Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Enabling Object Equality Reasoning for Python

Bachelor's Thesis

Micha Greutmann

May 25, 2025

Supervised by Dr. Marco Eilers, Prof. Dr. Peter Müller

Programming Methodology Group
Institute for Programming Languages and Systems
Department of Computer Science, ETH Zürich

Abstract

User-defined equality functions are fundamental in object-oriented programming languages like Python. Verifying their correctness, however, is non-trivial due to subtyping, function overriding, and the limited support in existing verifiers.

This thesis proposes a modular framework for specifying and verifying that user-defined equality functions satisfy the properties of an equivalence relation, i.e., reflexivity, symmetry, and transitivity. In particular, we model equality function applications in a way that the results remain consistent when objects are cast to more specific subtypes. We integrate our implementation into Nagini, an automated, modular verifier for Python programs based on the Viper verification infrastructure.

Since the framework is based on a permission logic, it extends to concurrent settings. Our evaluation demonstrates that user-defined equality functions can be verified in a reasonable amount of time and with an acceptable annotation and specification overhead for the user.

This work establishes a foundation to enable modular reasoning about user-defined equality functions in the context of object-oriented programming languages.

Acknowledgments

I wish to acknowledge the invaluable assistance and encouragement of the people who made this work possible.

I would like to sincerely thank Prof. Dr. Peter Müller for the opportunity to work on this thesis and explore this research topic.

I would like to express my deepest gratitude to my supervisor, Dr. Marco Eilers, for his exceptional guidance throughout this semester. He devoted many hours to brainstorming with me, answering my countless questions, and providing thoughtful feedback. His unmatched patience and passion for the subject were a great source of inspiration to me.

Last but not least, I am deeply grateful to my family and friends for their unwavering support, constant encouragement, and belief in me.

Contents

Chapter 1

Introduction

Equality is a fundamental concept that is essential across nearly every area of computer science. In particular, object-oriented programming languages like Python allow classes to define custom equality methods. However, this added complexity significantly increases the difficulty of ensuring correctness. This includes tasks such as verifying that user-defined equality methods satisfy the properties of an equivalence relation, i.e., reflexivity, symmetry, and transitivity, and establishing that equality results remain consistent when objects are cast to more specific subtypes[1].

Currently, formal verification support for object equality is limited, especially for object-oriented programming languages. One notable approach is that of OpenJML [1], which defines an integer ghost field for each object, whose value is produced by a unique hashing function. Consequently, two objects are equal if and only if their hash values are identical. Although promising in theory, the approach is currently constrained to integers and inherently non-modular, i.e., two independently verified code examples may not verify if they are combined. This is because each integer ghost field must be globally unique, consequently, code examples cannot be verified in isolation, since otherwise two hash values could be identical, even though the objects are not equal. Hence, extending the approach to a modular verification system like Nagini becomes challenging.

In this thesis, we propose a modular approach to verify user-defined equality functions for Python. In particular, the methodology allows using equality in specifications, e.g., a method requires its two arguments to be equal. Additionally, the approach is based on a permission logic and therefore also works in a concurrent setting. Furthermore, we verify that for user-defined equality functions, the properties of an equivalence relation hold,

[1] This sentence was revised using ChatGPT 4o

and establish that equality defined for subtypes is consistent with the equality semantics of their supertypes[2].

In Chapter 3, we propose different approaches to encode pure function overrides. In Chapter 4, we begin by introducing a concept to verify that user-defined equality functions satisfy the properties of an equivalence relation, i.e., reflexivity, symmetry, and transitivity. Next, we present a framework for specifying equality functions that can be overridden. Based on the two approaches from the previous chapter, we model equality function applications with different guarantees for completeness and performance (see Chapter 4). In Chapter 5, we evaluate our solution in terms of performance, usability, and completeness. Finally, in Chapter 6, we conclude with a summary and closing remarks.

[2]This sentence was revised using ChatGPT 4o.

Chapter 2

Preliminaries

In this section, various background concepts are introduced that will be used throughout this thesis.

2.1 Nagini

Nagini [3] is an automated, modular verifier that leverages type information obtained from the Mypy type checker [4] to statically verify a rich subset of Python programs. It functions as a front-end to the Viper verification infrastructure [6], which uses a variation of separation logic [7] called *implicit dynamic frames* [9] (see Section 2.8) and SMT-solvers (see Figure 2.1) for verification.

The Python source code is encoded into 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 encode Python constructs into simpler Viper representations in a sound way.

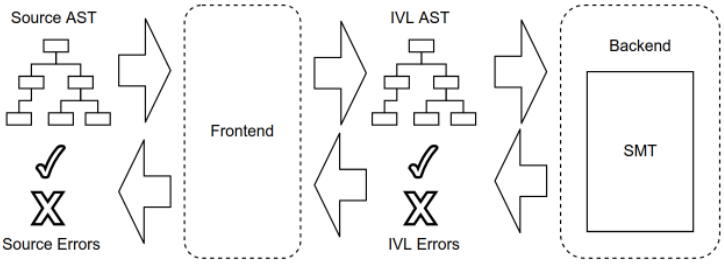

Figure 2.1: The architecture of the Viper verification infrastructure. The figure is taken from page 17 of [2].

```
class X:
    def foo(self, i: int) -> int:
        Requires(i > 0)
        Ensures(Result() > 1)
        return i + 1

class SubX(X):
    def foo(self, i: int) -> int:
        Requires(i >= 0)
        Ensures(Result() >= 5)
        return i * i + 5
```

Figure 2.2: The function `SubX.foo` is a behavioural subtype of `X.foo`.

2.2 Deductive Verification

Nagini and Viper use deductive verification to reason about programs: Each function has a specification that describes the behaviour of a function. Thus, function invocations are modelled using preconditions, which must hold before a call, and postconditions, which are assumed after the function returns. Consequently, deductive verification enables reasoning about essential properties, such as termination or functional correctness, for example, the fact that a user-defined function satisfies the properties of an equivalence relation.

2.3 Behavioural Subtyping

A type satisfies behavioural subtyping [5] if all its functions and methods comply with the specification of the function or method it overrides. In particular, the precondition of the overriding function is weakened (or maintained), and its postcondition is strengthened (or maintained). Nagini ensures that all method overrides comply with this requirement. In the example in Figure 2.2 the method `SubX.foo` is a behavioural subtype of `X.foo`, since the following conditions hold:

$$\begin{aligned} \texttt{i > 0} &\models \texttt{i >= 0} && \text{(Precondition of SubX is weaker)} \\ \texttt{int} &<: \texttt{int} && \text{(for type(Result()) in SubX and X)} \\ \texttt{Result() >= 5} &\models \texttt{Result() > 1} && \text{(Postcondition of SubX is stronger).} \end{aligned}$$

2.4 Pure Functions

Pure functions are deterministic and side-effect free, i.e., evaluate an expression without modifying any non-local state. Thus, such functions can be

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

used in specifications. In the example in Figure 2.3, the function `minimum` computes the minimal integer of the list `nums`. Since `minimum` only reads the values of `nums` and does not modify any non-local data and does not contain any non-determinism, it is pure.

2.5 Constant `NotImplemented`

Python uses the constant `NotImplemented`, which can be returned in equality functions, if a specific comparison is not implemented. If `a == b` is evaluated and the first equality call, e.g., `a.__eq__(b)` returns this constant, then Python also calls `b.__eq__(a)`. If this invocation also returns the constant, then the initial comparison returns `False`. Otherwise, Python just returns the first result. Since we verify symmetry, i.e., `a == b` and `b == a`, we do not require this constant.

2.6 Opaque Functions

Opaque functions hide implementation details, exposing only their specification to a caller, i.e., the function signature and contract, but not the function implementation. If the dynamic type is known, an opaque function `foo` in Viper is invoked using `(@reveal foo(...))`, such that the function definition is assumed after the call, rather than only relying on its postconditions. In contrast, calls to transparent (i.e., non-opaque) functions assume the function implementation by default and therefore cannot be soundly overridden.

2.7 Dunder Methods

In Python, dunder (short for *double underscore*) methods are special functions that 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.

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

In the example in Figure 2.4, the type `IntVec` is defined; it represents an integer vector $x \in \mathbb{Z}^n$ for some $n \in \mathbb{N}$. By default, the `__eq__` method is inherited from the object class and defined as *reference equality*, i.e., it is `True` if and only if both references point to the same object of type `IntVec`. However, `__eq__` and `__add__` are overridden to change `==` and `+`, respectively.

When the equality operator `==` is used, as in the comparison `IntVec([1,2]) == IntVec([1,2])`, the following happens: Since `IntVec` is not a proper subtype of the right-hand operand's type, the left-hand side's `__eq__` function is called. Hence, the vector elements are compared component-wise instead of using reference equality. The invocation returns `True`. A call to the equality function could also evaluate to `False` or the constant `NotImplemented` (see Section 2.5), which signals that the binary operation is not implemented for the given ordering of operands. The `+` operator calls `__add__` in a similar manner and adds `other` component-wise to `self` instead of being undefined. Consequently, the statement `IntVec([1,2,3]) + IntVec([6,5,4])` sets `self` (i.e., `IntVec([1,2,3])`) to have the same value as `IntVec([7,7,7])`.

2.8 Implicit Dynamic Frames

Implicit dynamic frames (IDF) [9] is a logic that enables reasoning about mutable heap data. Only methods or functions that own a permission to a location are allowed to access it. The assertion $P \stackrel{\text{def}}{=} \texttt{acc(x.f) * acc(y.f)}$ defines the permission to access the field `f` of the two objects `x` and `y`. The operator $*$, called *separating conjunction*, is used in P; it is defined as ordinary

```
# assume class A with one integer field i is defined
@Predicate
def mem(a: A) -> bool:
    return Acc(a.i)

def foo(a: A) -> int:
    Requires(mem(a))
    Ensures(mem(a))
    Unfold(mem(a))
    res: int = a.i
    Fold(mem(a))
    return res
```

Figure 2.5: An example of how predicates are used in Nagini.

conjunction, but `x` and `y` cannot reference the same object, i.e., they point to different heap locations. Otherwise, P is unsatisfiable. Hence, P denotes two different permissions. Nagini and Viper use IDF to reason about permissions. In Nagini, `Acc(x.f) and Acc(y.f)` denotes the same permissions as P.

2.9 Predicates

In Nagini, a predicate is a function that returns a boolean value with a body that consists only of a single return statement. It is annotated with the decorator `@Predicate`. Predicates can also include permissions; to acquire or give away the permissions, we unfold and fold the predicate, respectively. Consider the example in Figure 2.5, where the predicate `mem` is defined. We require the predicate in the precondition of the method `foo`. To read or modify the field `a.i`, we use `Unfold(mem(a))` to acquire the permissions. Subsequently, we read the value of `a.i`, save it in the variable `res`, give away the permissions using `Fold(mem(a))`, and return the value of `res`.

