the possible types of a branch. In the example of Fig. 2, we know that the `return` statement is unreachable when the parameter is a string. In addition, our analysis tracks raised exceptions precisely, meaning that we know that `z2 = None` at the end, as a `TypeError` is raised during `f('a')`.

```
if isinstance(p, str):
  r = '/'
else:
  r = b'/'
```

Fig. 3. Example inspired from posixpath._get_sep

Polymorphism. In some cases, a variable may have different types depending on the control-flow, and may be related to another. For example, in Fig. 3, assuming that the variable `p` is an instance of `bytes` or `str`, the variables `p` and `r` have the same type, which is the set $S = \{\texttt{bytes}, \texttt{str}\}$. Our analysis is able to perform a relational analysis and infer that variables `p` and `r` have the same type $\alpha \in S$.

Partially Modular Analysis. Due to the semantics calling many builtin functions and the object-orientation of Python, a lot of function and method calls need to be analyzed even in small programs. As the inlining of function calls was too costly, we developed a cache mechanism reusing the results of previous function analyses provided the abstract environment has not changed. For example, in the benchmark `bm_chaos.py` (see Table 1), the analysis inlines around 5400 calls to user-defined functions. Using the cache, this is reduced to 1700 calls. With this cache reducing the context-sensitivity and with the polymorphism, we can also establish relations between input and output variables: a function inspired from Fig. 3 taking as argument `p` and returning `r` has type $\alpha \rightarrow \alpha$, with $\alpha \in \{\texttt{str}, \texttt{bytes}\}$.

Sound Analysis. We aim at providing a sound analysis, although we do not take support neither metaprogramming nor dynamic code execution through the "eval" statement yet. As of today, we have not found programs where the soundness entailed a big loss in the precision of the analysis. We also believe that specialized sound analyses may help keep sufficient precision in the other cases. An example is the analysis of JavaScript's evals is presented in [13].

Experimental results. We have implemented our analysis into MOPSA. The type analysis consists in 2500 lines of OCaml code, the container abstractions consists in 2100 lines of OCaml, and there are 5500 lines of OCaml code defining the semantics of Python. We currently support more than 200 functions from the standard library. We show the results of our analysis in Table 1, on benchmarks used by the standard Python interpreter. We focused on 7 benchmarks out of 44, which were chosen for their low number of external dependencies. We found one `TypeError` in `bm_chaos.py`[2], which was never reached in the actual test, but could be triggered by instanciating a class using non-default arguments. This error was also detected by Pytype. The last benchmark `bm_hexiom.py` has a number of false alarms mainly due to our analysis being unable to distinguish empty lists from non-empty ones, meaning that we cannot be sure that variables created during iterations over lists are actually defined. We have compared our analyzer with Typpete[12], Pytype[1] and a tool developed by Fritz and Hage[8]: we believe we are uniquely taking into account mutability, control-flow and dynamic attribute addition when analyzing programs.

Name	LOC	Analysis time	# Alarms	# False Alarms
bm_fannkuch.py	59	0.07s	0	0
bm_float.py	63	0.06s	0	0
bm_spectral_norm.py	74	0.33s	0	1
bm_nbody.py	157	1.5s	0	1
bm_chaos.py	324	5.6s	1	0
bm_unpack_sequence.py	458	3.1s	0	0
bm_hexiom.py	674	2m58s	0	52

Table 1. Analysis of official Python benchmarks[3]

Research Approach. Our approach is the following: once we have a kind of analysis in mind, we search for bugs or pieces of real-world code that would benefit from it. We then design the analysis, check that it is sound and implement it. We benchmark it, and try to improve performance and precision issues. For example, we implemented the partially modular analysis after the type analysis was found to be too slow, and the polymorphism was added to gain precision.

3 Related Work

Semantics. We are currently using a slightly upgraded concrete semantics of [9]. Other semantics for Python have been defined in [10,17,18]. Contrary to Python, the JavaScript language is defined through a standard, and different semantics exists, including formal ones[5].

Dynamic Analysis. Tools such as PyAnnotate[4] and MonkeyType[5] run programs to collect the types during execution. While this approach helps during manual program annotation, the types are not sound, as not all branches are explored at runtime.

Gradual Typing. A compromise between statically and dynamically typed languages is gradual typing, where the user annotates parts of the program,

[2] `https://github.com/python/pyperformance/issues/57`
[3] `https://github.com/python/pyperformance/`
[4] `https://github.com/dropbox/pyannotate`
[5] `https://github.com/Instagram/MonkeyType`

which can then be typechecked. Other parts have an unknown, “top” type from which any static type can be cast to and from. If a program gradually typechecks, the only type errors that can occur at runtime are casts from variables having type “top”. Gradual typecheckers for Python include Mypy[6] and Pyre[7]. Both tools however restrict the input language, as a variable should have only one type at runtime. By constrast, our type analysis is more permissive: our goal is not to restrict the dynamic typing features of Python, but to find uncaught type exceptions. Mypy’s type annotations have inspired a new standard for optional type annotations in Python, defined in the PEP 484 [8].

