Modular Reasoning about Object Relations

Micha Greutmann, Marco Eilers(✉), and Peter Müller

Department of Computer Science, ETH Zurich, Zurich, Switzerland
mgreutmann@ethz.ch, {marco.eilers, peter.mueller}@inf.ethz.ch

Abstract. In imperative and object-oriented languages, programmers often define relations such as equality or orderings between instances of structs or classes. These *object relations* must satisfy well-known algebraic properties, such as reflexivity, transitivity, etc. Violations may cause standard library components, such as collections, to behave incorrectly. Crucially, these properties must hold across types, for instance, when comparing instances of different classes. However, studies show that they are commonly violated, and existing techniques for verifying them are non-modular: they give no guarantees if any types are present at runtime that were not known at verification time.
This paper presents the first modular verification technique for object relations. Our key idea is to express algebraic properties in a novel, expressive normal form that allows us to statically identify, for each type T and relation R, a set of other types that are relevant for the correctness of R in T. We prove the intended algebraic properties of R between T and each type in this set. This approach is generic: it applies to different relations (such as equality and orderings), to various imperative and object-oriented languages, and to multiple program logics. We show how it can be used in standard deductive verification tools by encoding the relevant proof obligations using a simple source-to-source rewriting. We implement our technique for the equality relation in the Python verifier Nagini and evaluate it on challenging benchmarks using both Nagini and Dafny, demonstrating its practical applicability and effectiveness.

Keywords: Deductive verification, modularity, relations, equality

1 Introduction

In imperative languages, it is common to define binary relations like equality or orderings between instances of classes or structs. Equality is particularly central in object-oriented languages, where each class may define its own equality method (e.g., `equals` in Java or `__eq__` in Python) that can be (and often is) overridden in subtypes. These methods are critical for common use cases: for instance, checking whether a collection contains some element `e` is usually performed via `e.equals(e')` for some element `e'` in the collection.

As a result, the correctness of many operations on collections rely on `equals` defining an equivalence relation, i.e., a relation that is reflexive, symmetric, and

```
class Object {
  virtual boolean equals(Object o)
  { ... }
}

class Point extends Object {
  private int x;
  private int y;

  override boolean equals(Object o)
  {
    if (o instanceof Point) {
      Point p = ((Point) o);
      return this.x == p.x &&
             this.y == p.y;
    }
    return false;
  }
}
```

```
class ColorPoint extends Point {
  private Color c;

  override boolean equals(Object o)
  {
    if (o instanceof ColorPoint) {
      ColorPoint p = ((ColorPoint) o);
      return this.x == p.x &&
             this.y == p.y &&
             this.c.equals(p.c);
    }
    return false;
  }
}
```

Fig. 1. Two classes that implement the equals relation. Each implementation is correct when comparing instances of the same class, but together, they violate symmetry when comparing objects across the subtype hierarchy. Adapted from [28].

transitive [28]. Similar requirements exist for other relations, for example, orderings defined via <-operators must satisfy irreflexivity, asymmetry, and transitivity. We call these properties of relations *algebraic properties*.

Violations of such properties are easy to introduce, as shown in Fig. 1, where two classes Point and ColorPoint each define custom versions of equality. Individually (i.e., when only comparing instances of the same class), these definitions satisfy all constraints. However, when checking the equality of a Point instance p and a ColorPoint instance cp that has the same coordinates, p.equals(cp) will return true (since o instanceof Point is true if o has type Point or a *subtype* of Point), but cp.equals(p) returns false, that is, symmetry is violated. The fact that both implementations are correct in isolation but not when combined demonstrates the difficulty of determining the correctness of a given implementation *modularly*, i.e., without knowing all possible implementations of a given relation. Modularity is crucial for example when implementing library code, which may be used in contexts unknown to the library implementer. Empirical studies have found frequent violations of algebraic properties in real-world code and even in code patterns suggested by textbooks [28].

The basic problems of modularly specifying and reasoning about class hierarchies and calls with dynamic binding have had well-established solutions since the early 2000s, like behavioral subtyping [22] or specification inheritance [10], implemented in tools based on JML [18] like KeY [1] and others like Spec# [2]. However, these solutions only show how to reason about individual method calls inside a subtype hierarchy; surprisingly, the problem of proving crucial algebraic properties of relations used in virtually every program remains unsolved.

While there are some existing tools that verify algebraic properties, in particular of equals methods in Java [29,30], these assume that all types are known at

verification time and simply check properties between all pairs of known types. As a result, they provide no guarantees if any additional types are present at runtime. A different approach is suggested by the Java verifier OpenJML [8], which models these properties for equals indirectly: It allows implementers of new classes to define a way of abstracting each object to a single abstract value. The equals method must then return true for two objects if and only if their abstract values are identical. If this property holds, then the algebraic properties of equality on abstract values also hold for equals. However, this approach is inherently non-modular, since there is no mechanism to ensure that two completely unrelated classes do not abstract to the same value.

In this paper, we present the first modular verification methodology for algebraic properties of binary object relations like orderings and equality. Our key idea is a novel, expressive normal form for the specification of algebraic properties. This normal form allows us to determine, for the implementation of a relation R in a type T, a set of types C_T that are relevant for the validity of R's algebraic properties. In particular, each instance for which T's implementation of R yields true has a type in C_T. Our normal form allows us to phrase all necessary proof obligations for the implementation of R in T such that they concern only T's own implementation and the specifications of the types in C_T. These types are all known to the implementer of T, which allows us to discharge the proof obligations modularly. For instance, the proof obligations for Points.equals can be proved using only the implementation and specification of this method. Subtype implementations that violate the algebraic properties (such as the one in ColorPoint) are ruled out by the standard behavioral subtyping [22] requirement.

Our approach is sound, that is, the validity of all modular proof obligations implies that the specified algebraic properties hold globally. In particular, unlike with existing techniques, extending a program with additional (modularly-verified) types cannot lead to violations of the algebraic properties.

Our approach does not prescribe any specific form for relation implementations, and allows instances of different types to be related. As a result, it is sufficiently flexible to accommodate all typical coding patterns for standard object relations.

To summarize, we make four core contributions:

- We identify a normal form for algebraic properties of object relations, which allows us to extract proof obligations that can be proved modularly (Sec. 2). Unlike existing techniques, our technique ensures that verified algebraic properties hold even when the program is extended with additional verified types.
- We present our technique as a collection of proof obligations that can be represented in any verification framework that supports pure functions, for instance, dynamic frames [16] as in Dafny [19] or implicit dynamic frames [31] as in Viper [23], in the form of a source-to-source encoding (Sec. 3).
- We prove the soundness of our technique (Sec. 4).
- We demonstrate the practicality of our technique by implementing it for equality in Nagini [12], a verifier for statically- and nominally-typed Python.

Our evaluation based on our Nagini implementation and manually-encoded examples in Dafny shows that our techniques supports a diverse set of realistic use cases and design patterns (Sec. 5).

2 Verification Technique

In this section, we explain the modularity problem outlined in the introduction in detail and present our solution. Our technique applies to languages that (1) have nominal subtyping (via traits, interfaces, or subclassing), and (2) implement binary relations as boolean methods taking two object or struct arguments, such that calls are dynamically resolved based on the type of the first argument (the receiver). This is the case in all object-oriented languages (like Java or C#), some imperative languages (like Dafny), and multi-paradigm languages with object-oriented aspects (like Python or Scala). Relations implemented as non-boolean methods, such as Java's compare methods, are not supported. We will show most examples in a Java-like language and call instances of types "objects". We use the terms *subtype* and *strict subtype* to mean the reflexive and non-reflexive relations, resp. Similarly, we say that an object has type T or *exact* type T if its dynamic type is a subtype of T or exactly T, resp.