2.10 Predicate Families

Predicate families [8] are used to reason about objects in the presence of subtyping. In typical Python programs, subclass instances have the same fields and methods as instances of their superclass with a few additional fields and/or methods, as illustrated in the example in Figure 2.6. For each class, we define a method `square`, which squares all available fields of the object instance. Since the class `SubX` is a subtype of `X`, Nagini checks that behavioural subtyping holds. Since `Acc(self.x) and Acc(self.y)` is not a logical consequence of `Acc(self.x)`, `SubX` is not a behavioural subtype according to the definition in Section 2.3, and Nagini reports an error.

```
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 2.6: The class `X` and its subclass `SubX` implement the method `square`, for which they require access to different heap locations.

To address this common case, Nagini supports the concept of predicate families, i.e., predicates whose meaning depends on the type of their receiver argument. In Nagini, predicates can only be extended and not completely redefined. Hence, if a subclass overrides a predicate of a superclass, then the subclass predicate is defined as the conjunction of the bodies of both predicates. Consider the example in Figure 2.7, where we extend the two classes with a predicate family `started()` to model the access to all available fields. The permissions are replaced with the defined predicate in the `square` methods to satisfy behavioural subtyping. If `self` has type `X`, then `started(self)` returns `Acc(self.x)`. If `self` has type `SubX`, then `started(self)` evaluates to `Acc(self.x) and Acc(self.y)`. In the predicate definition of `SubX.started`, however, we only include `Acc(self.y)`, since predicates can only be extended in Nagini. Now the preconditions of both methods are identical, and Nagini accepts the program.

We can use it in a function `squareX`, which calls `obj.square` (see Figure 2.8). If `obj` has type `X`, the function squares the field `obj.x`. If `obj` has type `SubX`, `squareX` squares the fields `x` and `y`.

```
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 2.7: The same two classes from Figure 2.6, but with the added predicate family `started()`.

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

Figure 2.8: We define the method `squareX`, which takes an instance of `X`.

Chapter 3

Pure Function Overrides

Prior to this thesis, Nagini only allowed overriding impure functions, hereafter referred to as *methods*, to comply with the naming conventions of Viper. We extend Nagini to also include the overriding of pure functions, hereafter referred to as *functions*, as described in Section 3.1. Furthermore, behavioural subtyping checks were limited to methods. We have generalised this mechanism to also support functions (see Section 3.2). We use these additions to encode overridden opaque function applications in Viper, as described in Section 3.3.

3.1 Opaque Function Support

By default, Nagini shows that for each call to a dynamically bound method, the preconditions of the statically bound method are satisfied. Additionally, it is verified that behavioural subtyping (see Section 2.3) holds. Hence, the postconditions of the overridden method are stronger (or maintained), clearly satisfying the specification of the statically bound method. Consequently, we can assume the postconditions of the overridden method after a call. Furthermore, the current verifier may use the function definition (in addition to its postconditions) for every function call, unlike method calls, which only use its contract.

We extend Nagini as follows: We treat pure functions overrides similarly to method overrides. By default, we mark functions as opaque. This ensures that the caller learns only the specification and not the implementation of a function (see Section 2.6). Additionally, behavioural subtyping is verified to hold for each function and its overrides. Hence, for each dynamically bound function call, Nagini shows that the preconditions of the statically bound function hold and assumes its postconditions.

If function implementations, rather than just contracts, were used during

```
class X:
    @Pure
    def foo(self, i: int) -> int:
        Ensures(i >= 0)
        return i * i

class SubX(X):
    @Pure
    def foo(self, i: int) -> int:
        Ensures(i >= 1)
        return i ** 4 + 1

@Pure
def bar(x: X, y: X):
    i = 2
    a = x.foo(i)  # a = 2 * 2 = 4
    b = y.foo(i)  # b = 2^4 + 1 = 17
    assert a != b

x = X()
y = SubX()
bar(x, y)
```

Figure 3.1: Motivational example highlighting the need for opaque functions when overriding functions.

verification, unsoundness could be introduced as illustrated in the example in Figure 3.1. The function `bar` declares the parameters `x` and `y` with static type `X`; however, `y` has the type of `SubX` at runtime. According to the definitions of `X.foo` and `SubX.foo`, the function calls `x.foo(i)` and `y.foo(i)` yield $2 \cdot 2 = 4$ and $2^4 + 1 = 17$, respectively. If, instead, the function definitions associated with the static types were used during verification, `y.foo(i)` would have returned 4, leading to a violation of the assertion `assert a != b`. Therefore, to ensure soundness, we treat all overridden functions and their corresponding overrides not only as pure, but also as opaque. Furthermore, we implicitly define functions that are not subject to overriding as opaque to minimise annotation overhead, but they can be set to transparent, i.e., non-opaque, by using the `@Transparent` decorator.

3.2 Override Check Encoding

As with impure methods, we also require pure functions to satisfy behavioural subtyping. As mentioned in Section 2.3, an overriding function must comply with the specification of the overridden function. Con-

sider the example in Figure 3.2, which uses arbitrary preconditions P and P', and postconditions Q and Q'. We define an override check function `SubX_bar_override_check` in Viper with the same contract as `X.bar`, differing only in the type of the parameter `self`, which has the type `SubX` instead of `X`. The body of the override check function consists of a call to the overriding function `SubX.bar`, which is opaque. This allows the verifier to prove that `SubX.bar` complies with the specification of the function it overrides, i.e., that we have:

$$P \models P' \qquad \text{(Precondition of SubX is weaker or maintained)}$$
$$Q' \models Q \qquad \text{(Postcondition of SubX is stronger or maintained).}$$

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

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

```
0  function SubX_bar_override_check(self: Ref, i: Ref)
1      requires type(self) <: SubX
2      requires type(i) <: int
3      requires P
4      ensures Q
5  {
6      res := SubX_bar(self, i)
7  }
```

Figure 3.2: Encoding the behavioral subtyping check for the Python class `X` and its subclass `SubX` in the Viper function `SubX_bar_override_check`. The example is adapted from the example on page 39 of [2].

3.3 Modelling Overridden Function Applications

If someone calls `x.foo()` on some receiver of type `X`, what should they be allowed to assume, and how can that be encoded into Viper? To address this question, we explore multiple approaches to encode applications of pure

overridden functions into Viper[1]. We present two possibilities in this section, each with different trade-offs regarding performance and completeness. In the first option, we utilise a merge function, as described in Subsection 3.3.1. In the second approach, we encode an extended contract-only version for each function, additionally ensuring compliance with the supertype's specification, as detailed in Subsection 3.3.2.

3.3.1 Merge Function-Based Encoding

The first approach to encoding pure overridden functions into Viper is to define a merge function that aggregates all the pre- and postconditions of both the overriding and the overridden functions in its contract.

Consider the example in Figure 3.3, which shows a function `A.foo` and its overriding function `B.foo`. Behavioural subtyping is satisfied. The resulting merge function `A_foo_merged` combines the pre- and postconditions of all relevant classes into a single contract. First, type and null checks are encoded, which are identical to those in `A.foo`. Next, the preconditions are encoded using one conditional expression that distinguishes the classes using subtyping checks. For each type, the conditional expression contains a subtyping check as the condition and the corresponding precondition as the resulting expression. If `self` is of type `B`, then $\texttt{i} > 0$ holds. If `self` is of type `A`, the alternative case applies, i.e., $\texttt{i} > 2$ holds. Subsequently, the type checks for the return values are encoded. Finally, the postconditions are encoded as implications, where the antecedent consists of the relevant subtyping check and the consequent corresponds to the postcondition of the appropriate function (e.g., `A.foo` or `B.foo`)[2].

The preconditions are in reverse topological order with respect to the class hierarchy. Thus, the condition `type(self) <: B() ? unbox(i) > 0` is placed before `type(self) <: A() ? unbox(i) > 2`, since `B` $<:$ `A`. This ordering prevents a situation where a precondition of the overridden function incorrectly holds. Consider the example shown in Figure 3.4, where `self` is of type `B`. Then, $\texttt{i} > 0 \land \texttt{i} > 2 \equiv \texttt{i} > 2$ must hold since `B` $<:$ `A`. However, this is unsound, due to the fact that the precondition of `B.foo` only requires the weaker precondition $\texttt{i} > 0$. `B.foo` should be callable with the values $\texttt{i} = 1$ or $\texttt{i} = 2$, in contrast to `A.foo`, which requires $\texttt{i} > 2$.

For postconditions, implications can be used, as the postcondition of the overriding function may only be strengthened (or maintained). Consequently, the postcondition of the overridden function still holds, e.g., for `B.foo` $\texttt{Result()} > 5 \land \texttt{Result()} > 3 \equiv \texttt{Result()} > 5$.

[1]This sentence was revised using ChatGPT 4o.
[2]This sentence was revised using ChatGPT 4o.

```
class A:
    @Pure
    def foo(self, i: int) -> int:
        Requires(i > 2)
        Ensures(Result() > 3)
        return i + 1

class B(A):
    @Pure
    def foo(self, i: int) -> int:
        Requires(i > 0)
        Ensures(Result() > 5)
        return i + 5
```

```
function A_foo_merged(self: Ref, i: Ref): Bool
    requires type(self) <: A()
    requires type(i) <: int()
    requires self ≠ null
    requires type(self) <: B() ? unbox(i) > 0 :
             (type(self) <: A() ? unbox(i) > 2 : true)

    ensures type(self) <: B() ⟹ type(result) <: int()
    ensures type(self) <: A() ⟹ type(result) <: int()
    ensures type(self) <: A() ⟹ result > 3
    ensures type(self) <: B() ⟹ result > 5

```

Figure 3.3: Encoding of the merge function `A_foo_merged`, which combines the contracts of the two functions `A.foo` and `B.foo`.

```
function SomeClass_foo_merged(self: Ref, i: Ref): Bool
    ...
    requires type(self) <: A()) ==> unbox(i) > 2
    requires type(self) <: B()) ==> unbox(i) > 0
    ...

```

Figure 3.4: A merge function that encodes preconditions using implications instead of a conditional expression.

```
0  @opaque()
1  function SubX_foo(self: Ref, i: Ref): Ref
2      requires type(self) <: SubX()
3      ...
4      ensures unbox(i) >= 1
5  {
6      box(
7          power(unbox(i), 4) + 1
8      )
9  }
10
11 function SubX_foo_extended(self: Ref, i: Ref): Ref
12     requires type(self) <: SubX()
13     ...
14     ensures unbox(i) >= 1
15     ensures result == X_foo(self, i)
16
```

Figure 3.5: Encoding of `SubX.foo` and its extended function into Viper.

