Static Type Analysis by Abstract Interpretation of Python Programs (Artifact)

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

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

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

Abstract

Python is an increasingly popular dynamic programming language, particularly used in the scientific community and well-known for its powerful and permissive high-level syntax. Our work aims at detecting statically and automatically type errors. As these type errors are exceptions that can be caught later on, we precisely track all exceptions (raised or caught). We designed a static analysis by abstract interpretation able to infer the possible types of variables, taking into account the full control-flow. It handles both typing paradigms used in Python, nominal and structural, supports Python's object model, introspection operators allowing dynamic type testing, dynamic attribute addition, as well as exception handling. We present a flow- and context-sensitive analysis with special domains to support containers (such as lists) and infer type equalities (allowing it to express parametric polymorphism). The analysis is soundly derived by abstract interpretation from a concrete semantics of Python developed by Fromherz et al. Our analysis is designed in a modular way as a set of domains abstracting a concrete collecting semantics. It has been implemented into the MOPSA analysis framework, and leverages external type annotations from the Typeshed project to support the vast standard library. We show that it scales to benchmarks a few thousand lines long, and preliminary results show it is able to analyze a small real-life command-line utility called PathPicker. Compared to previous work, it is sound, while it keeps similar efficiency and precision.

2012 ACM Subject Classification Theory of computation → Program analysis; Software and its engineering → Semantics

Keywords and phrases Formal Methods, Static Analysis, Abstract Interpretation, Type Analysis, Dynamic Programming Language, Python Semantics

Digital Object Identifier 10.4230/DARTS.6.2.11

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

Related Article Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné, "Static Type Analysis by Abstract Interpretation of Python Programs", in 34th European Conference on Object-Oriented Programming (ECOOP 2020), LIPIcs, Vol. 166, pp. 17:1–17:29, 2020.
`https://doi.org/10.4230/LIPIcs.ECOOP.2020.17`

Related Conference 34th European Conference on Object-Oriented Programming (ECOOP 2020), November 15–17, 2020, Berlin, Germany (Virtual Conference)

1 Scope

This is the artifact accompanying the research paper "Static Type Analysis by Abstract Interpretation of Python Programs" published in ECOOP 2020. It allows the reproduction of the table presented in the experimental evaluation of the paper, Table 16.

2 Content

This artifact consists in a virtualbox image containing:

- the static analyzer presented in the paper, based on the Mopsa static analysis framework,
- as well as the other analyzers we compare with (a tool by Fritz and Hage, Pytype, Typpete, a tool by Fromherz et al., RPython),
- the programs used in the experimental evaluation of the paper,
- a Python script helping to reproduce the main table presented in the experimental evaluation of the paper (Table 16).

All analyzers have been compiled and installed; they are ready to be run.

The username and the password of the virtual machine are `ecoop20`, but autologin is enabled. The default keyboard is `en_UK`. In the virtual machine, the most important files are located in `~/Desktop/mopsa/`:

- the source code of the analyzer is in `analyzer`, described in Appendix A.2
- the benchmark files are located in `ecoop20_benchmarks` (Appendix C).
- **to quickly check the artifact**, we recommend launching the following command: `cd ~/Desktop/mopsa/ && python3 ecoop20.py`. This script allows a reproduction of the performance results described in Table 16 of the research paper (see Appendix C.1). The analysis times are displayed progressively, as soon as they are established. It takes the script around 42 minutes to complete, within the virtual machine, on an Intel Core i7-8650U.

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS).

4 Tested platforms

Alongside this document, the artifact consists in a virtualbox image, which has been tested using virtualbox 6.0 on an Ubuntu 19.10. We recommend allocating at least 8GB of RAM to the virtual machine; there is no need for a multi-core CPU.

5 License

The virtual machine image contains software using different licenses:

- Mopsa and the tool from Fromherz et al. are licensed under the GNU LGPL v3,
- Pytype and RPython are under MIT license,
- Typpete uses a Mozilla license,
- the tool from Fritz and Hage uses a BSD License 2.0.

The full licenses are provided within the virtual machine.

