Specification of Python Math and Data Science Libraries

Pascal Devenoge
Supervised by Dr. Marco Eilers, Prof. Dr. Peter Müller
ETH Zurich
Zürich, Switzerland

December 2023

1 Introduction

The Python language forms part of the backbone of today's data science, machine learning and related fields. Its popularity is largely built on the wide availability of various data science and mathematical libraries such as NumPy, PyTorch, etc.

Due to its dynamically-typed nature, Python provides a trade-off of a lot of flexibility and minimal annotation overhead for the programmer since no types need to be specified in code, at the cost of not being able to rule out classes of simple errors that could be found by type checking.

A more recent development is the addition of optional type hints, which can be used by external tools such as mypy [myp] to statically type check Python code. A large variety of popular libraries now provide such type hints, allowing client code to be type checked. However, interfaces provided by these libraries can come with various usage rules and requirements that the client code must follow. Type checking is not sufficient in most cases to ensure that client code abides by a library's API requirements. Static verification of Python code can help fill some of these gaps, and allows for more of a library's interface requirements to be checked ahead of execution, ruling out more usage errors than what is possible through type checking.

2 Background

2.1 Viper

Viper [MSS16] is a formal verification ecosystem designed at ETH Zurich. It is centered around a human readable intermediate language used to specify sequential and concurrent programs, using a separation logic [Rey02] approach. It provides support for reasoning about a program's heap state using an explicit permission based approach. In order to verify code written in a variety of programming languages, the Viper ecosystem includes a number of front-end verifiers, translating code and specifications written in an input language such as Java, Rust, Go, etc. to the Viper intermediate language, before handing off the generated Viper code to an underlying verification tool. Figure 1 gives an overview of the Viper infrastructures structure.

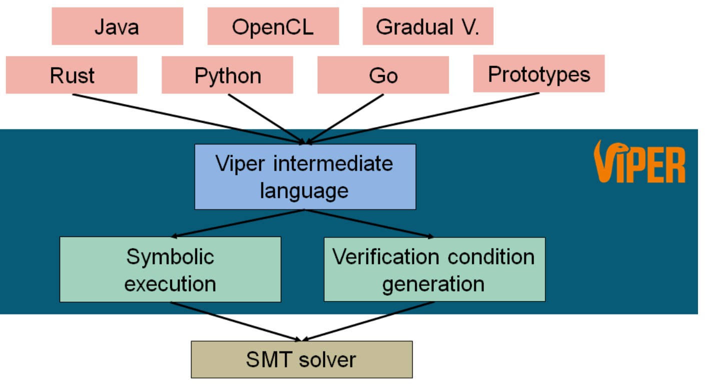

Figure 1: Viper ecosystem overview

2.2 mypy

Along with the introduction of optional type hints to the Python language, a number of static type checking tools have been developed (see Listing 1 for an example of type annotations in Python code). One of the most well-known of these tools is the mypy typechecker.

```
from typing import List

def find_largest(list : List[int]) -> int:
    candidate : int = list[0]
    for n in list:
        if n > candidate:
            candidate = n
    return candidate
```

Listing 1: Example of Python code with type annotations

Since client code might depend on a large set of direct and transitive dependencies, type checking it together with all the used library implementations would be impractical. To simplify the process, a library can define stub files. These files contain type annotated definitions of the libraries interface (function, class prototypes, etc.) without any of the interface's implementation code.

mypy relies on the type information in these stub files to check client code. Using this mechanism, Python packages can remain modularized and rely on information hiding at the library level from the point of view of the type checker.

2.3 Nagini Verifier

The Nagini verifier for Python [EM18] is a front-end for the Viper verification infrastructure, allowing for static verification of Python code using a similar permission based reasoning approach as the underlying Viper verifier. In Nagini, Python code is annotated with specifications of logical assertions, pre- and post-conditions about the program's state and permissions to access this state. The following listing shows a simple example of a functions specification in Nagini.

```
from typing import List
from nagini_contracts.contracts import *

def find_largest(list : List[int]) -> int:
    Requires(Acc(list_pred(list), 1/2))
    Requires(len(list) > 0)

    Ensures(Acc(list_pred(list), 1/2))
    Ensures(Exists(list, lambda n: n == Result()))
    Ensures(Forall(list, lambda n: n >= Result()))
    ...
```

Listing 2: Example of a function stub with Nagini assertion annotations

The Python code is statically verified to always satisfy the provided specification, using the Viper infrastructure as its back-end. Using this approach, Nagini is designed with the explicit purpose of allowing complete functional correctness verification. However, writing full functional specifications and verifying them is challenging and often infeasible for programs performing complex mathematics, due to limitations in the underlying tools' ability to handle non linear arithmetic and complex language semantics (e.g., floating point numbers).