2.1 Modularity Problem

The implementation of a binary relation R is potentially split over several methods, for instance, all overrides of Java's Object.equals method. For simplicity, we treat an inherited method as if it was overridden with the identical implementation, such that each subtype of the type introducing the relation has its own implementation. Since we assume that method calls are resolved dynamically based on the type of the receiver, $R(o_1, o_2)$ is defined by the method implementing R in o_1's dynamic type T_1. We denote this implementation as $I^R_{T_1}$ and expect it to be *pure*. That is, it yields a deterministic value and has no side effects on outside state. For verification, pure methods can be encoded as mathematical functions and used in specifications and proofs [9].

Using this notation, the overall relation R, when defined by implementations in a set of types $\{T_1, \ldots, T_n\}$, is defined as follows:

$$R(o_1, o_2) = \begin{cases} I^R_{T_1}(o_1, o_2) & \text{if } \mathit{type}(o_1) = T_1 \\ I^R_{T_2}(o_1, o_2) & \text{if } \mathit{type}(o_1) = T_2 \\ \ldots \end{cases} \tag{1}$$

Algebraic properties of the overall relation, such as reflexivity, symmetry, or transitivity, must be ensured by all implementations. Some algebraic properties like (ir)reflexivity can be established by proving each implementation *individually*: R is (ir)reflexive if and only if $I^R_{T_i}$ is (ir)reflexive for every type T_i. Hence, one can verify each implementation modularly. In particular, when extending a program, it is sufficient to prove (ir)reflexivity for the newly-added types.

```
class SpecialPoint extends Point {
  ...
  // does not override equals
}

class SelectivePoint extends Point {
  override boolean equals(Object o) {
    if (o.getClass() == Point.class ||
      o.getClass() == SelectivePoint.class) {
      Point p = (Point) o;
      return this.x == p.x && this.y == p.y;
    }
    return false;
  }
}
```

Fig. 2. Two classes that each satisfy symmetry of equality when compared to themselves and instances of Point, but not when compared to each other.

However, other algebraic properties like symmetry and transitivity involve several instances of potentially-different types and, thus, need to be ensured *collectively* by the implementations of a relation in *different* types. For example, symmetry of the equality relation (which we will abbreviate as *eq* when convenient) requires that $I^{eq}_{T_1}(o_1, o_2) = I^{eq}_{T_2}(o_2, o_1)$, where T_1 is the type of o_1 and T_2 is the type of o_2. As illustrated by the example from Fig. 1, the implementer of one type (in our example, Point) is not necessarily aware of the implementations of the other types (ColorPoint), which makes modular reasoning challenging.

In the simple example from Fig. 1, one could impose the responsibility for checking symmetry on the implementer of ColorPoint, who is aware of both types. That is, one could prove that for any two objects o_1 and o_2 of type Point or ColorPoint, o_1.equals(o_2) always returns the same result as o_2.equals(o_1).

However, proving algebraic properties for instances of known types (e.g. a class's supertypes and types from imported packages) is generally insufficient, as the example in Fig. 2 demonstrates. Class SpecialPoint extends Point and inherits its implementation of equals, whereas class SelectivePoint also extends Point and overrides equals such that a SelectivePoint considers itself equal only to exact instances of Point and SelectivePoint that have the same coordinates. Since there is no dependency between SpecialPoint and SelectivePoint, they may be created independently by different implementers. The equals method of SpecialPoint satisfies symmetry for comparisons among SpecialPoint and Point instances, and analogous for SelectivePoint. However, symmetry is violated when comparing a SpecialPoint to a SelectivePoint in a client that imports both!

The challenge, thus, is: How can we verify algebraic properties modularly, such that the verified properties remain valid when adding types to the program?

2.2 Approach

In the following, we provide an overview of our solution. We will discuss details of our technique in Sec. 3 and its soundness in Sec. 4.

$$
\begin{array}{lrl}
(AlgProp) & AP & ::= R(o_1, o_2) \Rightarrow P \\
(Prop) & P & ::= \neg P \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \mid TR \mid e \\
(TypedRelation) & TR & ::= R'(o_1, x) \mid R'(o_2, x) \\
(Expressions) & e & ::= c \mid x \mid e \oplus e \text{ where } c \in \mathbb{Z} \text{ and } \oplus \in \{+, -, \times, =, <\} \\
 & & \quad \mid o_1.f(\overline{e}) \mid o_2.f(\overline{e})
\end{array}
$$

Fig. 3. Grammar for supported algebraic properties. $\overline{e}$ denotes a (possibly empty) list of expressions e. R' denotes an arbitrary relation including R.

Table 1. Algebraic properties of object relations that can be expressed in the normal form we support. Column n shows how many objects are related by the property; for example, transitivity of equality is a 3-relational algebraic property. All o_i are universally quantified.

R	Name	n	Property
$=$	Symmetry	2	$o_1 = o_2 \Rightarrow o_2 = o_1$
$=$	Transitivity	3	$o_1 = o_2 \Rightarrow (o_2 = o_3 \Rightarrow o_1 = o_3)$
$=$	Hash-Consistency	2	$o_1 = o_2 \Rightarrow o_1.hash() = o_2.hash()$
$=$	Contains-Consistency	3	$o_1 = o_2 \Rightarrow (o_1 \ contains \ o_3 \Rightarrow o_2 \ contains \ o_3)$
$<$	Asymmetry	2	$o_1 < o_2 \Rightarrow \neg(o_2 < o_1)$
$<$	Transitivity	3	$o_1 < o_2 \Rightarrow (o_2 < o_3 \Rightarrow o_1 < o_3)$
$<$	Converse	2	$o_1 < o_2 \Rightarrow o_2 > o_1$
contains	Equals-Consistency	3	$o_1 \ contains \ o_2 \Rightarrow (o_2 = o_3 \Rightarrow o_1 \ contains \ o_3)$

Normal form specifications. The key observation enabling our approach is that the algebraic properties of many important object relations can be expressed in the form $R(o_1, o_2) \Rightarrow P$, where P is a formula expressed using the grammar in Fig. 3. In particular, P may mention R or other relations R', the arguments of R (o_1 and o_2), free variables x denoting objects, as well as other mathematical expressions and functions. We assume that the algebraic properties of an object relation are specified in this normal form in the type that introduces the object relation, such that all implementations of the relation have them in scope.

All standard algebraic properties of equality, orderings and containment *except* reflexivity can be expressed in this form, as shown in Table 1. As stated before, modular verification of reflexivity is straightforward and does not require our technique. Since property P may refer to other relations, it can express properties such as consistency between less-than and greater-than; referring to functions enables expressing consistency between the equals relation and hash functions, and free variables allow us to express ternary relations like transitivity.

Constraining sets. Our normal form has two crucial properties that enable modular verification. The first is that an algebraic property of $R(o_1, o_2)$ holds trivially if $R(o_1, o_2)$ is false. Consequently, it is sufficient to prove properties of the implementation in o_1's type, $I^R_{T_1}$, only when it returns true. In these cases, we may safely assume that the implementation $I^R_{T_1}$ has sufficiently-precise type information for the other argument o_2, such that we can use the specification of

that type to reason about properties of that object. This assumption is justified because an implementation cannot meaningfully relate its arguments if it does not even know their types. In our Point and ColorPoint examples from Fig. 1, the type of o_2 is known due to the instanceof expression (we will discuss subtyping below), and in SelectivePoint from Fig. 2 due to the comparison of class objects.