6 MD5 sum of the artifact

c275ca7c1de3b346fe30bf52625a818b ECOOP20_Artifact_Monat_Ouadjaout_Mine.ova

7 Size of the artifact

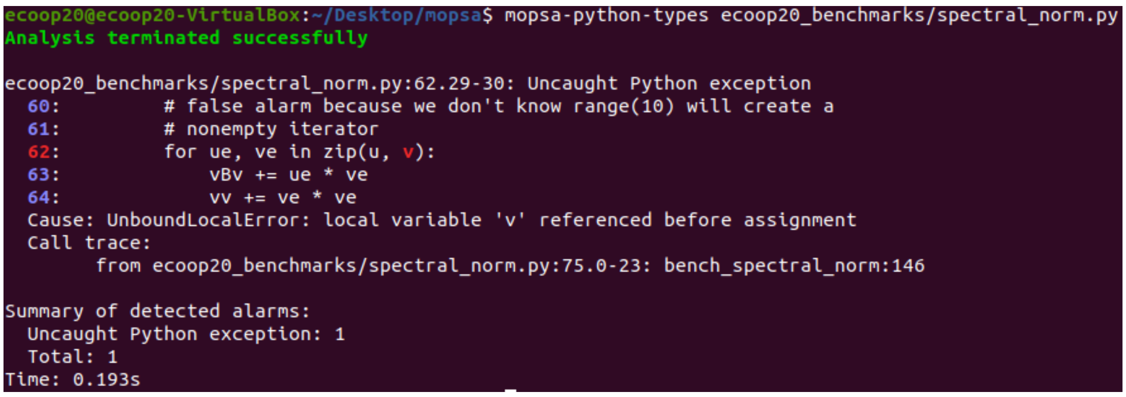
6.1GB

A About Mopsa

A.1 Running the Analyzer

Basic Usage. You can run the analyzer using the command `mopsa-python-types` (which corresponds to the configuration 2 defined in the research paper), and providing the file you want to analyze as argument. A slower version (configuration 1) is available as `mopsa-python-types-conf1`.

▶ Example. If we want to run the analyzer on the `spectral_norm.py` benchmark, you can run `mopsa-python-types ecoop20_benchmarks/spectral_norm.py` from the `mopsa` root directory. The results are provided in Fig. 1. If the analyzer detects any alarm, it will report them. In this case, Mopsa detects a potential `UnboundLocalError` line 62, and highlights the location from where this exception originates. Here, the `UnboundLocalError` is a false alarm, meaning it cannot happen in practice.



```
ecoop20@ecoop20-VirtualBox:~/Desktop/mopsa$ mopsa-python-types ecoop20_benchmarks/spectral_norm.py
Analysis terminated successfully

ecoop20_benchmarks/spectral_norm.py:62.29-30: Uncaught Python exception
  60:         # false alarm because we don't know range(10) will create a
  61:         # nonempty iterator
  62:         for ue, ve in zip(u, v):
  63:             vBv += ue * ve
  64:             vv += ve * ve
  Cause: UnboundLocalError: local variable 'v' referenced before assignment
  Call trace:
        from ecoop20_benchmarks/spectral_norm.py:75.0-23: bench_spectral_norm:146

Summary of detected alarms:
  Uncaught Python exception: 1
  Total: 1
Time: 0.193s
```

Figure 1 Running mopsa on `spectral_norm.py`.

Other options. If you want to get the abstract state at the end of the analysis, you can use the `-lflow` option.

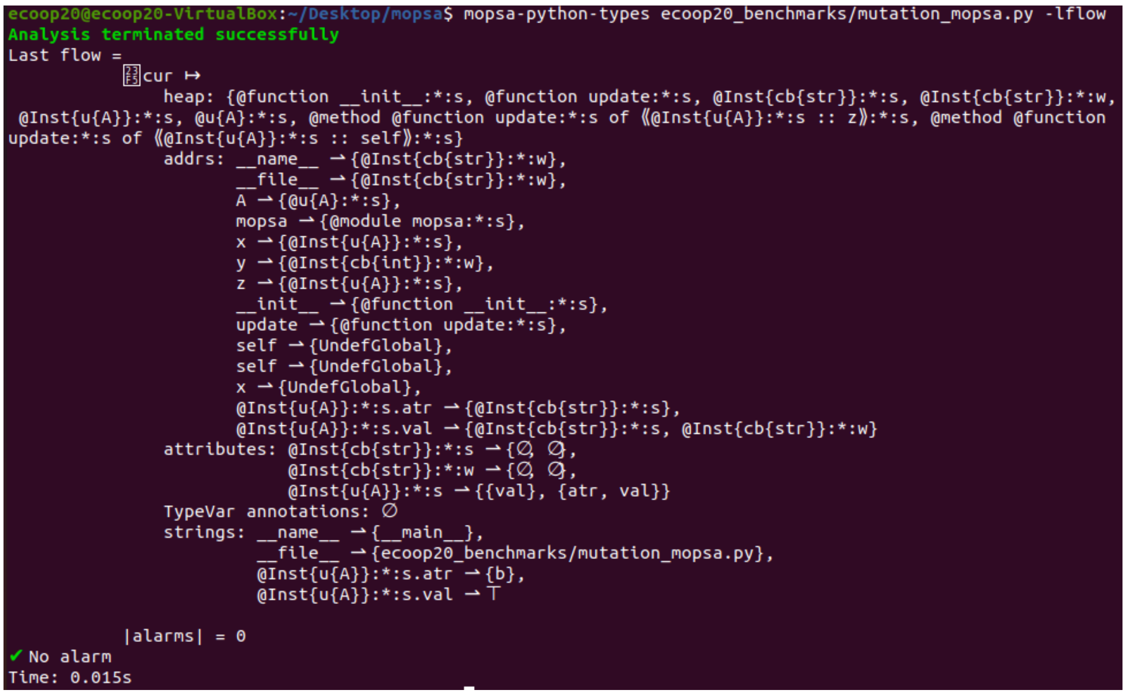
▶ Example. Running the analyzer on `ecoop20_benchmarks/mutation_mopsa.py` yields the result provided in Fig. 2 (this is the running example shown in Figures 3, 4 and 10 of the research paper). The abstract state consists in a mapping from flow tokens $\mathcal{F}$ to the set of abstract addresses currently allocated (`heap`), the abstract environment (`addrs`), and the abstract heap (`attributes`). In our case, we notice in `addrs` that both x and y point to the same strong address, which is an instance of the class `A`. In `attributes`, we see that the analysis inferred that `val` is an attribute always defined, and that `atr` may be undefined.