When a user calls `A.foo` or `B.foo`, we translate the invocation to a call to the corresponding merge function. For instance, if `B().foo(5)` is used, Nagini verifies that the precondition corresponding to the type `B` of `A_foo_merged` is satisfied. In particular, it checks that `i` is of type `int`, that $\texttt{i} := 5 > 0$ holds, etc. Finally, the corresponding postcondition $\texttt{Result()} > 5 \wedge \texttt{type(Result())} <: \texttt{int}$ is assumed.

3.3.2 Extended Function-Based Encoding

An alternative to using a merge function is to extend each function with an additional postcondition that ensures compliance with the supertype's specification. We impose an additional restriction that the precondition of the overridden function is not weaker, otherwise, the added postcondition may not be well defined. First, we verify that the function satisfies behavioural subtyping, as detailed in Section 3.2. Next, we create a contract-only version of the function and add a postcondition that ensures compliance with the supertype's specification. Consider the example in Figure 3.5, where the class `SubX` from the example in Figure 4.10 is encoded into Viper. Since `SubX.foo` overrides `X.foo`, the postconditions of `SubX` are extended with `ensures result == X_foo(self, other)`. Finally, all calls to `SubX.foo` during verification are redirected to this extended function.

3.3.3 Trade-Offs in Override Encoding

If we use a single merge function, the completeness is improved in comparison to the extended function approach, because of the incorporated subtyping checks. They enable the verifier to reason using exact runtime

information, rather than relying solely on static type information[3]. However, the introduction of the merge function may lead to performance degradation. In contrast, the extended function approach ensures compliance with the supertype's specification. Hence, the performance is increased at the cost of reduced completeness, due to the fact that only static type information is available for the verifier. The evaluation details are described in Chapter 5.

[3]This sentence was revised using ChatGPT 4o.

Chapter 4

Object Equality Reasoning

In Section 4.1, we describe the verification of the equality function's definition, showing how to modularly prove that a user-defined equality function satisfies the properties of an equivalence relation. The Section 4.2 illustrates the specifications of all built-in and user-defined equality contracts, the definition of the predicate family `state`, and the modelling of equality function applications.

4.1 Verifying Equality Function Definitions

We want a modular and sound verification approach. Hence, two independently verified code examples should still verify if they are combined in a module, for example. Soundness is required, as otherwise the verification results would be practically meaningless. Additionally, completeness and efficient performance are desirable properties; we aim to enable verification for as many programs as possible and to do so within a reasonable amount of time.

Furthermore, we aim to establish that equality defined for subtypes is consistent with the equality semantics of their supertypes. Consider the example in Figure 4.1. The function `foo` compares its two arguments `o1` and `o2` twice. First, they are compared for equality when only static type information is available. Subsequently, if both are instances of integers, their casted versions are compared. Nagini should be able to successfully verify that if we have `o1` == `o2`, then `i1` == `i2` should also hold.

We use Nagini to independently verify that all user-defined equality functions' postconditions are provable assuming their preconditions. Additionally, we prove that the equality functions comply with the supertype's equality specification, as detailed in Section 3.2.

In this section, we aim to verify the following properties:

```
def foo(o1: object, o2: object):
    tmp = o1 == o2
    if isinstance(o1, int) and isinstance(o2, int):
        i1 = cast(int, o1)
        i2 = cast(int, o2)
        assert tmp == (i1 == i2)
```

Figure 4.1: Example demonstrating the problem of equality consistency.

- Modularity
- Reflexivity
- Symmetry
- Transitivity

For each user-defined equality function `C.__eq__` we encode a modularity postcondition (see Postcondition 4.1), a reflexivity postcondition (see Postcondition 4.2), a method to verify symmetry (see Figure 4.5), and a method to prove transitivity (see Figure 4.6).

We introduce the notation $\texttt{a} ==_{\texttt{X}} \texttt{b}$ for some references `a` and `b`. This denotes that `a` and `b` are compared for equality according to the implementation or contract of `X.__eq__`, depending on whether the dynamic type of `a` is known. Hence, `a` is an instance of type `X`. The exact encoding of these calls into Viper is discussed in Section 4.2.4. For now, we assume that $\texttt{a} ==_{\texttt{X}} \texttt{b}$ and $\texttt{b} ==_{\texttt{Y}} \texttt{a}$ are encoded as calls to two distinct functions in Viper, representing `X.__eq__` and `Y.__eq__`, respectively.

4.1.1 Modular Verification

By modular verification, we mean that if all components of a program are verified in isolation, then they must also verify when combined. Consider the example in Figure 4.2, where two classes `X` and `Y` are defined. Suppose class `Y` is unknown when verifying `X`. An instance of `X` might be equal to an instance of `Y`, if the values of `self.i` and `other.i` are identical. However, we cannot ensure that the equality function of `Y` behaves the same way, since `Y` is unknown when `X` is verified. Thus, symmetry might be violated.

To prevent this, we establish for each class `X` a *set of mentioned classes* $\mathcal{M}_{\texttt{X}}$ such that an instance of `X` can only be equal to instances of subtypes of classes in $\mathcal{M}_{\texttt{X}}$. Thus, if we check symmetry between `X` and all classes in $\mathcal{M}_{\texttt{X}}$, we know that there is no currently unknown type `Y` whose instances an `X` instance might be equal to, for which symmetry is violated: All `X` instances are not equal to any instances of types not in $\mathcal{M}_{\texttt{X}}$.

```
# Suppose class Y is unknown when verifying X
class X:
    def __init__(self, i: int):
        self.i: int = i

    def __eq__(self, other: object) -> bool:
        if hasattr(other, 'i') and self.i == other.i:
            return True
        return False

class Y:
    def __init__(self, i: int):
        self.i: int = i
    ...
```

Figure 4.2: Motivational example illustrating problems in modular verification.

```
class X:
    def __eq__(self, other: object) -> bool:
        if type(other) in (A, B) or type(self) == type(other) :
            return True
        return False
```

Figure 4.3: Example illustrating the definition of $\mathcal{M}_X$ in 4.1.

Definition 4.1 *Let $\mathcal{S}$ be the set of all user-defined classes. The mentioned set $\mathcal{M}_X$ for a class $X \in \mathcal{S}$ that implements an equality function f, contains all explicitly mentioned classes $C \in \mathcal{S}$ defined within its function body. Consequently, $\mathcal{M}_X \subseteq \mathcal{S}$ holds.*

$$\mathcal{M}_X := \{C \mid C \in \mathcal{S} \text{ and } C \in body(f) \text{ is explicitly mentioned}\}$$

For the example in Figure 4.3, the mentioned set for `X.__eq__` is $\mathcal{M}_X := \{A, B\}$, since only the classes `A` and `B` are explicitly mentioned; the class `X` is implicitly mentioned.

Definition 4.1 enables the use of the following postcondition in each equality function's contract:

$$\texttt{ensures result} \implies \left(\bigvee_{C \in \mathcal{M}_X} \texttt{type(other)} <: C \right) \vee \texttt{type(self)} == \texttt{type(other)} \tag{4.1}$$

If `result` is `true`, `other` must either be an instance of one of the classes explicitly mentioned in $\mathcal{M}_X$, or must have the same type as `self`.

Consider the example in Figure 4.2 again. If we add the postcondition in 4.1 to both equality functions, then an instance of `X` can only be considered equal

to some instance of Y if both $\texttt{Y} \in \mathcal{M}_\texttt{X}$ and $\texttt{X} \in \mathcal{M}_\texttt{Y}$ hold. Consequently, with the added postcondition, Nagini now rejects the example.

4.1.2 Verifying Reflexivity

With modularity now established, we can proceed to the properties of an equivalence relation. We begin by considering reflexivity. We aim to prove that for `a` and `a'`, which are aliases of type `A`, the condition $\texttt{a} ==_\texttt{A} \texttt{a'}$ is satisfied. We add the following postcondition to each custom equality function, to verify that reflexivity holds:

$$\texttt{ensures self} == \texttt{other} \implies \texttt{result} \tag{4.2}$$

If `self` is an alias of `other`, `result` is `true`. Consequently, we add the postcondition 4.2 to each user-defined equality function to enable Nagini to verify whether reflexivity holds.

4.1.3 Verifying Symmetry

The second property of an equivalence relation, symmetry, is more involved to verify than reflexivity. We want to verify that for two given instances `self` and `other` of types `X` and `O`, respectively, the condition $\texttt{self} ==_\texttt{X} \texttt{other} \implies \texttt{other} ==_\texttt{O} \texttt{self}$ is satisfied. Since `X` and `O` may be distinct types, we must establish symmetry between potentially two different equality functions.

Additionally, we exploit the fact that for each user-defined equality function, the modularity postcondition 4.1 holds. Hence, for each class `X`, we want to show the following: If $\texttt{self} ==_\texttt{X} \texttt{other}$ returns `true`, then for any class $\texttt{C} \in \mathcal{M}_\texttt{X}$, if `other` is an instance of `C`, then $\texttt{other} ==_\texttt{C} \texttt{self}$ evaluates to `true`. Or if `self` and `other` have the same type, we aim to prove: If $\texttt{self} ==_\texttt{X} \texttt{other}$ returns `true`, then $\texttt{other} ==_\texttt{X} \texttt{self}$ also evaluates to `true`.

Consider the example in Figure 4.4, where two classes `A` and `B` are defined with `B <: A`. Because `B` inherits the equality definition from `A`, every instance of `A` or `B` is equal to every instance of `A` or `B`. Ideally, we could encode symmetry as a postcondition, analogous to reflexivity, additionally allowing every caller to learn that the function is symmetric. However, this would introduce a cyclic dependency between the two proof obligations. The postcondition of `A.__eq__` would include $\texttt{other} ==_\texttt{B} \texttt{self}$ and the postcondition of `B.__eq__` would include $\texttt{other} ==_\texttt{A} \texttt{self}$. Hence, Viper could not verify symmetry in this case.