More precisely, we assume that for each object relation R and type T, we are given a set of types C_T^R that overapproximates the types of o_2 for which R's implementation in T yields true. We call this set the *constraining set of R for T*:

Definition 1. *A set of types C_T^R is a* constraining set *of object relation R for type T iff for all instances o_1 of the exact type T and all objects o_2, o_2 has some type $T_2 \in C_T^R$ in case $I_T^R(o_1, o_2)$ yields* true*.*

In the following, we omit the relation R when it is clear from context and simply say *constraining set for T*, C_T. Note that the type T_2 is not necessarily o_2's exact type. Hence in Java, { Object } is always a valid constraining set, but since it is imprecise, it will not be useful for verification. For our examples so far, { Point } is a more precise constraining set.

We assume in the following that the constraining set C_T is declared by the programmer and impose a proof obligation that o_2 has a type in C_T whenever $I_T^R(o_1, o_2)$ returns true. However, constraining sets could also be inferred automatically, as we discuss in Sec. 3.2. All types in C_T are in scope of type T and, thus, available during the modular verification of T.

Modular verification. To prove an algebraic property, we need to show for each implementation I_T^R that $I_T^R(o_1, o_2) \Rightarrow P$ holds. This implication can be proved modularly due to the second crucial property of our normal form: It restricts the occurrence of relations R' in P such that the first argument is either o_1 or o_2 (see Fig. 3); in particular, a free variable x may occur only as the *second* argument and, thus, does not determine which implementation of R' applies. Consequently, the type of the first argument is either o_1's type T or a type in the constraining set for T (unless $I_T^R(o_1, o_2)$ yields false, in which case the desired implication holds trivially). Either way, this type is available during modular verification, such that one can use its specification to prove that P holds. Similarly, our normal form restricts the receiver of a function application to also be either o_1 or o_2 and, thus, the specification of the function is also available during modular verification.

Generally, we can express any algebraic property AP as a logical formula that incorporates all known guarantees about the relations and functions it refers to according to the statically-known types. This transformation is shown in Fig. 4. $[\![AP]\!]_{T'}^{T}$ denotes the proof obligation for AP for a given type T under the assumption that o_2 has the type T'. The transformation exchanges applications of relations and functions with invocations of the methods that implement them. These methods must be pure (terminating, deterministic, and side-effect free) and, thus, may occur in proof obligations. Since the exact type of o_1 is known to be T, the transformation can determine the *exact* methods for relations and

$$
\begin{array}{ll}
[\![R(o_1, o_2) \Rightarrow P]\!]^T_{T'} & = \mathit{type}(o_1) = T \wedge I^R_T(o_1, o_2) \Rightarrow [\![P]\!]^T_{T'} \\
[\![P_1 \wedge P_2]\!]^T_{T'} & = [\![P_1]\!]^T_{T'} \wedge [\![P_2]\!]^T_{T'} \\
[\![P_1 \Rightarrow P_2]\!]^T_{T'} & = [\![P_1]\!]^T_{T'} \Rightarrow [\![P_2]\!]^T_{T'} \\
[\![\neg P]\!]^T_{T'} & = \neg [\![P]\!]^T_{T'} \\
[\![R'(o_1, x)]\!]^T_{T'} & = I^{R'}_T(o_1, x) \\
[\![R'(o_2, x)]\!]^T_{T'} & = I^{R'}_{\leq T'}(o_2, x) \\
[\![e_1 \oplus e_2]\!]^T_{T'} & = [\![e_1]\!]^T_{T'} \oplus [\![e_2]\!]^T_{T'} \\
[\![o_1.f(\overline{e})]\!]^T_{T'} & = I^f_T(o_1, \overline{[\![e]\!]^T_{T'}}) \\
[\![o_2.f(\overline{e})]\!]^T_{T'} & = I^f_{\leq T'}(o_2, \overline{[\![e]\!]^T_{T'}}) \\
[\![c]\!]^T_{T'} & = c \\
[\![x]\!]^T_{T'} & = x
\end{array}
$$

Fig. 4. Transformation of algebraic properties in our normal form that incorporates type information. $I^{R'}_{\leq T'}$ represents the implementations of R' in T' and all of its subtypes, and $I^f_{\leq T'}$ analogously for function f.

functions invoked with o_1 as receiver. In contrast, o_2's dynamic type may be a subtype of T' and, hence, we have to consider method overrides in subtypes of T'. Consequently, an invocation of a relation R' on o_2 is transformed to an invocation of a *virtual method* $I^{R'}_{\leq T'}$, which represents the implementations of R' in T' and all of its subtypes (and analogously for functions).

To reason about such virtual methods, we equip all method implementations with specifications in the form of pre- and postconditions in some assertion language. We enforce the standard requirement that all subtypes in a program are *behavioral subtypes* [22], that is, the subtype implementations satisfy the specifications of their supertypes, in particular:

- the precondition of an overriding method must be identical or *weaker* than that of the method it overrides, and
- the postcondition of an overriding method must be identical or *stronger* than that of the method it overrides.

Behavioral subtyping allows one to reason soundly about dynamically-bound method calls (such as $I^{R'}_{\leq T'}$) based on the specification of the static type T' of the receiver, even when an override may be invoked at runtime. In particular, the implementation invoked at runtime need not be known at the call site. As a result, our transformation has the following key feature: For any property P, the transformed version $[\![P]\!]^T_{T'}$ is a logical formula whose truth value is fully determined by the *implementation of* T and the *specification of* T'. $[\![P]\!]^T_{T'}$ implies that P holds in all programs containing T, T', *and any additional types* that satisfy behavioral subtyping.

Consider proving symmetry for the equals relation in class Point. It yields true only if the parameter other has type Point. Applying the transformation from Fig. 4 for symmetry (see Table 1) yields $\mathit{type}(o_1) = \mathsf{Point} \wedge I^{eq}_{\mathsf{Point}}(o_1, o_2) \Rightarrow I^{eq}_{\leq \mathsf{Point}}(o_2, o_1)$. We can prove it modularly using a suitable method specification, which we will discuss later.

```
class Structural:
    def __init__(self):
        self.val = 0

    def __eq__(self, other: object) -> bool:
        if hasattr(other, 'val'):
            return other.val == self.val
        return False
```

Fig. 5. Equality definition in Python that is not modularly correct. Since instances of arbitrary, unknown classes could be claimed to be equal, symmetry and transitivity of the equality relation cannot be established modularly.

We can now define a notion of *modular correctness* of a type:

Definition 2. *Given a type T that implements relation R with the implementation I^R_T, T is* modularly correct *w.r.t. an algebraic property in the normal form $R(o_1, o_2) \Rightarrow P$ iff it has a constraining set C_T, and for any object o_1 of exact type T, if $I^R_T(o_1, o_2)$ returns* true*, then there exists a type $T' \in C_T$ such that o_2 has type T' and $[\![P]\!]^T_{T'}$ holds.*

Proving modular correctness of each type ensures that the algebraic properties of all relations in a program hold globally (see Sec. 4).

2.3 Examples

We illustrate our approach on several relevant examples.