If you want to see all evaluations done by the analyzer, you can add the `-hook logs` option to the analyzer call. You can also specify `-hook logs -short-logs` to see all evaluations, but without any abstract state displayed. Using `-interactive`, triggers a step by step analysis, in a debugger-like fashion (the commands related to the interactive mode are displayed by typing `h`).

A.2 Structure of the Analyzer

The architecture of Mopsa is described in details in [5]. We give a brief overview of the file organization for the curious reader. The source code of the analyzer is in `analyzer/src`:

- The core of the analyzer is defined in `framework`.
- The analysis of C programs is described in `lang/c`, but is out of the scope of this artifact.



```
ecoop20@ecoop20-VirtualBox:~/Desktop/mopsa$ mopsa-python-types ecoop20_benchmarks/mutation_mopsa.py -lflow
Analysis terminated successfully
Last flow =
           ⏵cur ↦
               heap: {@function __init__:*:s, @function update:*:s, @Inst{cb{str}}:*:s, @Inst{cb{str}}:*:w,
 @Inst{u{A}}:*:s, @u{A}:*:s, @method @function update:*:s of ⟪@Inst{u{A}}:*:s :: z⟫:*:s, @method @function
update:*:s of ⟪@Inst{u{A}}:*:s :: self⟫:*:s}
               addrs: __name__ ⇀{@Inst{cb{str}}:*:w},
                      __file__ ⇀{@Inst{cb{str}}:*:w},
                      A ⇀{@u{A}:*:s},
                      mopsa ⇀{@module mopsa:*:s},
                      x ⇀{@Inst{u{A}}:*:s},
                      y ⇀{@Inst{cb{int}}:*:w},
                      z ⇀{@Inst{u{A}}:*:s},
                      __init__ ⇀{@function __init__:*:s},
                      update ⇀{@function update:*:s},
                      self ⇀{UndefGlobal},
                      self ⇀{UndefGlobal},
                      x ⇀{UndefGlobal},
                      @Inst{u{A}}:*:s.atr ⇀{@Inst{cb{str}}:*:s},
                      @Inst{u{A}}:*:s.val ⇀{@Inst{cb{str}}:*:s, @Inst{cb{str}}:*:w}
               attributes: @Inst{cb{str}}:*:s ⇀{∅, ∅},
                           @Inst{cb{str}}:*:w ⇀{∅, ∅},
                           @Inst{u{A}}:*:s ⇀{{val}, {atr, val}}
               TypeVar annotations: ∅
               strings: __name__ ⇀{__main__},
                        __file__ ⇀{ecoop20_benchmarks/mutation_mopsa.py},
                        @Inst{u{A}}:*:s.atr ⇀{b},
                        @Inst{u{A}}:*:s.val ⇀⊤

           |alarms| = 0
✓ No alarm
Time: 0.015s
```

Figure 2 Displaying the last abstract state of `mutation_mopsa.py`.