To prevent the issue of mutual recursion, we propose the following approach: For each class $\texttt{X} \in \mathcal{S}$ and the associated set $\mathcal{M}_\texttt{X} := \{C_0, C_1, \ldots, C_{k_\texttt{X}}\}$, where $k_\texttt{X} \in \mathbb{N}$ and $\mathcal{M}_\texttt{X}$ is ordered in reverse topological ordering with respect to the class hierarchy, we encode the method illustrated in Figure 4.5.

```python
class A:
    def __eq__(self, other: object) -> bool:
        # ensures type(other) == A or type(other) == B ==> result
        if type(other) == A or type(other) == B:
            return True
        return False

class B(A):
    pass
```

Figure 4.4: Motivational example demonstrating the problem of mutual recursion.

```
0  method X___eq___check_symmetry(self: Ref, other: Ref) {
1      var res: Ref
2      assume type(res) <: bool()
3      assume type(self) == X()
4      assume
                ( ⋁_{C_i ∈ M_X} type(other) <: C_i ) ∨ type(self) == type(other)
5      inhale acc(state(self), wildcard)
6      inhale acc(state(other), wildcard)
7      assume Body(self ==_X other)[__eq__ ↦ eq]
8      if (type(self) == type(other) {
9          assert Body(other ==_X self)[__eq__ ↦ eq]
10     } elseif (type(other) <: C_0) {
11         assume Posts(other ==_{C_0} self) [__eq__ ↦ eq, result ↦ res]
12         assert res
13     } ... {
14         ...
15     } elseif (type(other) <: C_{k_X}) {
16         assume Posts(other ==_{C_{k_X}} self) [__eq__ ↦ eq, result ↦ res]
17         assert res
18     }
19 }
```

Figure 4.5: A Viper method to modularly prove symmetry of an equality function `X__eq__`.

Conceptually, we assume that `self` is equal to `other` according to the implementation of the equality function of type `X`. Subsequently, we verify that `other` is equal to `self` according to the definition of equality for each possible type `other` may have. Depending on the situation, the implementation or only the contract is used.

The verification process should be modular, i.e., code examples verified in isolation also verify combined. However, many equality functions call other equality functions in their implementations, e.g., on fields of the current class. If we just naively called all these equality functions, we could depend on lots of other user-defined equality functions, for which we also aim to prove the properties of an equivalence relation. Hence, the initial equality function cannot be verified in isolation, as it could depend on other user-defined equality functions. To prevent this, we substitute all recursive calls to the domain function `eq`, which satisfies all properties of an equivalence relation.

The method has two distinct components: The first part consists of the initial assumptions and variable declarations (from line 2 to 7). The method assumes that `self` is of type `X`, as the method is generated for all classes $\texttt{X} \in \mathcal{S}$. Next, the type of the local variable `res` is assumed to be of type `bool`. The purpose of the variable will be discussed shortly. Subsequently, we assume that `other` is either an instance of one of the classes in $\mathcal{M}_{\texttt{X}}$, or has the exact same type as `self`, i.e., `X`. This is the consequent of the modularity postcondition 4.1. Then, we inhale a wildcard amount of the permissions required by the corresponding precondition of $\texttt{self} ==_{\texttt{X}} \texttt{other}$, i.e., the permissions to the fields of `self` and `other`. Finally, we assume the body of $\texttt{self} ==_{\texttt{X}} \texttt{other}$ to be `true`, where all recursive calls to equality functions are substituted with the domain function `eq`.

The second component consists of verifying that the equality function call with flipped arguments holds, i.e., $\texttt{other} ==_{\texttt{O}} \texttt{self}$, where $\texttt{O} := \texttt{type(other)}$ (from line 8 to 21). We perform a case distinction on the type of `other`. In the first case, `self` and `other` have the same type `X`, hence, we assert that the body of $\texttt{other} ==_{\texttt{X}} \texttt{self}$ is `true`. Additionally, all recursive calls to equality functions are substituted with the domain function `eq`. Next, we encode a case for each $\texttt{C}_i \in \mathcal{M}_{\texttt{X}}$, where $i \in \{0, ..., k_{\texttt{X}}\}$. Each case has the following structure: If `other` is an instance of the current class $\texttt{C}_i \in \mathcal{M}_{\texttt{X}}$, then we assume the postcondition of the call that compares `other` to `self` according to $==_{\texttt{C}_i}$. All recursive calls to equality functions are also substituted with the domain function `eq`. Additionally, the `result` variable in the postconditions is replaced with the variable `res` to locally capture the guarantees for the result of the call. Finally, we assert that `res` is `true` to verify that the equality function call with flipped arguments returns `true`.

Soundness

To begin with, the substitution of recursive calls to equality functions with the domain function `eq`, which is assumed to be an equivalence relation, is sound. This is because we only prove the implication: assuming that all other user-defined and built-in equality functions, as well as all recursive calls to `X.__eq__`, satisfy the properties of an equivalence relation, it follows that `X.__eq__` itself is also an equivalence relation. Since the assumption is independently verified for all other custom equality functions, the conclusion holds. Second, we inhale all necessary permissions between lines 5 and 6, i.e., the ones required in the precondition of $==_{\texttt{X}}$. This is sound, since we only want to verify symmetry in cases where `X.__eq__` can be invoked, and in those situations, this precondition must have been established. If the precondition is unsatisfiable (e.g., contains `false`), the function cannot be called, and we do not need to show symmetry. However, there can be well-definedness issues anywhere in the method, e.g., a function call fails due to missing permissions not covered by the inhale statements. Third, we are allowed to assume that the implementation of `X`'s equality function is true, due to the assumption on line 3 that `self` has the exact type `X`. Consider the example where the implementation of `self` $==_{\texttt{X}}$ `other` is `false`. Then we assume `false` on line 7 and the lines between 8 and 21 are skipped. In the method generated for the type of `other`, namely `O`, we either assume `false` too and symmetry holds, or otherwise one of the latter assertions must be violated, signalling that the two functions are not symmetric. Finally, we consider the following cases.

Case 1 - `type(self)` $==$ `type(other)`:
Since `type(self)` $==$ `type(other)` holds, the branch on line 9 is taken regardless of the elements contained in $\mathcal{M}_{\texttt{X}}$. Based on the assumption on line 4, and the assertion on line 9, we establish that `self` $==_{\texttt{X}}$ `other` $\implies$ `other` $==_{\texttt{X}}$ `self` and in both cases, the implementation of $==_{\texttt{X}}$ is utilised. This is valid because both `self` and `other` have the exact type of `X`.

Case 2 - $\mathcal{M}_{\texttt{X}} \neq \emptyset \wedge$ `type(self)` $\neq$ `type(other)`:
For each $\texttt{C}_\texttt{i}$ in $\mathcal{M}_{\texttt{X}}$, where $\texttt{i} \in \{0, ..., k_{\texttt{X}}\}$, a conditional branch is encoded, e.g., the branch on line 11 for $\texttt{C}_0$. In each branch, we assume the postcondition of `self` $==_{\texttt{C}_\texttt{i}}$ `other`. The guarantees provided by the result of the invocation are transferred to the local variable `res`, which we subsequently assert to be `true`. As a result, we obtain:

$$\forall \texttt{C}_\texttt{i} \in \mathcal{M}_{\texttt{X}} \, (\texttt{self} ==_{\texttt{X}} \texttt{other} \implies \texttt{other} ==_{\texttt{C}_\texttt{i}} \texttt{self}) \tag{4.3}$$

Thus, it is possible to verify whether symmetry holds in each of the cases discussed above.

4.1.4 Verifying Transitivity

Verifying transitivity is similar to proving symmetry. For the given instances `self`, `other1`, and `other2` of types `X`, $\texttt{O}_1$, and $\texttt{O}_2$, respectively, we aim to prove that $\texttt{self} ==_{\texttt{X}} \texttt{other1} \land \texttt{other1} ==_{\texttt{O}_1} \texttt{other2} \implies \texttt{self} ==_{\texttt{X}} \texttt{other2}$ holds. Since `X`, $\texttt{O}_1$, and $\texttt{O}_2$ may be different types, we must establish transitivity between potentially three distinct equality functions.

Ideally, we could encode transitivity as a postcondition, allowing every caller to learn that the equality function is transitive. This would require permissions to access an arbitrary object, which cannot be obtained within a postcondition in Viper. Hence, we propose the following approach: For each class $\texttt{X} \in \mathcal{S}$ and the associated set $\mathcal{M}_{\texttt{X}} := \{C_0, C_1, \ldots, C_{k_{\texttt{X}}}\}$, where $k_{\texttt{X}} \in \mathbb{N}$ and $\mathcal{M}_{\texttt{X}}$ is ordered in reverse topological ordering with respect to the class hierarchy, we encode the method illustrated in Figure 4.6.

Conceptually, we assume that `self` is equal to `other1` and equal to `other2` according to the implementation of the equality function of type `X`. Additionally, we exploit the fact that the modularity postcondition 4.1 holds; we know that `self` can only be equal to `other1` instances of type `X`, or instances of subclasses of $\texttt{C}_i \in \mathcal{M}_{\texttt{X}}$. The same holds for instances of `other2`. Since we additionally assume that symmetry holds, it suffices to show that `other1` is equal to `other2` according to the equality definition each possible type `other1` may have. Depending on the situation, the implementation or only the contract is used.

The method has two distinct components: The first part consists of the initial assumptions and variable declarations (from line 2 to 11). The method assumes that `self` is of type `X`, as the method is generated for all classes $\texttt{X} \in \mathcal{S}$. Next, the type of the local variable `res` is assumed to be of type `bool`; it is used for the same purpose as in the Figure 4.5, i.e., to capture the guarantees of the result of the subsequent call in the second component. Subsequently, we assume that `other1` and `other2` are instances of one of the classes in $\mathcal{M}_{\texttt{X}}$, or have the same type as `self`, i.e., `X`; this is the consequent of the modularity postcondition 4.1. In particular, it is possible that `other1` has the same type as `self` while `other2` is an instance of one of the classes in $\mathcal{M}_{\texttt{X}}$, or vice-versa. Then, we inhale a wildcard amount of the permissions required by the corresponding precondition of $\texttt{self} ==_{\texttt{X}} \texttt{other1}$ and $\texttt{self} ==_{\texttt{X}} \texttt{other2}$, i.e., the permissions to the fields of `self`, `other1`, and `other2`. Finally, we assume the implementation of $\texttt{self} ==_{\texttt{X}} \texttt{other1}$ and $\texttt{self} ==_{\texttt{X}} \texttt{other2}$ to be `true`, where all recursive calls to equality functions are substituted with the domain function `eq`.