Point hierarchy: Recall the proof obligation for I^{eq}_{Point} from the previous subsection: $type(o_1) = \mathsf{Point} \wedge I^{eq}_{\mathsf{Point}}(o_1, o_2) \Rightarrow I^{eq}_{\leq \mathsf{Point}}(o_2, o_1)$. To prove it modularly, we equip Point.equals with the postcondition result $\Leftrightarrow$ other : Point $\wedge$ this.x = other.x $\wedge$ this.y = other.y, where result refers to the result of the method and $x : T$ expresses that x has type T. The implementation $I^{eq}_{\mathsf{Point}}(o_1, o_2)$ yields true only if o_2 has type Point and o_1 and o_2 have identical x and y values. Using the postcondition, we can prove that $I^{eq}_{\leq \mathsf{Point}}(o_2, o_1)$ must also yield true.

This postcondition allows inheriting the same implementation in SpecialPoint but rules out the implementations in ColorPoint and SelectivePoint, as they do not satisfy the requirements of behavioral subtyping. Any postcondition that can be proved for those implementations of equals would not be sufficient to prove the above obligation for Point.equals. This illustrates that our approach eliminates the violations of symmetry observed in ColorPoint and SelectivePoint.

Structural comparisons: The __eq__ method of the Python class Structural in Fig. 5 (which could also be written in Java and similar languages using reflection) defines self to be equal to any other object that has a val field containing the same value as the one in self. This implementation is correctly rejected by our technique. Since the type of other is not known, we have no information about the behavior of its equality method, such that symmetry or transitivity cannot be

guaranteed modularly. Technically, verification of this example fails because the only useful constraining set for Structural is { object } (no other type is known to be a supertype of the dynamic type of other), but any reasonable specification of object.__eq__ will be too weak to discharge our proof obligation.

Reference equality: In languages like Java, the default implementation of equality in the Object class is reference equality. We can verify this default implementation by choosing the constraining set {Object}, which yields the proof obligation $type(o_1) = \mathsf{Object} \wedge I^{eq}_{\mathsf{Object}}(o_1, o_2) \Rightarrow I^{eq}_{\leq \mathsf{Object}}(o_2, o_1)$. To prove it, we equip Object.equals with the postcondition this.getClass() = Object ⇒ (result ⇔ this = other). The fact that $I^{eq}_{\mathsf{Object}}(o_1, o_2)$ yields true implies that o_1 and o_2 are identical (reference-equal) and, thus, o_2's exact type is Object. This allows us to apply the postcondition for $I^{eq}_{\leq \mathsf{Object}}(o_2, o_1)$ to show the proof obligation.

Note that this postcondition constrains only the implementation of equals in Object; it is trivially true if the receiver's type is not exactly Object. Thus, overrides may use different equality criteria without violating behavioral subtyping.

Dependent implementations: Our proof obligations do not constrain how an implementation achieves the intended behavior. In particular, it is possible to implement one relation in terms of another, e.g., a less-than relation by negating the result of a call to an existing greater-or-equal implementation. Given a reasonably precise functional specification for greater-or-equal, all proof obligations (e.g., for transitivity) of less-than can be discharged based on that specification.

3 Encoding

In this section, we explain how to verify the modular correctness of implementations of object relations. We present the concrete proof obligations as a source-to-source encoding. Correctness of the resulting program implies that the relations' algebraic properties hold. It can be checked with any verification tool offering some basic features. In particular, we assume that the used verification tool supports *pure methods*, i.e., program methods that are deterministic, terminating, and side-effect free. Pure methods are widely supported in existing program logics, such as dynamic frames [16] and implicit dynamic frames [31]) and tools (such as JML-based verifiers [18], Dafny [19], Viper-based verifiers [23], and many others). We assume for simplicity that methods do not have preconditions (since methods implementing object relations may typically be invoked with arbitrary arguments), but our implementation supports them.

We will first present the encoding of modular correctness checks, show some possible variations that enable a higher degree of automation, and discuss how the encoding can be used to infer constraining sets. Subsequently, we will show an encoding of virtual pure methods, which are crucial to our approach, into a language with standard pure methods as supported by existing tools.

```
// pure
boolean check_C^T(Object o)
  ensures result ⇒
          (o : T_1 ∨ ··· ∨ o : T_m)
{
  I_T^R
}
```

```
void check_AP^T(Object o_2, ..., Object o_n)
{
  assume type(this) == T
  assume I_T^R(this, o_2)

  if (o_2 : T_1 ∧ [[P]]^T_{T_1} ∨
      o_2 : T_2 ∧ [[P]]^T_{T_2} ∨
      ...
      o_2 : T_m ∧ [[P]]^T_{T_m} ) {
    return
  } else {
    assert false
  }
}
```

Fig. 6. Encoding of proof obligations for object relations: validity of constraining sets (left) and validity of algebraic properties (right).

3.1 Verifying Algebraic Properties

As explained in Sec. 2.2, we impose two kinds of proof obligations for each object relation R: (1) We check for each type T implementing R that the declared constraining set C_T is valid, and (2) we verify that each implementation of R satisfies the declared algebraic properties. Fig. 6 shows the encoding of both kinds proof obligations for an n-relational algebraic property, assuming $C_T = \{T_1, \ldots, T_m\}$. Our encoding does not show how to prove that the methods implementing an object relation are indeed pure (terminating, deterministic, and side-effect free) and satisfy their postconditions; those checks are standard.

The method $\mathsf{check\_C}^T$ encodes the check that C_T is a valid constraining set for T. It can be verified modularly by proving the postcondition for each path through I_T^R that returns true.

The encoding in method $\mathsf{check\_}AP^T$ proves the algebraic property AP. Recall from Def. 2 that we need to prove that there is a type $T' \in C_T$ such that o_2 has type T' and $[\![P]\!]^T_{T'}$ holds, in case $I_T^R(o_1, o_2)$ returns true. We encode this check as follows: We represent the objects o_1 and o_2 from the definition by parameters this and o_2. The additional parameters of $\mathsf{check\_}AP^T$ represent the free variables of P and follow the naming in Table 1. Our encoding assumes that the implementation of R in T for this and o_2 yields true, as the property to check holds trivially otherwise. The encoding then enumerates the types in the constraining set and checks whether the required condition holds for at least one of them. This proof encoding can be verified modularly using the specifications of the methods called in P. If the required condition cannot be proved for any type in the constraining set, the else branch is reached and verification fails.

Note that both the initial invocations of I_T^R and P may contain recursive calls to the same relation (for example, the equality method of some object may call the equality relations of its fields). Since we assume that all pure methods are checked to terminate, one may soundly assume that the desired algebraic property AP holds for these recursive calls. We do not describe how such an

assumption may be expressed here, as this is specific to the used verification logic and tool; for example, such assumptions might come in the form of postconditions of pure methods, axioms describing the pure methods, or relational specifications about the pure methods in relational verification tools.

In situations where some types in the constraining set are in a subtype relation, it is generally easier to prove the disjunct in $\mathsf{check\_}AP^T$ for the subtype because, due to behavioral subtyping, the methods called in P will have stronger postconditions in the subtype. Therefore, a simple performance optimization is to sort the types in the constraining set according to the subtype relation and check the disjunct for a subtype before the disjunct for its supertypes.

For calls of methods on o_2 within P, the encoding discussed so far relies on their specifications to establish the desired property. In the general case that the exact type of o_2 is not statically known, using these specifications is essential to obtain modular proofs via behavioral subtyping. However, the implementations of object relations often branch on the *exact* type of their argument, e.g. by comparing class objects in Java as in Fig. 2. In such cases, we know the exact implementation of the relations and functions called on o_2. Thus, we can use their implementations rather than their specifications to prove that P holds. Implementations generally provide stronger information, for instance, since specifications express only those properties that are also satisfied by all overrides.

