Enabling Object Equality Reasoning for Python

Bachelor Thesis Description

Micha Greutmann
Supervised by Dr. Marco Eilers, Prof. Dr. Peter Müller
ETH Zürich

November 2024

1 Introduction

Python is currently one of the most popular programming languages and fundamental in many research areas such as data science due to its ease of use, flexibility and vast number of libraries.

It is dynamically-typed, therefore, type safety is checked at runtime. By using the static type checker mypy [1] and Python type hints defined in PEP 484 [2], the user is enabled to check type safety statically, comparable to the experience with statically typed programming languages such as C.

2 Background

In this section, various background concepts are introduced that will be used to achieve the goals of the thesis (see Section 3).

2.1 Nagini

Nagini [3] is an automated, modular verifier, which leverages the mypy type information to statically verify a rich subset of Python programs. It functions as a front-end to the Viper verification infrastructure [4], which uses a variation of separation logic [5] called *implicit dynamic frames* [6] (see Section 2.4) and SMT-solvers (see Figure 1) for verification.

The Python source code is encoded to the *intermediate verification language* (IVL) Viper by Nagini. Since Viper is a simple, imperative language lacking many features of the object-oriented Python language, Nagini must ensure to encode the source code in a sound way. In the example in Figure 2, behavioral subtyping checks [8] for the class `X` and its subclass `SubX` are encoded, i.e., overrides must satisfy the specification of the supertype method. Nagini enforces this for all overrides. Thus, in `SubX.bar()` the the precondition can only be weakened (or maintained) and its postcondition can only be strengthened (or maintained).

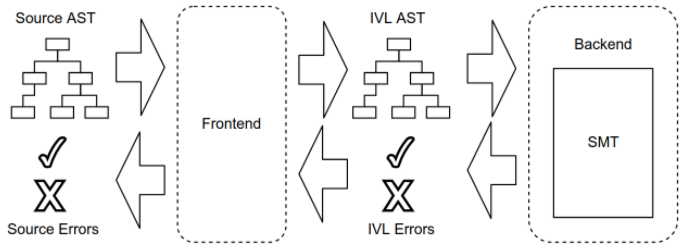

Figure 1: The architecture of the Viper verification instrastructure. The figure is taken from page 17 of [7].

```
class X:
    def bar(self, i: int) -> int:
        # requires P
        # ensures Q
        ...

class SubX(X):
    def bar(self, i: int) -> int:
        # requires P'
        # ensures Q'
         ...
```

```
method SubX_bar_override_check(self: Ref, i: Ref)
returns (res: Ref)
    requires issubtype(typeof(self), SubX)
    requires issubtype(typeof(i), int)
    requires P
    ensures Q
{
    res := SubX_bar(self, i)
}
```

Figure 2: Encoding the behavioral subtyping check for the class `X` and its subclass `SubX` to the IVL Viper. The example is taken from page 39 of [7].

```python
# requires len(nums > 0)
def minimum(nums: List[int]) -> int:
    cur_min: int = nums[0]
    for num in nums[1:]:
        if num < cur_min:
            cur_min = num
    return cur_min
```

Figure 3: Implementation of the `minimum` function, which computes the minimal integer `cur_min` in a given list of integers `nums`.

2.2 Pure Functions

Pure functions are deterministic and side-effect free, i.e., perform a certain task without modifying any non-local state. Thus, such functions can be used in specifications. In the example in Figure 3 the function `minimum` computes the minimal integer of the list `nums`. Since `minimum` only reads the values of `nums` and only modifies the local variable `cur_min`, it is pure.

2.3 Dunder Methods

Dunder (short for *double underscore*) methods are special functions which define operators (e.g., `==, +, >`), containment checks (e.g., `x in some_list`), assignments (e.g., `x.f = 5`) etc. for a specific type of object. These methods can be overridden to change the default functionality.

In the example in Figure 4 the type `IntVec` is defined; it represents an integer vector $x \in \mathbb{Z}^n$ for some $n \in \mathbb{N}$. Ordinarily, the `__eq__` method is defined as *reference equality*, i.e., it is `True` if and only if both references point to the same object of type `IntVec`. But `__eq__` and `__add__` have been overridden to change `==` and `+` respectively.