The second component consists of verifying that the equality function call $\texttt{other1} ==_{\texttt{O}_1} \texttt{other2}$ returns `true`, where $\texttt{O}_1 := \texttt{type(other1)}$ (from line 12 to 22). We perform a case distinction on the type of `other1`. In the first

```
0  method X___eq___check_transitivity(self: Ref, other1: Ref, other2
       : Ref)
1  {
2      var res: Ref
3      assume type(res) <: bool()
4      assume type(self) == X()
5      assume
                 ( ⋁_{C_i ∈ M_X} type(other1) <: C_i ) ∨ type(self) == type(other1)
6      assume
                 ( ⋁_{C_i ∈ M_X} type(other2) <: C_i ) ∨ type(self) == type(other2)
7      inhale acc(state(self), wildcard)
8      inhale acc(state(other1), wildcard)
9      inhale acc(state(other2), wildcard)
10     assume Body(self ==_X other1)[__eq__ ↦ eq]
11     assume Body(self ==_X other2)[__eq__ ↦ eq]
12     if (type(self) == type(other1)) {
13         assert Body(other1 ==_X other2)[__eq__ ↦ eq]
14     } elseif (type(other1) <: C_0) {
15         assume Posts(other1 ==_{C_0} other2)[__eq__ ↦ eq, result ↦ res]
16         assert res
17     } ... {
18         ...
19     } elseif (type(other1) <: C_{k_X}) {
20         assume Posts(other1 ==_{C_{k_X}} other2)[__eq__ ↦ eq, result ↦ res]
21         assert res
22     }
23 }
24
```

Figure 4.6: A Viper method to modularly prove transitivity of an equality function `X___eq__`.

case, `self` and `other1` have the same type `X`, hence, we assert that the body of `other1` $==_{\texttt{X}}$ `other2` is `true`. Additionally, all recursive calls to equality functions are substituted with the domain function `eq`. Next, we encode a case for each $\texttt{C}_i \in \mathcal{M}_{\texttt{X}}$, where $i \in \{0, ..., k_{\texttt{X}}\}$. Each case has the following structure: If `other1` is an instance of the current class $\texttt{C}_i \in \mathcal{M}_{\texttt{X}}$, then we assume the postcondition of the call that compares `other1` to `other2` according to $==_{\texttt{C}_i}$. All recursive calls to equality functions are also substituted with the domain function `eq`. Additionally, the `result` variable in the postconditions is replaced with the variable `res` to locally capture the guarantees for the result of the call. Finally, we assert that `res` is `true` to prove or disprove that the equality function call `other1` $==_{\texttt{C}_i}$ `other2` holds.

Soundness

First, analogous to symmetry, substituting any recursive call to an equality function with the domain function `eq` is sound. This is because we only prove the implication: assuming that all other user-defined and built-in equality functions, as well as all recursive calls to `X.__eq__`, satisfy the properties of an equivalence relation, it follows that `X.__eq__` itself is also an equivalence relation. Since the assumption is independently verified for all other custom equality functions, the conclusion holds. Second, we inhale all necessary permissions between lines 7 and 9, i.e., the ones required in the precondition of `self` $==_{\texttt{X}}$ `other1` and `self` $==_{\texttt{X}}$ `other2`. This is sound, since we only want to verify transitivity in cases where `X.__eq__` can be invoked, and in those situations, this precondition must have been established. If the precondition is unsatisfiable (e.g., contains `false`), then the function cannot be called, and we do not need to show transitivity. However, there can be well-definedness issues anywhere in the method, e.g., a function call fails due to missing permissions not covered by the inhale statements. Third, we are allowed to assume that the implementation of `X`'s equality function is true on lines 10 and 11, due to the assumption on line 4 that `self` has the exact type `X`. Finally, we consider the following cases.

Case 1 - `type(self)` $==$ `type(other1)`:
Since `self` and `other1` have the same type, the branch on line 13 is taken regardless of the elements contained in $\mathcal{M}_{\texttt{X}}$. Based on the assumptions on lines 10 and 11, the already established symmetry, and the assertion on line 13, the resulting value is bound to `res`. Thus, by asserting `res` is `true`, it is verified whether transitivity holds. Conceptually, we obtain:

$$\texttt{other1} ==_{\texttt{X}} \texttt{self} \land \texttt{self} ==_{\texttt{X}} \texttt{other2} \implies \texttt{other1} ==_{\texttt{X}} \texttt{other2} \tag{4.4}$$

Since all exact types are known, the function definition is utilised for each proof.

Case 2 - $\mathcal{M}_{\mathtt{X}} \neq \emptyset \wedge \mathtt{type(self)} \neq \mathtt{type(other1)}$:
For each $\mathtt{C}_i$ in $\mathcal{M}_{\mathtt{X}}$ for $i \in \{0, ..., k_{\mathtt{X}}\}$, a conditional branch is encoded, e.g., the branch on line 15 for $\mathtt{C}_0$. Based on the assumptions on lines 10 and 11, the already established symmetry, and the corresponding assumption of the postcondition of $\mathtt{other1} ==_{\mathtt{C}_i} \mathtt{other2}$, the resulting value is bound to `res`. Conceptually, we have that:

$$\begin{aligned} &\forall \mathtt{O}_i \in \mathcal{M}_{\mathtt{X}}(\mathtt{other1} ==_{\mathtt{C}_i} \mathtt{self} \wedge \mathtt{self} ==_{\mathtt{X}} \mathtt{other2} \\ &\implies \mathtt{other1} ==_{\mathtt{C}_i} \mathtt{other2}) \end{aligned} \tag{4.5}$$

Thus, it is possible to verify whether transitivity holds in all cases discussed above.

Hence, we modularly verify that each user-defined equality function satisfies the properties of an equivalence relation or reject it otherwise.

4.2 Specifications of Equality Functions

In this section, we present a framework for specifying equality functions that can be overridden. For this, we define the contract for object equality, as described in Subsection 4.2.1. Next, we propose the predicate family `state`, which enables reasoning about permissions of different types while satisfying behavioural subtyping (see Section 4.2.2). Subsequently, we present the contracts for all built-in subtypes of `object`, as described in Section 4.2.3. Finally, we model equality function applications, as illustrated in Section 4.2.4.

4.2.1 Object Equality Contract

We proceed to define the object equality contract, as illustrated in Figure 4.7. It is important because it establishes minimal properties that all equality definitions must adhere to; it should be as strong as possible while allowing all reasonable implementations. In the two preconditions, we require access to the predicate family `state`, which is intended to include all necessary permissions needed. The predicate `state` and the function `stateless` will be defined shortly in Subsection 4.2.2. Next, we add a postcondition for `None` equality. Finally, we add two postconditions to ensure the desired properties of the equality operator `==`, i.e., reflexivity (except for floating points) and symmetry. Floating-point numbers are not included in the reflexivity postcondition, due to their inaccurate encoding. Consider the example `0.1 + 0.2 == 0.3` in Python; it returns `False` due to small rounding errors. Transitivity, however, cannot be encoded as a postcondition, as in Viper, we do not have access to the predicate family `state` for some arbitrary object.

We use the object equality contract as a base and ensure that each user-defined and built-in equality override satisfies behavioural subtyping.

```
0 def object___eq__(self, other: object) -> bool:
1     # requires ¬stateless(self) ⟹ Acc(state(self))
2     # requires ¬stateless(other) ⟹ Acc(state(other))
3     # ensures self == None ⟹
4     #     result == (other is None)
5     # ensures type(other) ⊀: float() ⟹ self is other ⟹ result
6     # ensures result == (other ==object self)
7
```

Figure 4.7: The object equality contract.

4.2.2 Predicate Family `state`

Equality functions typically access the state of the objects they are comparing, and therefore require permission to those heap locations. They differ from type to type. Consider the example previously mentioned in Section 2.10, where a type `A` requires `Acc(self.i)`, but the subtype `B` requires `Acc(self.i) and Acc(self.j)`. Since the precondition in the subtype is strengthened, `B` is not a behavioural subtype of `A`.

To allow different types to require access to distinct heap locations and satisfy behavioural subtyping, we define a predicate family called `state` (see Section 2.10). If a subclass overrides the predicate `state` of a superclass, then the subclass predicate `state` is defined as the conjunction of the implementation of both predicates. It is intended to contain permissions to all heap locations required by the equality function of a given object. For example, if `self` is a list, accessing `state(self)` should give the permissions to the heap locations of all elements in the list. For the base type `object`, the `state` predicate is defined as `true`.

We use `state` in the object equality contract (see Figure 4.7) and therefore also in all other classes that require access to heap locations in their equality functions. Hence, in a typical equality function, we access `state` to acquire the necessary permissions for built-in collections and fields of custom object instances that are involved when comparing two arbitrary Python objects for equality. For example, it is required in the preconditions of the equality merge function, as will be discussed in Section 4.2.4.

Consider the example where two integers are compared within a function `foo`. To perform the comparison, the function requires access to the predicate family `state` and must unfold `state` before the comparison and fold it again afterwards. Hence, even for inherently stateless objects such as integers, all of this additional overhead is required. To solve this problem, we define a function called `stateless`, as illustrated in Figure 4.8. As the name suggests, this function returns `true` if and only if the argument is stateless, i.e., an instance of any primitive type $p \in \mathcal{P}$ or a tuple of stateless objects. We require the `state` predicate family only for non-stateless objects. Thus, when we compare two integers, no predicate folding is necessary.

```
0 def stateless(x: object) -> bool:
1   # ensures result ==
```

$$\left(\bigvee_{T \in \mathcal{P}} \texttt{isinstance(x, T)}\right) \vee \left(\texttt{isinstance(x, tuple)} \wedge \bigwedge_{y \in x} \texttt{stateless(y)}\right)$$

Figure 4.8: The `stateless` function definition.

Collection Predicates `state`

Collections are inherently stateful. Thus, we define a predicate `state` for each collection type in Python. All these predicate definitions follow a similar pattern. If `self` is an instance of the collection type, we acquire the permissions to access each reference `x` in said collection, provided that the reference is not stateless. As illustrated in Figure 4.9, the predicate `state` for lists uses Nagini's built-in predicate P_l, which grants permission to access each element of a list. Similarly, `state` for sets uses P_s; `state` for dictionaries uses P_d for permission to access each key and value in the dictionary. For tuples, the same behaviour is achieved via the function $\mathcal{V}_t$, which returns a tuple as a sequence of references.

Custom Class Predicates `state`

For every user-defined class C that overrides the `__eq__` function, a user can define a predicate `state` to specify the permissions required to access the fields relevant for equality. If a user does not define it, `state` is inherited from its superclass.

Usage of the Predicate Family `state` in Nagini

Consider the example shown in Figure 4.10, where we defined the custom class `X` and its subclass `SubX`. Since `X` declares the field `i`, the call `self.state()` specifies the necessary permissions to access `i`. In contrast, `SubX` introduces an additional field `j`, for which access is also required. We define a call to the `stateless` function (see Figure 4.8) in Nagini as a call to the contract function `Stateless`. We additionally encode access to the `state` predicate family for some variable `v` as `self.state(v)`, if the predicate is defined within the class, or using `state_pred(v)` otherwise. The latter encoding is used to allow the specifications to type-check using Mypy. Within the function body, the relevant predicates must be unfolded to acquire the permissions to access the fields.