Based on this observation, we can refine the encoding in $\mathsf{check\_}AP^T$ by replacing the $\mathsf{assert\ \ false}$ with the following assertion:

$$\begin{aligned}\mathsf{assert}\ \ & \mathit{type}(\mathsf{o}_2) == T_1 \wedge [\![P]\!]^T_{=T_1} \vee \\ & \mathit{type}(\mathsf{o}_2) == T_2 \wedge [\![P]\!]^T_{=T_2} \vee \\ & \ldots \\ & \mathit{type}(\mathsf{o}_2) == T_m \wedge [\![P]\!]^T_{=T_m}\end{aligned}$$

which checks for exact type equality. The transformation $[\![P]\!]^T_{=T_i}$ is a variation of $[\![P]\!]^T_{T_i}$ that uses concrete implementations for calls on o_2, that is, $I^{R'}_{T'}$ and $I^{f}_{T'}$ instead of $I^{R'}_{\leq T'}$ and $I^{f}_{\leq T'}$, resp. For each $[\![P]\!]^T_{=T_i}$, we can inline all calls to relations and functions since our normal form requires them to be invoked on this or o_2, and we know the exact types of both. This process transforms $[\![P]\!]^T_{=T_i}$ into a program that computes a boolean value. We can now use relational reasoning to check whether this program yields true whenever $I^R_T(\mathsf{this}, \mathsf{o}_2)$ does. That is, we get one relational check $\mathit{type}(\mathsf{this}) = T \wedge I^R_T(\mathsf{this}, \mathsf{o}_2) \wedge \mathit{type}(\mathrm{o}_2) = T_i \Rightarrow [\![P]\!]^T_{=T_i}$ per type T_i in the constraining set. These checks are independent of each other, which makes them simple enough to be performed using a sequential composition of the two program parts (similar to self-composition [4]).

3.2 Inferring the Constraining Set

We have assumed so far that constraining sets are declared by the programmer, and their validity is checked by a proof obligation. To reduce the annotation overhead, they can also be inferred automatically. One option is a simple abstract interpretation over the domain of sets of types. The constraining set is the union of all sets of types for o_2 for which the code possibly returns true.

The encoding presented in the previous subsection offers an alternative inference approach, which does not require implementing an abstract interpretation. Instead, we can use a counterexample-guided iterative algorithm. Starting with an (initially empty) candidate constraining set, we attempt to verify *only* $\mathsf{check\_C}^T$ with this candidate set. If verification succeeds, our candidate set is a valid constraining set. Otherwise, we can extract the dynamic type of o_2 from the counterexample, add this type to the candidate set, and re-iterate. Assuming some care is taken to abstract over generic types, this algorithm will always terminate (due to a finite number of struct or class types known modularly) with a valid constraining set. Since we populate the set with the dynamic types from concrete counterexamples, this algorithm yields the trivial constraining set {Object} (which is always valid but imprecise) *only* if necessary, that is, if the object relation may indeed return true for o_2 of exact type Object.

3.3 Push-Button Automation

Our technique relies on method specifications to reason about dynamically-bound calls via behavioral subtyping. However, many practical implementations of object relations satisfy two restrictions that allow us to verify them even without user-provided method specifications: (1) They know the exact type of the receiver of each method call in the implementation of the object relation and in the property P of the intended algebraic property, and (2) they do not invoke recursive methods. The former is typically satisfied when implementations check types via exact comparisons of class objects rather than an instanceof check.

When these restrictions are met, we can check an algebraic property by restricting ourselves to the modified else-branch of method $\mathsf{check\_}AP^T$, as presented in Sec. 3.1. All of the programs resulting from the construction explained there can be verified without method specifications (as calls have been inlined).

Combined with the inference of constraining sets, this approach yields a push-button automation for many practical implementations of object relations.

3.4 Virtual Pure Methods

Our technique relies on virtual pure methods to implement object relations and functions such as hashCode throughout a type hierarchy. As stated earlier, many verification tools support reasoning about pure methods, but most are restricted to *non-virtual* pure methods, that is, their pure methods cannot be overridden. In this subsection, we show how to encode virtual pure methods into non-virtual pure methods, which can be verified using existing tools. This encoding enforces behavioral subtyping to enable modular verification of dynamically-bound calls. The encoding is not novel in itself; variations of it are used in existing tools [19,8] and have been described in the literature [20]. However, our encoding improves on an aspect that is relevant for the verification of object relations.

Fig. 7 shows the encoding of two virtual pure methods (on the left) into non-virtual pure methods (on the right). The difference between the two virtual

```
class T1 {
  ...
  // pure
  virtual TR fn(x̄)
    ensures Q1
  {
    I1
  }
}

class T2 extends T1 {
  ...
  // pure
  override TR fn(x̄)
    ensures Q2
  {
    I2
  }
}
```

```
class T1 {
  ...
  // pure opaque
  TR T1_fn(x̄)
    ensures [Q1]
  {
    [I1]
  }
}

class T2 extends T1 {
  ...
  // pure opaque
  TR T2_fn_verify_body(x̄)
    ensures [Q1] ∧ [Q2]
  {
    [I2]
  }

  // pure abstract
  TR T2_fn(x̄)
    ensures [Q1] ∧ [Q2]
    ensures result == T1_fn(x̄)
}
```

Fig. 7. Arbitrary overridden and non-overridden virtual pure methods (left) are encoded to non-virtual pure methods (right).

methods is that the one in T_1 does not override a supertype method, whereas the one in T_2 does. We discuss both encodings in turn.

For each virtual pure method fn declared in some type T_1 that does *not* override a supertype method, we generate a non-virtual pure method T_1_fn. This method represents fn and all of its overrides: It models the behavior of a dynamically-bound call to method fn on a receiver of static type T_1. Its *conceptual implementation*, which dispatches on the dynamic type of the receiver, is shown in Fig. 8 (left). We need to ensure that clients reason about dynamically-bound calls only in terms of fn's specification, not its implementation in T_1. We achieve that by making the method *opaque* to instruct verification tools to not make the body available to callers. Note that this opaqueness does not affect the verification explained earlier in this section: the encoding discussed there (in particular, inlining of calls) applies to the source program, where pure methods are not opaque; the encoding of pure methods described here is applied afterward.

During the encoding, all dynamically-bound calls to virtual pure methods in implementations and in specifications are replaced by calls to the encoded method (we write this transformation $[e]$ for any expression e). That is, a call $e_r.fn(\overline{e})$, where e_r has static type S, is replaced by a call $[e_r].S\_fn(\overline{[e]})$.

With this encoding, a verification tool will check that fn's body satisfies its specification and use the specification to reason about calls.

The encoding for a pure virtual method fn declared in some type T_2 that *does* override a supertype method is slightly more involved. For overrides, we have to check behavioral subtyping. As we mentioned at the beginning of this section, we assume here that pure methods do not have preconditions. The behavioral

```
class T1 {
  TR T1_fn(x̄) {
    if (type(this) == T1) {
      I1
    } else if (type(this) == T2) {
      I2
    } else if (type(this) == T3) {
      I3
    } else ...
  }
}
```

```
class T2 extends T1 {
  TR T2_fn(x̄) {
    if (type(this) == T2) {
      I2
    } else if (type(this) == T3) {
      I3
    } else ...
  }
}
```

Fig. 8. Conceptual implementation of methods T_1_fn and T_2_fn from Fig. 7, where T_3 is a strict subtype of T_2.

subtyping requirement for postconditions can be checked by proving that the method implementation satisfies both the postcondition of the current type and the postconditions of all methods its overrides (that is, we effectively verify the conjunction of all postconditions, which is always stronger than any of its conjuncts [10]). To achieve that, our encoding introduces a pure, non-virtual opaque method T_2_fn_verify_body that is analogous to T_1_fn for non-overriding methods, but also has the supertype postcondition.