Static Analysis. Three other tools perform a static type analysis of Python programs: Typpete [12] encodes the typing problem into a MaxSMT instance and lets Z3 solve it; Pytype [1] performs a dataflow analysis which is not formally described and a tool written by Fritz and Hage [8] performs another dataflow analysis. Both Typpete and Pytype provide PEP484-compliant type annotations upon successful typing, while the last tool displays the type of each variable at each program point. A value analysis by abstract interpretation is presented in [9]: it is able to infer numerical properties, while our analysis is able to infer polymorphic types.

4 Future work

Target Programs. Our short-term goal is to analyze open-source Python utilities of moderate size (less than 20,000 LOC), as a first step to analyzing general software. In order to analyze these utilities, we plan to research on:

Modular Interprocedural Analysis. Due to the complex semantics of Python, it seems difficult to analyze functions without any context. We would like to develop a summary-based analysis which is able to infer precisely which part of the context is needed to analyze a function. This would improve the reusability of the current cache we have and the overall performances. This analysis would also be able to yield a general type for functions that have one.

Library Analysis. Python uses a lot of libraries and has a vast standard library. In order to scale to bigger Python programs, we plan to design analyses able to infer information on library calls, or able to generate stubs automatically.

Multilingual Analysis. As Python has a lot of libraries written in C, having a multilingual analysis capable of analyzing both the Python program and the C library calls would help analyze projects more precisely.

[6] `http://mypy-lang.org/`
[7] `https://github.com/facebook/pyre-check`
[8] `https://www.python.org/dev/peps/pep-0484/`

Acknowledgments. This is joint work with Abdelraouf Ouadjaout and my PhD advisor, Antoine Miné.

References

1. Pytype. `https://github.com/google/pytype` (2018)
2. Balakrishnan, Reps: Recency-abstraction for heap-allocated storage. In: SAS Proceedings (2006). https://doi.org/10.1007/11823230_15
3. Bertrane, Cousot, Cousot, Feret, Mauborgne, Miné, Rival: Static analysis and verification of aerospace software by abstract interpretation. Foundations and Trends in Programming Languages (2015). https://doi.org/10.1561/2500000002
4. Blanchet, Cousot, Cousot, Feret, Mauborgne, Miné, Monniaux, Rival: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones (2002). https://doi.org/10.1007/3-540-36377-7_5
5. Bodin, Charguéraud, Filaretti, Gardner, Maffeis, Naudziuniene, Schmitt, Smith: A trusted mechanised javascript specification. In: POPL. pp. 87–100. ACM (2014). https://doi.org/10.1145/2535838.2535876
6. Calcagno, Distefano, Dubreil, Gabi, Hooimeijer, Luca, O'Hearn, Papakonstantinou, Purbrick, Rodriguez: Moving fast with software verification. In: NFM 2015 Proceedings. Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
7. Cousot, Cousot: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977. pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973
8. Fritz, Hage: Cost versus precision for approximate typing for python. In: PEPM 2017 Proceedings (2017). https://doi.org/10.1145/3018882.3018888
9. Fromherz, Ouadjaout, Miné: Static value analysis of python programs by abstract interpretation. In: NFM 2018 Proceedings. vol. 10811, pp. 185–202 (2018). https://doi.org/10.1007/978.3.319.77935.5.14
10. Guth, D.: A formal semantics of python 3.3 (2013)
11. Hackett, Guo: Fast and precise hybrid type inference for javascript. In: PLDI. pp. 239–250. ACM (2012). https://doi.org/10.1145/2254064.2254094
12. Hassan, Urban, Eilers, Müller: Maxsmt-based type inference for python 3. In: CAV (2) (2018). https://doi.org/10.1007/978-3-319-96142-2_2
13. Jensen, Jonsson, Møller: Remedying the eval that men do. In: ISSTA 2012. pp. 34–44. ACM (2012). https://doi.org/10.1145/2338965.2336758
14. Jensen, Møller, Thiemann: Type analysis for javascript. In: SAS. vol. 5673, pp. 238–255. Springer (2009). https://doi.org/10.1145/2535838.2535876
15. Kirchner, Kosmatov, Prevosto, Signoles, Yakobowski: Frama-c: A software analysis perspective. Formal Asp. Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
16. Logozzo, Venter: RATA: rapid atomic type analysis by abstract interpretation - application to javascript optimization. In: CC. Springer (2010). https://doi.org/10.1007/978-3-642-11970-5_5
17. Politz, Martinez, Milano, Warren, Patterson, Li, Chitipothu, Krishnamurthi: Python: the full monty. In: Proceedings of OOPSLA (2013). https://doi.org/10.1145/2509136.2509536
18. Smeding, G.J.: An executable operational semantics for python. Universiteit Utrecht (2009)