Encoding the Predicate Family `state` into Viper

To encode the predicate family `state` into Viper, we use Nagini's existing encoding for predicates and predicate families. We define three sets to

```
# list
@Predicate
def state(self) -> bool:
    return
```

$$\texttt{Acc}(P_{\texttt{l}}(\texttt{self})) \wedge \left(\bigwedge_{\texttt{x} \in P_{\texttt{l}}(\texttt{self})} \neg\texttt{stateless(x)} \implies \texttt{Acc(state(x))} \right)$$

```
# set
@Predicate
def state(self) -> bool:
    return
```

$$\texttt{Acc}(P_{\texttt{s}}(\texttt{self})) \wedge \left(\bigwedge_{\texttt{x} \in P_{\texttt{s}}(\texttt{self})} \neg\texttt{stateless(x)} \implies \texttt{Acc(state(x))} \right)$$

```
# dict
@Predicate
def state(self) -> bool:
    return
```

$$\texttt{Acc}(P_{\texttt{d}}(\texttt{self})) \wedge \left(\bigwedge_{\texttt{x} \in P_{\texttt{d}}(\texttt{self})} \neg\texttt{stateless(x)} \implies \texttt{Acc(state(x))} \right)$$

```
# tuple
@Predicate
def state(self) -> bool:
    return
```

$$\left(\bigwedge_{\texttt{x} \in \mathcal{V}_{\texttt{t}}(\texttt{self})} \neg\texttt{stateless(x)} \implies \texttt{Acc(state(x))} \right)$$

Figure 4.9: The Definitions of the collection `state` predicates.

formalise the encoding: Let S_U denote the set that includes all user-defined `state` predicates encoded into Viper. Let S_B denote the set that contains all built-in collection `state` predicates (see Figure 4.9) encoded into Viper and the predicate `state` for `object`, which is defined simply as `true` in Viper. Let $\mathcal{T}$ denote the set of all user-defined and built-in types encoded into Viper types. With these preparations, we encode the predicate family `state`, as illustrated in Figure 4.11. In Viper, the predicate family `state` is also just a predicate called `state`. It is a conjunction of two components: First, for each predicate $\texttt{state}_\texttt{C} \in S_U \cup S_B$ of class `C`, if `self` is an instance of `C`, then we access $\texttt{state}_\texttt{C}\texttt{(self)}$. The second part encodes that if `self` is not any of the types in $\mathcal{T}$, then an abstract predicate is accessed.

4.2.3 Equality Contracts

We start by formally defining all built-in contracts to enable equality comparisons of built-in types. We ensure that each built-in type is a behavioural

```
class X:
    # Assume self.i is defined
    @Pure
    def __eq__(self, other: object) -> bool:
        Requires(self.state())
        Requires(Implies(not Stateless(other), state_pred(other)))
        ...
        if isinstance(other, X):
            return Unfolding(self.state(),
                Unfolding(other.state(),
                    self.i == other.i
                )
            )
        return False

    @Predicate
    def state(self) -> bool:
        return Acc(self.i)

class SubX(X):
    # Assume self.i and self.j are defined
    @Pure
    def __eq__(self, other: object) -> bool:
        Requires(self.state())
        Requires(Implies(not Stateless(other), state_pred(other)))
        ...
        if isinstance(other, SubX):
            return Unfolding(self.state(),
                Unfolding(other.state(),
                    self.i == other.i and self.j == other.j
                )
            )
        return False

    @Predicate
    def state(self) -> bool:
        return Acc(self.j)
```

Figure 4.10: An example of an `__eq__` override in Python.

```
predicate state(self: Ref) {
    ( ⋀_{state_C ∈ S_U ∪ S_B}  type(self) <: C ⟹ acc(state_C(self)) ) ∧
    ( ⋀_{T ∈ 𝒯}  type(self) ≠ T ⟹ acc(state_abstract_rest(self)) )
}

predicate state_abstract_rest(self: Ref)
```

Figure 4.11: The predicate family `state` encoded as a Viper predicate.

```
function bool___eq__(self: Ref, other: Ref): Bool
    decreases _
    requires
    ensures type(self) <: bool() ∧ type(other) <: bool() ⟹
        result == unbox(self) == unbox(other)
```

Figure 4.12: The contract for boolean equality encoded as a Viper function.

```
function int___eq__(self: Ref, other: Ref): Bool
    decreases _
    ensures type(self) <: int() ∧ type(other) <: int() ⟹
        result == unbox(self) == unbox(other)
```

Figure 4.13: The contract for integer equality encoded as a Viper function.

subtype of `object`. We begin with the contracts of primitive types.

Contracts of Primitives

The contracts of primitives are relatively simple, as their canonical equality semantics are straightforward and access to the `state` predicate is not needed, as primitive types are inherently stateless. All contracts follow a similar pattern: If `self` and `other` are both instances of the corresponding type, then we encode a postcondition that ensures canonical equality for the corresponding type. For example, as shown in Figure 4.12, the postcondition encoding canonical equality ensures that if both `self` and `other` are instances of `bool`, the two arguments are unboxed as Viper boolean values and compared for equality. Integer and string equality are demonstrated in Figures 4.13 and 4.14, respectively, where $\mathcal{L}$ denotes the length function and $\mathcal{V}_{\texttt{str}}$ denotes the string value function.

Equality for floating-point numbers also follows the same structure as the

```
0 function str___eq__(self: Ref, other: Ref): Bool
1     decreases _
2     ensures type(self) <: str() ∧ type(other) <: str()  ⟹
3         result ==
4             (ℒ(self) == ℒ(self) ∧ 𝒱str(self) == 𝒱str(other))
```

Figure 4.14: The contract for string equality encoded as a Viper function.

```
0 function float___eq__(self: Ref, other: Ref): Bool
1     decreases _
2     ensures type(self) <: float() ∧ type(other) <: float()  ⟹
3         isNaN(self) ∨ isNaN(other)  ⟹  !result
```

Figure 4.15: The default contract of equality for floating-point numbers encoded as a Viper function.

```
0 function generic_collection___eq__(self: Ref, other: Ref): Bool
1     decreases _
2     requires ¬stateless(self) ⟹ acc(state(self))
3     requires ¬stateless(other) ⟹ acc(state(other))
4     ensures type(self) <: <Collection Type> ∧ type(other) <: <
      Collection Type>  ⟹
5         <Canonical Equality for Collection Type>
```

Figure 4.16: A generic equality contract for a collection encoded as a Viper function.

other contracts. However, Nagini supports multiple encodings with different guarantees: By default, the only guarantee is that the result is `false` if at least one of the arguments is `NaN`, as shown in Figure 4.15. If floating-point numbers are encoded as reals, then we use the contract shown in Figure A.4. If the floating-point numbers are encoded using IEEE, then we utilise the contract illustrated in Figure A.5.

Contracts of Collections

The contracts of collections are more involved, since canonical equality for collections is not as straightforward as for primitive types, and we need access to the `state` predicate family for both arguments, `self` and `other`. We encode most of the contracts in a similar manner: First, we encode access to the `state` predicate family for `self` and `other`, if they are not stateless, as a precondition to ensure sufficient permissions to access the collection and its elements. Finally, if `self` and `other` are both subtypes of said collection type, then we encode canonical equality as a postcondition.

In Figure 4.17, we define the contract for list equality. It follows the structure of the generic equality contract with corresponding subtype conditions and canonical equality definitions (see Figure 4.16). The canonical equality for

```
0  function list___eq__(self: Ref, other: Ref): Bool
1      decreases _
2      requires ¬stateless(self) ⟹ acc(state(self))
3      requires ¬stateless(other) ⟹ acc(state(other))
4
5      ensures type(self) <: list(...) ∧ type(other) <: list(...)
       ⟹
6          result ==
7          (¬stateless(self) ⟹ unfolding acc(state(self)) in (
8              (¬stateless(other) ⟹ unfolding acc(state(other)) in (
9
                                  ℒ(self) == ℒ(other) ∧
                                  ⋀_{x∈[0, ℒ(self))} (self[x] ==_object other[x])
10             ))
11         ))
```

Figure 4.17: List equality contract encoded as a Viper function.

lists encodes the following: First, `self` and `other` must have the same size; the contracts use $\mathcal{L}$ to denote the length function. Second, each element in `self` at index `x` must be equal to the corresponding element in `other`, where $0 \leq \texttt{x} < \mathcal{L}(\texttt{self})$. If and only if all these conditions are satisfied, `result` is `true`.

In Figure A.1, the contract for set equality is defined similarly to `list___eq__` using the structure of the generic equality contract (see Figure 4.16). The canonical equality for sets encodes the following: If for every element $\texttt{x} \in \texttt{self}$, there exists an element $\texttt{y} \in \texttt{other}$ such that they are equal, and the sizes of the two sets are equal, then the two sets are equal.

Dictionary equality is also comparable to `list___eq__` as shown in Figure A.2. The canonical equality for dictionaries encodes the following: First, `self` and `other` must have the same size. Second, for every key $\texttt{x} \in \texttt{self}$, there must exist a key $\texttt{y} \in \texttt{other}$ such that they are equal, and the corresponding values `self[x]` and `other[y]` are also equal. The two dictionaries are equal if and only if all these conditions hold.

The tuple equality contract is identical to that of list equality, with the only difference that we ensure canonical equality if both `self` and `other` are tuples.

4.2.4 Modelling Equality Function Applications

To model equality function applications, we use the two different encodings defined in Section 3.3.

```
0  if (type(self) <: SubX()) {
1      <Preconditions of ==SubX>
2  } elseif (type(self) <: int()) {
3      <Preconditions of ==int>
4  } elseif (type(self) <: list()) {
5      <Preconditions of ==list>
6  } elseif (...) {
7      ...
8  } elseif (type(self) <: X()) {
9
                    acc(state(self))∧
                    (¬stateless(other) ⟹ acc(state(other)))
10 } elseif (...) {
11     ...
12 } else {
13
                    (¬stateless(self) ⟹ acc(state(self)))∧
                    (¬stateless(other) ⟹ acc(state(other)))
14 }
```

Figure 4.18: The merge function preconditions encoded as a conditional chain.

Merge Function-Based Equality Encoding

Consider the example in Figure 4.10 with the two classes `X` and `SubX`. Additionally, assume both `X.__eq__` and `SubX.__eq__` have contracts that capture their function definitions. Under these assumptions, the merge function for object equality follows the structure demonstrated in Figure 4.19. To reduce unnecessary complexity, only the relevant components of the contracts are shown in Figure 4.19, e.g., the contracts for integer equality (see Figure 4.13) or list equality (see Figure 4.17). To improve readability, the preconditions are presented as a chain of conditionals, rather than a single large conditional expression. Semantically, the same meaning is captured, but the conditional expression is less readable. In practice, the merge function uses a single conditional expression.

When the equality operator == is used in Python, the call to the corresponding `__eq__` function is redirected to the merge function.