A different approach is to leverage Nagini's support for mypy stub files. Stub files, normally used for type annotations, can be augmented with Nagini specifications, which are used in the verification of client code. This way, Nagini can be used to specify safety properties and usage rules of a Python library's API, and client code can be verified to uphold these rules statically, while maintaining abstraction and information hiding of internal implementation details.

3 Goals

The goal of this thesis is to evaluate the potential of the Nagini verifier for formal specification of interface usage rules and basic safety properties of widely-used Python libraries, and checking of whether client code meets those rules, while minimizing the annotation overhead for the client code author.

3.1 Core goals

1. **Select a subset of the NumPy library's interface to write specifications for**
   NumPy [Num], being a very widely used foundational library represents a good target to test static verification of API rules against. A subset of NumPy's API must be selected to write interface specifications for, based on whether they have usage rules that are feasible and interesting to verify.

2. **Identify common patterns, usage errors and properties important to users of NumPy**
   Safety and usage rules of the selected API subset that can be statically verified without much annotation needed in the client code should be identified, based on common usage patterns and problems faced by users. Verification involving complex API semantics and functional correctness of client code is outside of the scope of the thesis.

3. **Specify library interfaces to verify correct usage**
   Specifications for the selected interface subset of NumPy should be written to detect identified usage problems and API rules violations. The focus is on minimizing the annotation overhead for the client code to be verified against the specifications and to verify interface rules in as much generality as possible.

4. **Extend the verifier with an additional mode to reduce annotation overhead for the user**
   An additional mode should be added to the Nagini verifier to further reduce the annotation overhead posed to a potential client code author, at the cost of expressiveness, modularity or performance. In particular, this new mode should utilise inlining for verification of function calls and integrate an existing technique for reducing annotation overhead of loops (e.g., k-induction) that is already available in the Viper back-end used by Nagini.

5. **Evaluate the effectiveness of static verification for API rules checking**
   The results of the specification process should be evaluated to see how effective static verification is to prevent different types of library usage errors. How much annotation overhead for the user is absolutely necessary to verify correct usage should be determined.

3.2 Extension Goals

1. **Design API specifications for other libraries beyond NumPy**
   To better study the feasibility of static verification of client code's correct use of a library's interface with Nagini and better identify limitations and usability problems, additional libraries can be picked for specification to include a wider variety of interface designs to be tested.

2. **Implement support for additional Python language features**
   Additional Python constructs and features are introduced with each version of the language. Support for some of the missing language constructs that are frequently used in NumPy client code could be added to Nagini to better support newer Python versions.

References

[Rey02] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 2002, pp. 55–74. DOI: `10.1109/LICS.2002.1029817`. URL: `https://doi.org/10.1109/LICS.2002.1029817`.

[MSS16] P. Müller, M. Schwerhoff, and A. J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by B. Jobstmann and K. R. M. Leino. Vol. 9583. LNCS. Springer-Verlag, 2016, pp. 41–62. URL: `https://doi.org/10.1007/978-3-662-49122-5_2`.

[EM18] Marco Eilers and Peter Müller. "Nagini: A Static Verifier for Python". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 596–603. DOI: `10.1007/978-3-319-96145-3\_33`. URL: `https://doi.org/10.1007/978-3-319-96145-3%5C_33`.

[myp] mypy-project. *mypy - Optional Static Typing for Python*. URL: `https://www.mypy-lang.org` (visited on 12/10/2023).

[Num] NumPy-team. *NumPy - The fundamental package for scientific computing with Python*. URL: `https://numpy.org/` (visited on 12/15/2023).