- The `lang/universal` folder describes analyses common to the C and Python analyzers, such as interprocedural inlining, loop invariant inference, and the analysis of some intraprocedural constructs.
- The `lang/python` folder of the analysis consists in different files and directories:
  - `ast`, `ast_compare`, `pp` and `visitor` define the AST of Python used, as well as comparison operator on this ast, pretty printers and visitors.
  - the entry point of the Python analysis is `program` (though `frontend` is called before to perform the parsing).
  - `desugar` performs dynamic rewriting of Python expressions into other expressions (Python or universal ones), letting other domains handle them afterwards.
  - the `data_model` defines the semantics of Python for most operators:
    - arithmetic operators (`+`, `-`, ...)
    - comparison operators (`==`, `is`, `<=`, ...)
    - augmented assignments (`+=`, ...)
    - attribute accesses (`x.attr`)
    - calls (`obj()`)
    - subscript (`a[b]`)
  - the semantics of the `data_model` is supplemented by the semantics described in `objects`. For example, performing an attribute access `x.attr` usually calls `object.__getattribute__(x, attr)`, which is in `objects/py_object.ml` (and described in Fig. 6 of the research paper).
  - parts of some libraries are directly defined in OCaml, and found in folder `libs`
  - `flows/exn.ml` describes the handling of exceptions in Python, which is a special feature in the control-flow analysis.
  - the type analysis is described in `types`:
    - `addr_env` is an abstract environment, mapping variables to sets of addresses
    - `nominal_types` handles analysis of some operators such as `isinstance`.

- structural_types keeps the potential attributes of each addresses, and the analysis of low level attribute accesses used by data_model/attribute and objects/py_object.ml for example.
- type_annot handles type annotations potentially given as stubs in files share/mopsa/stubs/python/typedshed/.

All binaries and scripts to launch the analyzer are in bin. share/mopsa/ contains json configurations of different analyses, as well as stubs (which consists in type annotations for the Python part).

B Running the other analyzers

B.1 Fritz & Hage

The tool from Fritz and Hage [2] dates from 2011, and uses GHC 7.0.3 and cabal 1.14.0. We were unable to install it using more recent versions of Haskell, so we embedded a docker container into the virtual machine. Going back to the analysis of `spectral_norm.py`, you can launch the analyzer using the command `infer-python-types spectral_norm.py`. The script `infer-python-types` is located in `~/.local/bin`. The output of the analyzer consists in the type of each variable at each program point.

B.2 Pytype

Pytype [6] is run using the following command: `pytype -n spectral_norm.py`. The `-n` option disables the cache. The output of Pytype consists in the errors it has detected. It also provides the results of its inference as annotations in a directory given by `ninja` during the analysis (and then in the `pyi` directory).

B.3 Typpete

Typpete [4] is called using `typpete isinstance.py`. Similarly to Pytype, the inference results are provided as an annotated pyi file, in the generated `inference_output` directory. Upon errors, the analyzer will print `Unsat` in the output and provide a short explanation.

B.4 Value Analysis of Fromherz et al.

The analyzer from Fromherz et al. [3] is based on an older version of the Mopsa framework. You can call it using `~/nfm18-analyzer/bin/nfm18 isinstance.py`. Detected exceptions will be displayed, with their location in the file.

B.5 RPython

RPython [1] is called using `rpython -annotate -rtype isinstance.py`. It may reject some benchmarks as its goal is to optimize the performance of a subset of Python programs. It needs some special wrapping around programs, so the files analyzed are in a separate directory: `ecoop20_benchmarks/rpython`.

C Reproducing the experimental evaluation

This section explains how to reproduce the results established in Section 6.3 of the research paper.

C.1 Performance Evaluation

When your current working directory is `~/Desktop/mopsa`, you can run `python3 ecoop20.py`. This will output a table similar to Table 16 of the research paper. It takes the script around 42 minutes to complete, within the virtual machine, on an Intel Core i7-8650U. The timeout duration of one analysis can be defined at the beginning of the script (by default set to 15 minutes), as well as the number of times each analysis is run (by default it is set to 1, but you can switch to 10 to have a bit less variation on the lower running times).

C.2 Soundness Evaluation

Within the `ko` directory of `ecoop20_benchmarks`, you will find erroneous variants of the benchmarks, along with an explanatory markdown file for each file. In all erroneous variant, a simple bug has been added, where an integer is replaced by a string.

References

1 Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. Rpython: a step towards reconciling dynamically and statically typed OO languages. In *DLS*, pages 53–64. ACM, 2007.

2 Levin Fritz and Jurriaan Hage. Cost versus precision for approximate typing for Python. In *PEPM*, pages 89–98. ACM, 2017.

3 Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. Static value analysis of Python programs by abstract interpretation. In *NFM*, volume 10811 of *LNCS*, pages 185–202. Springer, 2018.

4 Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. MaxSMT-based type inference for Python 3. In *CAV (2)*, volume 10982 of *LNCS*, pages 12–19. Springer, 2018.

5 M. Journault, A. Miné, R. Monat, and A. Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyzer. In *Proc. of the 11th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE19)*, pages 1–17, July 2019.

6 Pytype. `https://github.com/google/pytype`, 2019. Accessed: 2019-10-22.