Extended Function-Based Equality Encoding

In the extended function-based encoding, we use separate functions, one per type, and connect them via an added postcondition. We impose an additional restriction that the precondition of the overridden function is not weaker, otherwise, the added postcondition may not be well defined.

```
0 function equality_merged(self: Ref, i: Ref): Bool
1     decreases _
2     requires <Preconditions in Figure 4.18 as one conditional expression>
3     ensures <Postconditions of ==object>
4     ensures type(self) <: X() ⟹ <Postconditions of ==X>
5     ensures type(self) <: SubX() ⟹ <Postconditions of ==SubX>
6     ensures type(self) <: int() ⟹ <Postconditions of ==int>
7     ensures type(self) <: list() ⟹ <Postconditions of ==list>
8     ensures ...
```

Figure 4.19: The merge function for object equality.

```
0 function SubX___eq___extended(self: Ref, i: Ref): Ref
1     decreases _
2     requires acc(state(self))
3     requires ¬stateless(other) ⟹ acc(state(other))
4     ensures result == (self ==X+ other)
```

Figure 4.20: Encoding of the extended function for `SubX.__eq__`.

Consider the example in Figure 4.20, where the extended function for $==_{\texttt{SubX}}$ from Figure 4.10 is encoded. We add the postcondition `ensures result ==` $(\texttt{self} ==_{\texttt{X}^+} \texttt{other})$, since `SubX` <: `X`.

We encode each use of the equality operator == in Python as a call to the corresponding extended function of the type of the left-hand operand in Viper.

Chapter 5

Evaluation

We evaluate our implementation for verification performance, completeness, and usability, for each of which we highlight practical trends a user can expect.

5.1 Performance Evaluation

We conduct a performance analysis focusing primarily on tests that verify reflexivity, symmetry, transitivity, and scalability. The tests are run using Nagini's and Viper's default behaviour, which verifies multiple methods in parallel to simulate the conditions observed by a user. As previously mentioned, we proposed and implemented two different approaches to model equality function applications based on a merge or an extended function, as described in Section 4.2.4. All tests are verified with both Viper back-ends, namely Silicon and Carbon. Each test was executed five times using Nagini's benchmark mode on an XMG Fusion 15 equipped with an Intel i7-9750H CPU (2.6 GHz base, 4.5 GHz turbo) and 32 GB of RAM, running Ubuntu.

We illustrate the results of the performance tests in the table in Figure 5.1. Tests that crash, time out, or fail for any other reason are marked as such. The table consists of the following columns:

1. Test Id: A unique identifier
2. # C.: The number of classes in the test
3. Model: The modelling approach used for equality function applications; *E* represents the measurement obtained using the extended function and *M* with the merge function approach.
4. Time S.: The amount of time the test required for verification using Silicon

5. Time C.: The amount of time the test required for verification using Carbon
6. Verified: If the example was successfully verified using Silicon and/or Carbon
7. Exp. V.: The expected verification result for Silicon and Carbon

We include three basic tests (E_1, E_2, and E_3) to verify the properties of an equivalence relation. The first test only compares instances of the same class for equality, i.e., verifying reflexivity; symmetry and transitivity trivially follow. Each class in the second test can be equal to instances of itself and one other class, i.e., verifying reflexivity and symmetry; transitivity follows. In the third test, we extend this structure to three classes, for each of which it is verified that reflexivity, symmetry, and transitivity hold. Each of these three tests defines classes with between 0 and 3 primitive type fields, slowly increasing the complexity. We additionally include three basic tests (EF_1, EF_2, and EF_3) that are structured similarly to E_1, E_2, and E_3, but they are all expected to fail. However, only certain properties are expected to fail; behavioural subtyping, for example, holds in all tests. In each class in the first test, reflexivity should fail to verify. In the second test, symmetry ought to be violated. In the third test, transitivity should not hold. Thus, Nagini should reject all these tests. The fourth test (B_1) models how equality functions could be used in a small 'real world' scenario. It includes multiple classes, whose equality functions depend on each other. The final four tests (U_1 to U_4) focus on scalability; they measure how scalable both approaches for modelling function application are. Each class defined in this test has three primitive type fields, and each function is called from a client method. U_1 defines 8, U_2 15, U_3 26, and U_4 36 classes and client methods.

Consider the performance results in the table in Figure 5.1. All tests that are expected to verify succeed with Carbon, but fail with Silicon, as there exist some incompleteness issues in Silicon. On average, the extended function-based approach took 140.01 seconds, while the merge function-based approach took 141.82 seconds. Furthermore, consider the chart in Figure 5.2, where we plotted the relative percentage difference of the verification times of the merge function in comparison to the extended function-based approach. The values were calculated using the formula:

$$r_i := \frac{t_{\text{merge},i} - t_{\text{ext},i}}{t_{\text{ext},i}} \cdot 100$$

where $i \in \{1, 2, 3, 4\}$, and $t_{\text{merge},i}$ and $t_{\text{ext},i}$ denote the verification times of U_i using Carbon. As the examples increase in size, the relative percentage difference decreases. The values range from 1.17% to 6.14%. In particular, in the largest test U_4, the relative percentage difference is 1.17%. In general, the

Test Id	# C.	Model	Time S. [s]	Time C. [s]	Verified	Exp. V.
E_1	12	E	Out of Memory	17.9	No/Yes	Yes/Yes
E_1	12	M	Time out	16.9	No/Yes	Yes/Yes
EF_1	3	E	6.3	4.9	No/No	No/No
EF_1	3	M	Time out	4.3	No/No	No/No
E_2	8	E	Time out	29.7	No/Yes	Yes/Yes
E_2	8	M	Time out	29.3	No/Yes	Yes/Yes
EF_2	8	E	Time out	22.2	No/No	No/No
EF_2	8	M	Time out	23.5	No/No	No/No
E_3	6	E	Time out	51.7	No/Yes	Yes/Yes
E_3	6	M	Time out	45.2	No/Yes	Yes/Yes
EF_3	6	E	Time out	32.5	No/No	No/No
EF_3	6	M	Time out	32.0	No/No	No/No
B_1	3	E	Out of Memory	11.5	No/Yes	Yes/Yes
B_1	3	M	Time out	13.5	No/Yes	Yes/Yes
U_1	8	E	Time out	44.0	No/Yes	Yes/Yes
U_1	8	M	Time out	46.7	No/Yes	Yes/Yes
U_2	15	E	Time out	123.5	No/Yes	Yes/Yes
U_2	15	M	Time out	127.8	No/Yes	Yes/Yes
U_3	26	E	Time out	373.5	No/Yes	Yes/Yes
U_3	26	M	Time out	382.4	No/Yes	Yes/Yes
U_4	36	E	Time out	828.7	No/Yes	Yes/Yes
U_4	36	M	Time out	838.4	No/Yes	Yes/Yes

Figure 5.1: The performance tests result table.

data indicates that the extended function-based approach performs slightly better.

5.2 Completeness Evaluation

We perform a completeness analysis, which demonstrates the differences between the two equality function modelling approaches and the facts observed by callers about the invoked equality function.

We illustrate the completeness tests in the table in Figure 5.3. It consists of some of the columns from Figure 5.1 and the following additional ones:

1. Verified Carbon: If the test was successfully verified using Carbon
2. Verified Silicon: If the test was successfully verified using Silicon

The first test (D_1) is illustrated in Figures A.6 and A.7, where a class `A` with one field `f` and a client method `foo` are defined. We compare two instances `o1` and `o2` of static type `object` for equality in `foo`. Additionally,

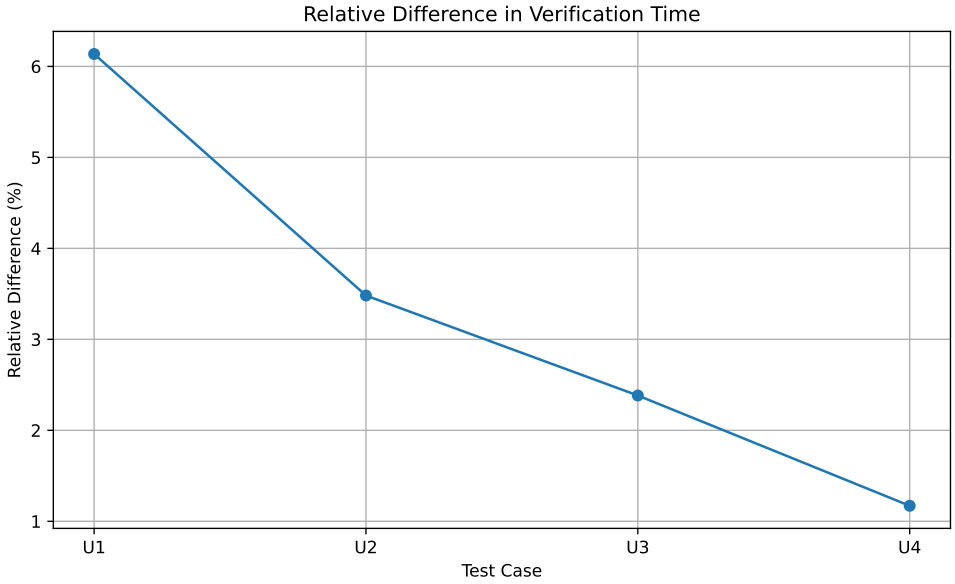

Figure 5.2: A chart illustrating the relative percentage difference in verification time using Carbon.

we add a precondition that requires `o1.f == o2.f`. Consequently, using the merge function-based approach, the verifier can prove the assertion `o1 == o2`, but not with the extended function-based approach. The second test (LR_1) demonstrated in Figure A.8, where a class and a client function `foo` is defined. It is used to test whether a caller learns reflexivity.

The third test (LS_1) defines two classes `E` and `F`, and a client function `foo`, where `e: E` and `f: F`. The client method calls `e == f` and `f == e`, and asserts that the results are equal. It is used to test whether a caller learns symmetry.

The fourth test (LT_1), defines three classes `G`, `H`, and `I`, and a client function `foo`, where `g: G`, `h: H`, and `i: I`. The client method calls `g == h` and `h == i`. If both results hold, we assert that `g == i` returns `true`. It is used to test whether a caller learns transitivity.

The test D_1 demonstrates that the merge function-based modelling approach is more complete, since it verifies using Carbon, but the extended function-based approach does not. From the test LR_1 and LS_1, we observe that when calling an equality function, we learn that it is reflexive and symmetric. Test LT_1 demonstrates that we cannot learn that an equality function is transitive as a caller; this is because we cannot obtain permissions to access an arbitrary object in Viper.

Test Id	Model	Verified Silicon	Verified Carbon
D_1	E	No	No
D_1	M	No	Yes
LR_1	E	Yes	Yes
LR_1	M	No	Yes
LS_1	E	No	Yes
LS_1	M	No	Yes
LT_1	E	No	No
LT_1	M	No	No

Figure 5.3: The completeness tests result table.