We could now encode calls to fn on receivers of static type T_2 as calls to T_2_fn_verify_body; this approach is sound and used in Dafny, but it is incomplete. Consider the statement y := x; assert x.fn(e) == y.fn(e) for any variables x and y of types T_3 and T_2, resp., where T_3 is a subtype of T_2. Since fn is pure and both calls resolve to the same implementation, the assertion always holds, but its encoding x.T_3_fn_verify_body(e) == y.T_2_fn_verify_body(e) can be verified only if fn's postcondition precisely specifies the result. The encoding uses different methods for the two calls since the receivers have different static types, and these methods are not related.

Proof obligations like this assertion occur in the verification of relations when different overrides of a relation invoke a function on receivers of different static types. To handle them, we introduce a second non-virtual pure method for T_2's fn, which is used to reason about calls. This method, T_2_fn, represents a dynamically-bound call to method fn on a receiver of static type T_2 (analogous to T_1_fn). It has the same postconditions as T_2_fn_verify_body, and an additional postcondition equates the function with each of the functions it overrides. These postconditions are known to hold: the former were already verified in method T_2_fn_verify_body, whereas the latter follows from the fact that for a receiver of type T_2, a call to fn is dispatched to T_2's implementation, even if the receiver's static type is T_1; in other words, the behavior of the conceptual implementations of T_1_fn and T_2_fn, shown in Fig. 8, for a receiver of type T_2 is identical. Hence, we can safely make the method abstract, such that no verification takes place. Calls to fn are encoded as calls to T_2_fn, just like for T_1. Thus, the aforemen-

tioned statement is now encoded as $\mathsf{assert\ x.}T_3\mathsf{\_fn(e) == y.}T_2\mathsf{\_fn(e)}$, which can be proved due to the additional postcondition of $T_3\mathsf{\_fn}$.[1]

4 Soundness

In this section, we justify the soundness of our technique.

Theorem 1. *If every type T is modularly correct w.r.t. some algebraic property AP of a relation R, then R satisfies AP globally.*

Proof. Let AP be an n-relational property of the form $\forall o_1, \ldots, o_n.\, R(o_1, o_2) \Rightarrow P$, where $n \geq 2$. We pick arbitrary objects $o_1, \ldots, o_n$. We assume $R(o_1, o_2)$ and prove P. Let T be the exact type of o_1. Then $R(o_1, o_2)$ is equivalent to $I_T^R(o_1, o_2)$ according to (1), and we therefore know $I_T^R(o_1, o_2)$ yields true. Since T is modularly correct, it has some constraining set C_T. Let T_2 be the exact type of o_2. Since o_2 must have some type $T' \in C_T$, T_2 must be a subtype of T'. By modular correctness, we know that $[\![P]\!]_{T'}^T$ holds. $[\![P]\!]_{T'}^T$ is syntactically identical to P except for invocations of relations and pure methods.

- For each invocation of a relation $R'(o_1, x)$ in P, $[\![P]\!]_{T'}^T$ contains a call to its implementation $I_T^{R'}(o_1, x)$, which is equivalent according to (1).
- For each invocation of a relation $R'(o_2, x)$ in P, $[\![P]\!]_{T'}^T$ contains a call to the virtual method $I_{\leq T'}^{R'}(o_2, x)$. According to (1), $R'(o_2, x)$ is equivalent to $I_{T_2}^{R'}(o_2, x)$. Since T_2 is a subtype of T', $I_{T_2}^{R'}$ must satisfy the specification of $I_{\leq T'}^{R'}$ due to behavioral subtyping. Thus, any statement that is true about $I_{\leq T'}^{R'}$ according to its specification is also true about $I_{T_2}^{R'}$.
- By analogous arguments, any statement involving the encoding of a pure method invocation in $[\![P]\!]_{T'}^T$ is also true about the invocation itself in P.

Therefore, since $[\![P]\!]_{T'}^T$ holds, P must also hold. □

The encoding presented in Sec. 3 checks modular correctness by directly asserting the required properties. The refined encoding shown at the end of Sec. 3.1 encodes a stronger, sufficient condition for modular correctness that is sound because if o_2 has the exact type T', then $[\![P]\!]_{=T'}^T$ is equivalent to P.

5 Implementation and Evaluation

Implementation. We implemented our technique in Nagini [12], an SMT-based verifier for statically-typed Python. Our implementation modularly verifies reflexivity, symmetry, transitivity, and hash consistency of user-defined equality methods, i.e., overridden $\mathsf{\_\_eq\_\_}$ methods. The implementation extends

[1] Note that this encoding is *more* complete than the one used in Dafny, but not absolutely complete in the sense that every property that holds can be proved; as is standard in modular verification, we still sacrifice some completeness for modularity and aim for being "sufficiently complete" to be useful in practice.

Table 2. Summary of case studies verified with Nagini and Dafny. For each benchmark, the table lists the verified relation (R), the total number of lines of code (LOC), the number of proof annotations (Ann.), the encoding mechanism, i.e., automatic or manual (Enc.), the implementation language (Python or Dafny), and the corresponding verification time in seconds (Time).

Benchmark	R	LOC	Ann.	Enc.	Lang.	Time
Core 1	$=$	177	73	Auto	Python	4.42
Core 1F	$=$	24	13	Auto	Python	2.31
Core 2	$=$	233	112	Auto	Python	5.36
Core 2F	$=$	215	105	Auto	Python	5.09
Core 3	$=$	245	123	Auto	Python	5.59
Core 3F	$=$	233	120	Auto	Python	5.24
Birds	$=$	81	36	Auto	Python	5.59
Fractions	$=$	132	68	Auto	Python	3.56
Fractions	$=$	99	40	Manual	Dafny	1.27
Hash Consistency	$=$	109	58	Auto	Python	3.64
LessThan	$<$	148	56	Manual	Python	2.82
LessThan	$<$	42	7	Manual	Dafny	1.18
Map	$=$	51	30	Auto	Python	4.92
Points	$=$	91	45	Auto	Python	3.17
Point Patterns	$=$	178	95	Auto	Python	4.02
Stateless Tuples	$=$	17	10	Auto	Python	3.33
Strings	$=$	81	43	Auto	Python	3.17

Nagini's existing encoding from Python into the intermediate verification language Viper [23] to add the proof obligations for algebraic properties from Fig. 6.

Instead of requiring the programmer to specify constraining sets, we determine them using a simple heuristic: We derive a candidate constraining set C_T for a class T from the definition of `__eq__` by extracting all types explicitly used in checks on the argument `other` (e.g., types T_1 and T_2 for an implementation containing the expressions `isinstance(other,`T_1`)` and `type(other)` $== T_2$) and their known subtypes. This heuristic works well in practice: We have not encountered any realistic cases where the derived candidate set was rejected during its verification. However, user-provided constraining sets or the inference from Sec. 3.2 are still necessary for some implementations, e.g., those using reflection.