The `==` operator now compares the vector elements component-wise instead of using reference equality. Thus, the statement `IntVec([1,2]) == IntVec([1,2])` returns `True`. The `+` operator now adds `other` component-wise to `self` instead of being undefined. Thus, the statement `IntVec([1,2,3]) + IntVec([6,5,4])` sets `self` (`IntVec([1,2,3])`) to have the same value as `IntVec([7,7,7])`.

2.4 Implicit Dynamic Frames

In *implicit dynamic frames*, the assertion $P \hat{=}$ `acc(x.f) * acc(y.f)` defines the permission to access the field `f` of the two objects `x` and `y`. Since the *separating conjunction* ($*$) is used in P, `x` and `y` cannot reference the same object, i.e., point to different heap locations, otherwise P is unsatisfiable.

```python
class IntVec:
    def __init__(self, nums: List[int]) -> None:
        self.vec: List[int] = nums

    # requires same dimensions
    def __eq__(self, other) -> bool:
        for i in range(len(self.vec)):
            if self.vec[i] != other.vec[i]:
                return False
        return True

    # requires same dimensions
    def __add__(self, other) -> None:
        for i in range(len(self.vec)):
            self.vec[i] += other.vec[i]
```

Figure 4: A class definition of custom type `IntVec`, which overrides multiple dunder methods.

2.5 Predicate Families

Predicate families [9] are used to reason about objects. In typical Python programs, subclasses consist of the same fields and methods as their superclasses with a few additional fields and/or methods as in the example in Figure 5. A method `square` is defined for both classes and squares all available fields of the object instance for which it needs write access to `self.x` and `self.x and self.y`, i.e., needs the permissions `acc(self.x)` and `acc(self.x) * acc(self.y)` respectively. Thus, since `acc(self.x)` $\not\models$ `acc(self.x) * acc(self.y)`, `SubX` is not a behavioral subtype according to the above definition (see Section 2.1) and Nagini reports an error.

```python
class X:
    def __init__(self):
        self.x: int = 0

    def square(self) -> None:
        Requires(Acc(self.x))
        Ensures(Acc(self.x))
        self.x *= self.x

class SubX(X):
    def __init__(self) -> None:
        self.x: int = 0
        self.y: int = 2

    def square(self) -> None:
        Requires(Acc(self.x) and Acc(self.y))
        Ensures(Acc(self.x) and Acc(self.y))
        self.x *= self.x
        self.y *= self.y
```

Figure 5: Custom class `X` and its subclass `SubX` implement a `square` method and use implicit dynamic frames (see Section 2.4) to verify their access.

To address this common case, Nagini supports the concepts of predicate families, i.e., predicates that can be redefined in subclasses. We extend the two classes with with a predicate family `started()` to model the access to all available fields (see Figure 6). The permissions are replaced with the defined predicate in the `square` methods to satisfy behavioral subtyping. Now the preconditions of both methods are identical and Nagini accepts the program.

```python
class X:
    def __init__(self):
        self.x: int = 0

    def square(self) -> None:
        Requires(self.started())
        Ensures(self.started())

        Unfold(self.started())
        self.x *= self.x
        Fold(self.started())

    @Predicate
    def started(self) -> bool:
        return Acc(self.x)

class SubX(X):
    def __init__(self) -> None:
        self.x: int = 0
        self.y: int = 2

    def square(self) -> None:
        Requires(self.started())
        Ensures(self.started())

        Unfold(self.started())
        self.x *= self.x
        self.y *= self.y
        Fold(self.started())

    @Predicate
    def started(self) -> bool:
        return Acc(self.y)
```

Figure 6: The same two classes from Figure 5, but with the added predicate family `started()`. The permission `Acc(self.x)` is automatically included in `SubX.started()`, since constraints in Nagini can only be extended (and not completely redefined).

We can use it in a function `squareX`, which calls `obj.square` (see Figure 7). If `obj` has type `X`, the function squares the field `obj.x`. If `obj` has type `SubX`, `squareX` squares the fields `x` and `y`.

```python
def squareX(obj: X) -> None:
    Requires(obj.started())
    Ensures(obj.started())
    obj.square()
```

Figure 7: `squareX` method, which takes an `obj` of type `X` or any subtype, e.g., `SubX`.