Test Id	Proportionality Factor
E_1	1.22
E_2	1.31
E_3	1.29
EF_1	1.35
EF_2	1.31
EF_3	1.32
B_1	1.27
U_1	1.40
U_2	1.40
U_3	1.40
D_1	1.50
LR_1	1.63
LS_1	1.35
LT_1	1.36
Avg	1.37

Figure 5.4: The usability tests result table.

5.3 Usability Evaluation

We perform a usability evaluation to measure the additional effort required by a user. This includes annotation and folding overhead (such as adding the decorator `@Transparent` or folding and unfolding predicates) as well as inserting additional `state` predicate preconditions. The analysis evaluates the total number of lines of code (LOC) with and without the additional overhead. In particular, for each test in the benchmark, we compute a proportionality factor that divides the total LOC by the LOC without the overhead.

Consider the table in Figure 5.4. The proportionality factor is 1.37 on average, which highlights that the additional overhead for the user is not significant

and acceptable in practice.

Chapter 6

Conclusion

This thesis addressed the modular verification of user-defined equality functions in the context of object-oriented programming languages like Python. In particular, it proposes a technique to establish that equality defined for subtypes is consistent with the equality semantics of their supertypes.

We presented a modular framework for specifying and verifying user-defined, overridable equality functions in Python. We demonstrated that this framework supports modular verification of equality functions across distinct types, ensuring that they satisfy the properties of an equivalence relation, i.e., reflexivity, symmetry, and transitivity[1]. Furthermore, it enables the use of equality functions in specifications. Our evaluation shows that a verification can be completed in a reasonable amount of time and with an acceptable annotation overhead for the user.

This work demonstrates that modular verification of equality is practical in object-oriented languages involving subtyping. Additionally, as the approach is based on permission logic, it extends to concurrent settings.

A caller cannot learn that an equality function is transitive, as we cannot obtain permissions to access an arbitrary object in Viper.

Future work could extend this framework to support other dunder methods, such as containment checks, as in Python, they implicitly use the equality operator. In particular, this could be done by adapting the `__contains__` definitions of built-in container types.

In summary, this thesis establishes a foundation to enable modular reasoning about user-defined equality functions in the context of object-oriented programming languages.

[1]This sentence was revised using ChatGPT 4o.

Appendix A

Appendix

A.1 Figures for Equality Contracts

The equality contracts for `sets`, `dicts`, and `tuples` are shown in the Figures A.1, A.2 and A.3.

A.2 Alternative Floating-Point Number Contracts

A.3 Tests

```
0  function set___eq__(self: Ref, other: Ref): Bool
1      decreases _
2      requires ¬stateless(self) ⟹ acc(state(self))
3      requires ¬stateless(other) ⟹ acc(state(other))
4
5      ensures type(self) <: set(...) ∧ type(other) <: set(...)  ⟹
6          result ==
7          (¬stateless(self) ⟹ unfolding acc(state(self)) in (
8              (¬stateless(other) ⟹ unfolding acc(state(other)) in (
9
```

$$\mathcal{L}(\texttt{self}) = \mathcal{L}(\texttt{other}) \wedge \bigwedge_{\texttt{x} \in \texttt{self}} \left(\bigvee_{y \in \texttt{other}} \texttt{x} ==_{\texttt{object}} \texttt{y} \right)$$

```
10             ))
11         ))
```

Figure A.1: Set equality contract encoded as a Viper function.

```
0  function dict___eq__(self: Ref, other: Ref): Bool
1      decreases _
2      requires ¬stateless(self) ⟹ acc(state(self))
3      requires ¬stateless(other) ⟹ acc(state(other))
4
5      ensures type(self) <: dict(...) ∧ type(other) <: dict(...)
       ⟹
6          result ==
7          (¬stateless(self) ⟹ unfolding acc(state(self)) in (
8              (¬stateless(other) ⟹ unfolding acc(state(other)) in (
9
                   ℒ(self) = ℒ(other) ∧
                   ⋀_{x∈self} ( ⋁_{y∈other} x ==_object y ∧ self[x] ==_object other[y] )
10             ))
11         ))
```

Figure A.2: Dictionary equality contract encoded as a Viper function.

```
0  function tuple___eq__(self: Ref, other: Ref): Bool
1      decreases _
2      requires ¬stateless(self) ⟹ acc(state(self))
3      requires ¬stateless(other) ⟹ acc(state(other))
4
5      ensures type(self) <: tuple(...) ∧ type(other) <: tuple(...)
       ⟹
6          result ==
7          (¬stateless(self) ⟹ unfolding acc(state(self)) in (
8              (¬stateless(other) ⟹ unfolding acc(state(other)) in (
9
                       ℒ(self) == ℒ(other) ∧
                       ⋀_{x∈[0, ℒ(self))} (self[x] ==_object other[x])
10             ))
11         ))
```

Figure A.3: Tuple equality contract encoded as a Viper function.

```
function float___eq__(self: Ref, other: Ref): Bool
    decreases _
    ensures type(self) <: float() ∧ type(other) <: float() ⟹
        isNaN(self) ∨ isNaN(other) ⟹ ¬result
    ensures type(other) <: float() ∧ type(other) <: float() ⟹
        (self = −∞ ∧ other = −∞) ∨ (self = +∞ ∧ other = +∞) ⟹ result
    ensures type(self) <: float() ∧ type(other) <: float() ⟹
        (self = −∞ ∧ ¬(other = −∞)) ∨
        (self = +∞ ∧ ¬(other = +∞)) ∨
        (other = −∞ ∧ ¬(self = −∞)) ∨
        (other = +∞ ∧ ¬(self = +∞)) ⟹ ¬result
    ensures type(self) <: float() ∧ type(other) <: float() ⟹
        (¬isNaN(self) ∧ ¬isNaN(other)∧
        self ≠ −∞ ∧ self ≠ +∞ ∧ other ≠ −∞ ∧ other ≠ +∞) ⟹
        result = (unbox(self) = unbox(other))
```

Figure A.4: Contract for floating-point numbers encoded as reals.

```
function float___eq__(self: Ref, other: Ref): Bool
    decreases _
    ensures type(self) <: float() ∧ type(other) <: float() ⟹
        result == IEEE(unbox(self), unbox(other))
```

Figure A.5: Contract for floating-point numbers encoded with IEEE.

```
class A:
    def __init__(self, i: int) -> None:
        self.i: int = i
        Fold(self.state())
        Ensures(self.state())

    @Pure
    def __eq__(self, other: object) -> bool:
        Requires(state_pred(self))
        Requires(Implies(not Stateless(other), state_pred(other)))
        Ensures(Implies(
            isinstance(other, A), Unfolding(self.state(),
                Unfolding(state_pred(other),
                    Result() == (self.i == cast(A, other).i)
                )
            )
        ))
        if self is other:
            return True
        elif isinstance(other, A):
            return Unfolding(
                self.state(),
                Unfolding(
                    state_pred(other),
                    self.i == cast(A, other).i
                )
            )
        return False

    @Predicate
    def state(self) -> bool:
        return Wildcard(self.i)
```

Figure A.6: The class definition of `A` for the test D_1.

```
def foo(o1: object, o2: object) -> int:
    Requires(isinstance(o1, A))
    Requires(isinstance(o2, A))
    Requires(state_pred(o1))
    Requires(state_pred(o2))
    Requires(Unfolding(state_pred(o1),
        Unfolding(state_pred(o2),
        cast(A, o1).i == cast(A, o2).i))
    )
    Ensures(state_pred(o1))
    Ensures(state_pred(o2))

    Unfold(state_pred(o1))
    Unfold(state_pred(o2))
    assert o1 == o2
    Fold(state_pred(o1))
    Fold(state_pred(o2))
    return 0
```

Figure A.7: The client method `foo` for the test D_1.

```
class A:
    def __init__(self, i: int, s: str, b: bool) -> None:
        self.i: int = i
        self.s: str = s
        self.b: bool = b
        Fold(state_pred(self))
        Ensures(state_pred(self))

    @Pure
    def __eq__(self, other: object) -> bool:
        Requires(state_pred(self))
        Requires(Implies(not Stateless(other), state_pred(other)))
        if self is other:
            return True
        return False

    @Predicate
    def state(self) -> bool:
        return Acc(self.i) and Acc(self.s) and Acc(self.b)

def foo(a: A, b: A) -> int:
    Requires(state_pred(a))
    Requires(state_pred(b))
    Ensures(state_pred(a))
    Ensures(state_pred(b))

    Unfold(state_pred(a))
    Unfold(state_pred(b))
    res: bool = a == b
    if res:
        assert a is b
    Fold(state_pred(a))
    Fold(state_pred(b))
    return 0
```

Figure A.8: The test LR_1 demonstrates that a caller learns reflexivity.

Bibliography

[1] David R. Cok. Openjml: Jml for java 7 by extending openjdk. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 472–479, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[2] Marco Eilers. *Modular Specification and Verification of Security Properties for Mainstream Languages*. Doctoral thesis, ETH Zurich, Zurich, 2022.

[3] Marco Eilers and Peter Müller. Nagini: A static verifier for python. In Hana Chockler and Georg Weissenbacher, editors, *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*, pages 596–603, Cham, 2018. Springer International Publishing.

[4] Jukka Lehtosalo et al. Mypy - optional static typing for python.

[5] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[6] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[7] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[8] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 247–258. ACM, 2005.

[9] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, 2012.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten schriftlichen Arbeit. Eine der folgenden drei Optionen ist in Absprache mit der verantwortlichen Betreuungsperson verbindlich auszuwählen:

○ Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Es wurden keine Technologien der generativen künstlichen Intelligenz[1] verwendet.

⊗ Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Als Hilfsmittel wurden Technologien der generativen künstlichen Intelligenz[2] verwendet und gekennzeichnet.

○ Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Als Hilfsmittel wurden Technologien der generativen künstlichen Intelligenz[3] verwendet. Der Einsatz wurde, in Absprache mit der Betreuungsperson, nicht gekennzeichnet.

Titel der Arbeit:

ENABLING OBJECT EQUALITY REASONING FOR PYTHON

Verfasst von:
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
GREUTMANN	MICHA

Ich bestätige mit meiner Unterschrift:
- Ich habe mich an die Regeln des «Zitierleitfadens» gehalten.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu und vollständig dokumentiert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Eigenständigkeit überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Zürich, 25.05.2025	

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie grundsätzlich gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

[1] z. B. ChatGPT, DALL E 2, Google Bard
[2] z. B. ChatGPT, DALL E 2, Google Bard
[3] z. B. ChatGPT, DALL E 2, Google Bard