Our implementation demonstrates that our technique is fully compatible with heap reasoning in separation logic [27] and the variant implicit dynamic frames [31,32] used in Nagini. In these logics, `__eq__` methods that read the heap require permissions (ownership) in the method's precondition. In order to allow this in a way that is compatible with behavioral subtyping, we use Nagini's existing support for abstract predicate families [25], i.e., abstract separation logic predicates whose definition varies with the type of their receiver argument. User-defined equality methods get the required permissions via an abstract predicate family that is expected to contain permissions to all equality-relevant heap locations for the given type.

Evaluation. We demonstrate the practical applicability of our approach by verifying several case studies with Nagini or Dafny, as summarized in Table 2. Python benchmarks using equality are automatically encoded by our extension of Nagini, while the *LessThan* benchmark involving the operator $<$ was encoded manually as our implementation is currently restricted to equality. To demonstrate that our technique is also applicable to imperative code (as opposed to object-oriented Python code) and program logics other than separation logic, we also encoded two benchmarks manually into Dafny. We measured the verification times on an Apple M4 Pro CPU with 24 GB of RAM and report the average over ten runs. In all cases, verification finishes in five seconds or less.

For all benchmarks, the number of lines of user-provided annotations is approximately equal to the number of lines of actual code, which is well within the standard range for SMT-based verifiers. Our benchmarks use both inexact (isinstance) and exact type checks on the other-object. For the latter, applying the refinement of our technique shown in Secs. 3.1 and 3.3 could further reduce the overhead.

Our benchmarks represent typical coding patterns used in real-world code or recommended in textbooks. Overall, we did not come across any implementation patterns that our technique does not support, except implementations of object relations that are not pure. None of the benchmark failed due to a limitation of our technique. *Points* corresponds to Figs. 1 and 2, translated into Python and extended with specifications. Nagini correctly rejects the code, since ColorPoint and SelectivePoint are not behavioral subtypes of Point. In *Point Patterns*, we show that we can support even complex correct-by-construction patterns from the literature by verifying all three correct implementation patterns for equality recommended by Rupakheti and Hou [28]. Several benchmarks demonstrate our ability to verify common practical use cases of cross-type equality: *Fractions* relates instances of different numeric types (fractions and integer classes), and *Strings* relates a standard string implementation to a representation as a list of characters that is optimized for specific use cases. The *Map* benchmark implements a key-value-store for arbitrary types using two internal lists and implements equality by recursively checking it for these lists, demonstrating that we can verify algebraic properties of complex data types based on the algebraic properties of their components. For the *Core* benchmarks that implement equivalence relations between up to three classes, we created incorrect versions (F1 to F3) that violate individual algebraic properties and are rightfully rejected by our extension of Nagini, showing that verification is fast also for incorrect examples. Finally, our *LessThan* benchmarks, in which we prove irreflexivity, asymmetry, and transitivity of a $<$-operator in an integer wrapper type, show that our encoding works in practice for relations other than equality.

6 Related Work

To the best of our knowledge, no existing verification tools or program logics support modularly verifying algebraic properties of binary object relations. Existing

```
class Integer {
  public uniqueHash = i;
  ...
}
class WrapBool {
  public uniqueHash =
      Integer.MAX_INT  + (b ? 1 : 2);
  ...
}
class WrapByte {
  public uniqueHash =
      Integer.MAX_INT  + by + 1;
  ...
}
```

```
class C extends Object {

  private int i;

  boolean equals(Object other)
    ensures result ==
              other instanceof C &&
              other.i == this.i
  {
    return other instanceof C &&
           other.i == this.i;
  }

}
```

Fig. 9. Two examples showing limitations of existing techniques. Left: An example that shows that the approach used by OpenJML is non-modular: Classes WrapBool and WrapByte can be written independently by different implementers and result in clashes between their unique hashes. Variables i, b, and by refer to the wrapped integer, boolean and byte value, respectively. Right: A class that cannot be verified when extending existing work [29,30] with behavioral subtyping checks.

work observes that equality relations are frequently implemented incorrectly and provides guidelines to avoid such problems [28,21,6]. Alternatively, some authors propose techniques to *generate* correct equals implementations [35,26,13]. Our focus is instead on modularly proving that existing implementations are correct.

Some verifiers (e.g., OpenJML [8] and Nagini) specify (ir)reflexivity, which can be proved independently for every implementation of a relation, as a post-condition on the most general definition of a relation (like Object.equals in Java). However, when checking algebraic properties that require reasoning about different implementations of the same relation, only non-modular solutions exist.

The standard library specifications of OpenJML express symmetry and transitivity for equals methods using a model field uniqueHash in the Object class that contains a single unbounded integer value encompassing all equality-relevant information about an object and can be redefined in subclasses. equals should return true for two objects iff their uniqueHashes are identical, i.e., it should satisfy the postcondition this.equals(obj) $\Leftrightarrow$ this.uniqueHash == obj.uniqueHash[2]. While this property entails symmetry and transitivity of all equals implementations, the approach is inherently non-modular, as demonstrated by the code shown in Fig. 9 (left). Assume that WrapBool and WrapByte are written independently by two developers who are unaware of each other's code and define their hash values outside the integer range to avoid collisions with the existing class Integer. The uniqueHash of a WrapBool object with value true collides with a WrapByte object with value 0. Since neither of the developers is aware of the other class, they could not have prevented this collision. If WrapBool and WrapByte define equals methods such that these two objects are not considered equal, and WrapBool and WrapByte are used in the same program, then the aforementioned

[2] https://github.com/OpenJML/Specs/blob/f34f2329a5f3cbe9953ea9adfaece2c1ae64a1d1/specs/java/lang/Object.jml#L78

postcondition is violated between instances of WrapBool and WrapByte. Once this property is violated, any algebraic properties based on it may not hold either. Thus, composing independently-verified modules in OpenJML's approach is generally unsound whenever a hash collision occurs.

Some static analyses for Java [29,30] encode the behavior of equals methods into Alloy and subsequently check symmetry and transitivity. They compare all known pairs of equals implementations, but are not sound if additional implementations exist at runtime. For example, they accept the combinations of Point and SpecialPoint on the one hand and Point and SelectivePoint from Fig. 2 on the other hand individually, even though symmetry is violated if all three classes exist in the same program. This approach has been extended to (non-modularly) check the consistency between equals and hashCode [24]. All cases for which these analyses are modularly sound (that is, all types that are relevant for the correctness of an implementation of equals are known at verification time) are also handled by the push-button technique described in Sec. 3.3. In addition, our general technique enables sound, modular verification in the common case that this strong assumption is not satisfied. Note that it is not sufficient to extend these analyses by checking algebraic properties between pairs of types based on their *specifications* instead of their implementations and rely on behavioral subtyping to cover unknown subtypes; such an approach would be sound, but extremely weak, as demonstrated by Fig. 9 (right). It would check symmetry between the equals methods C.equals and Object.equals, i.e., prove that they return the same result for an arbitrary pair of argument objects, using some postcondition for both. However, no suitable postcondition for Object.equals exists: The postcondition (this == other) == result prevents all meaningful overrides (due to the behavioral subtyping requirement), including the override in class C itself. In contrast, a weaker postcondition (this == other) ⇒ result allows such overrides, but is now too weak to prove symmetry. Our approach avoids this problem completely by using a constraining set { C } for C.equals, which allows us to prove symmetry for class C by reasoning only about C.equals.

The refinements of our technique discussed in Secs. 3.1 and 3.3 require proving relational properties of potentially different implementations of relations. These relational properties can be proved using relational program logics [5,36], or by reducing them to regular trace properties using self-composition [33] or product programs and related techniques [3,17,15,34,7,11].