3 Goals

Nagini currently allows overriding of impure functions and implements behavioral subtype checking for them. For pure functions, however, this is not yet possible. Furthermore, Nagini currently does not have a principled way to support object equality statements in its contracts. The same thing holds for containment checks for collections, since they depend on object equality.

3.1 Core Goals

1. Allow overriding pure functions in Nagini and to encode them into Viper to enable their modular verification.
   (a) Allow pure function overrides in subclasses.
   (b) Support modular pure function calls such that the caller learns only the specification, not the implementation.
   (c) Implement behavioral subtyping checks for pure functions.
   (d) Define and implement an encoding of overridden pure functions into Viper.
2. Enable verification in Nagini for Python programs that contain expressions like `obj1 == obj2` in pre- and/or postconditions. The integration should be done for built-in and custom classes, for which the equality operator == can be overridden (see Section 2.3).
   (a) Define a contract for `object.__eq__` using predicate families (see Section 2.5) that represent state for possibly mutable objects. Since primitives, such as `int`s, are stateless objects, the overhead of folding and unfolding ought to be minimized.
   (b) Adapt the `object.__eq__` definitions of built-in data types to work with the new system. Additionally, check the contract to be consistent with `object.__hash__` and the reflexive and transitive properties.
3. Integrate the contract of `object.__eq__` with built-in data structures, e.g., `list`s.

(a) Containment checks (e.g., `x in some_list`) implicitly use the equality operator to compare each element `y` in the list `some_list` with the object `x`.
Adapt the `__contains__` definitions of built-in container types to use `object.__eq__` in the natural way using an existential quantifier.

4. Evaluate the integrations from (2) and (3).

   (a) Test the completeness, performance, and usability of the integrations of `object.__eq__` and `object.__contains__` in real-world code examples.

3.2 Extension Goals

1. Adapt the definition of `__contains__` for built-in container types from (3) to avoid the existential quantifier if this definition is incomplete or leads to bad performance in real-world Python programs.
2. Define and implement contracts for other pure dunder methods.

References

[1] Jukka Lehtosalo et al. *Mypy - Optional Static Typing for Python*. 2017. URL: http://mypy-lang.org (visited on 2024-05-11).

[2] Jukka Lehtosalo Guido van Rossum and Łukasz Langa. *PEP 484: Type Hints*. 2014. URL: https://www.python.org/dev/peps/pep-%200484/ (visited on 2024-05-11).

[3] 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. Cham: Springer International Publishing, 2018, pp. 596–603. ISBN: 978-3-319-96145-3. DOI: 10.1007/978-3-319-96145-3_33. URL: http://doi.org/10.1007/978-3-319-96145-3_33.

[4] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62. ISBN: 978-3-662-49122-5. DOI: 10.1007/978-3-662-49122-5_2. URL: http://doi.org/10.1007/978-3-662-49122-5_2.

[5] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In: *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*. Ed. by Laurent Fribourg. Vol. 2142. Lecture Notes in Computer Science. Springer, 2001, pp. 1–19. DOI: 10.1007/3-540-44802-0_1. URL: https://doi.org/10.1007/3-540-44802-0%5C_1.

[6] Jan Smans, Bart Jacobs, and Frank Piessens. "Implicit dynamic frames". In: *ACM Trans. Program. Lang. Syst.* 34.1 (2012), 2:1–2:58. DOI: 10.1145/2160910.2160911. URL: https://doi.org/10.1145/2160910.2160911.

[7] Marco Eilers. "Modular Specification and Verification of Security Properties for Mainstream Languages". en. Doctoral Thesis. Zurich: ETH Zurich, 2022. DOI: 10.3929/ethz-b-000580641. URL: http://doi.org/20.500.11850/580641.

[8] Barbara Liskov and Jeannette M. Wing. "A Behavioral Notion of Subtyping". In: *ACM Trans. Program. Lang. Syst.* 16.6 (1994), pp. 1811–1841. DOI: 10.1145/197320.197383. URL: https://doi.org/10.1145/197320.197383.

[9] Matthew J. Parkinson and Gavin M. Bierman. "Separation logic and abstraction". In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. Ed. by Jens Palsberg and Martín Abadi. ACM, 2005, pp. 247–258. DOI: 10.1145/1040305.1040326. URL: https://doi.org/10.1145/1040305.1040326.