7 Conclusion

We have presented the first modular verification technique for algebraic properties of binary object relations. It is applicable to imperative and object-oriented languages, works with different program logics, and supports different relations. For future work, our technique could be extended to verify relations defined by non-boolean binary methods, such as compare methods in Java.

Acknowledgments. We thank the CAV 2026 reviewers for their helpful comments.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Data Availability Statement The artifact accompanying this paper is available online [14]. It contains the source code of our modified version of Nagini as well as the examples verified in our evaluation.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6, https://doi.org/10.1007/978-3-319-49812-6
2. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Commun. ACM **54**(6), 81–91 (2011). https://doi.org/10.1145/1953122.1953145
3. Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: Asymmetric product programs for relational program verification. In: Artëmov, S.N., Nerode, A. (eds.) Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7734, pp. 29–43. Springer (2013). https://doi.org/10.1007/978-3-642-35722-0_3
4. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Math. Struct. Comput. Sci. **21**(6), 1207–1252 (2011). https://doi.org/10.1017/S0960129511000193, https://doi.org/10.1017/S0960129511000193
5. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Jones, N.D., Leroy, X. (eds.) Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004. pp. 14–25. ACM (2004). https://doi.org/10.1145/964001.964003
6. Bloch, J.J.: Effective Java, 2nd Edition. The Java series ... from the source, Addison-Wesley (2008), https://www.worldcat.org/oclc/255160742
7. Churchill, B., Padon, O., Sharma, R., Aiken, A.: Semantic program alignment for equivalence checking. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 1027–1040. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314596
8. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 472–479. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_35
9. Darvas, Á., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: Dwyer, M.B., Lopes, A. (eds.) Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings. pp. 336–351. Lecture Notes in Computer Science, Springer (2007). https://doi.org/10.1007/978-3-540-71289-3_26, https://doi.org/10.1007/978-3-540-71289-3_26

10. Dhara, K.K., Leavens, G.T.: Forcing behavioral subtyping through specification inheritance. In: Rombach, H.D., Maibaum, T.S.E., Zelkowitz, M.V. (eds.) 18th International Conference on Software Engineering, Berlin, Germany, March 25-29, 1996, Proceedings. pp. 258–267. IEEE Computer Society (1996). https://doi.org/10.1109/ICSE.1996.493421, https://doi.org/10.1109/ICSE.1996.493421
11. Dickerson, R., Mukherjee, P., Delaware, B.: Kestrel: Relational verification using e-graphs for program alignment. Proc. ACM Program. Lang. **9**(OOPSLA1) (Apr 2025). https://doi.org/10.1145/3720474
12. Eilers, M., Müller, P.: Nagini: A static verifier for Python. In: Chockler, H., Weissenbacher, G. (eds.) 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. Lecture Notes in Computer Science, vol. 10981, pp. 596–603. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_33
13. Grech, N., Rathke, J., Fischer, B.: JEqualityGen: generating equality and hashing methods. In: Visser, E., Järvi, J. (eds.) Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010. pp. 177–186. ACM (2010). https://doi.org/10.1145/1868294.1868320
14. Greutmann, M., Eilers, M., Müller, P.: Modular reasoning about object relations (artifact) (May 2026). https://doi.org/10.5281/zenodo.19849601, https://doi.org/10.5281/zenodo.19849601
15. Hawblitzel, C., Kawaguchi, M., Lahiri, S.K., Rebêlo, H.: Towards modularly comparing programs using automated theorem provers. In: Bonacina, M.P. (ed.) Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7898, pp. 282–299. Springer (2013). https://doi.org/10.1007/978-3-642-38574-2_20
16. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4085, pp. 268–283. Springer (2006). https://doi.org/10.1007/11813040_19
17. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013. pp. 345–355. ACM (2013). https://doi.org/10.1145/2491411.2491452
18. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems, The Kluwer International Series in Engineering and Computer Science, vol. 523, pp. 175–188. Springer (1999). https://doi.org/10.1007/978-1-4615-5229-1_12
19. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science,

vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
20. Leino, K.R.M., Müller, P.: Verification of equivalent-results methods. In: Drossopoulou, S. (ed.) Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4960, pp. 307–321. Springer (2008). https://doi.org/10.1007/978-3-540-78739-6_24, https://doi.org/10.1007/978-3-540-78739-6_24
21. Liskov, B., Guttag, J.V.: Program Development in Java - Abstraction, Specification, and Object-Oriented Design. Addison-Wesley (2001)
22. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994). https://doi.org/10.1145/197320.197383
23. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings. Lecture Notes in Computer Science, vol. 9583, pp. 41–62. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_2
24. Okano, K., Harauchi, S., Sekizawa, T., Ogata, S., Nakajima, S.: Consistency checking between Java equals and hashcode methods using software analysis workbench. IEICE Trans. Inf. Syst. **102-D**(8), 1498–1505 (2019). https://doi.org/10.1587/TRANSINF.2018EDP7254
25. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005. pp. 247–258. ACM (2005). https://doi.org/10.1145/1040305.1040326
26. Rayside, D., Benjamin, Z., Singh, R., Near, J.P., Milicevic, A., Jackson, D.: Equality and hashing for (almost) free: Generating implementations from abstraction functions. In: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings. pp. 342–352. IEEE (2009). https://doi.org/10.1109/ICSE.2009.5070534
27. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings. pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
28. Rupakheti, C.R., Hou, D.: An empirical study of the design and implementation of object equality in Java. In: Chechik, M., Vigder, M.R., Stewart, D.A. (eds.) Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada. p. 9. IBM (2008). https://doi.org/10.1145/1463788.1463800
29. Rupakheti, C.R., Hou, D.: An abstraction-oriented, path-based approach for analyzing object equality in Java. In: Antoniol, G., Pinzger, M., Chikofsky, E.J. (eds.) 17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA. pp. 205–214. IEEE Computer Society (2010). https://doi.org/10.1109/WCRE.2010.30
30. Rupakheti, C.R., Hou, D.: EQ: checking the implementation of equality in Java. In: IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011. pp. 590–593. IEEE Computer Society (2011). https://doi.org/10.1109/ICSM.2011.6080837

31. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. ACM Trans. Program. Lang. Syst. **34**(1), 2:1–2:58 (2012). https://doi.org/10.1145/2160910.2160911
32. Spies, S., Mück, N., Zeng, H., Sammler, M., Lattuada, A., Müller, P., Dreyer, D.: Destabilizing Iris. Proc. ACM Program. Lang. **9**(PLDI), 848–873 (2025). https://doi.org/10.1145/3729284, https://doi.org/10.1145/3729284
33. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Hankin, C., Siveroni, I. (eds.) Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3672, pp. 352–367. Springer (2005). https://doi.org/10.1007/11547662_24
34. Unno, H., Terauchi, T., Koskinen, E.: Constraint-based relational verification. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I. pp. 742–766. Lecture Notes in Computer Science, Springer (2021). https://doi.org/10.1007/978-3-030-81685-8_35, https://doi.org/10.1007/978-3-030-81685-8_35
35. Vaziri, M., Tip, F., Fink, S., Dolby, J.: Declarative object identity using relation types. In: Ernst, E. (ed.) ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4609, pp. 54–78. Springer (2007). https://doi.org/10.1007/978-3-540-73589-2_4
36. Yang, H.: Relational separation logic. Theor. Comput. Sci. **375**(1-3), 308–334 (2007). https://doi.org/10.1016/J.TCS.2006.12.036