Towards a Declarative Approach to Object Comparison

Bruno Sofiato*[1], Fábio Levy Siqueira[1], and Ricardo Luis de Azevedo Rocha[1]

Escola Politécnica, Universidade de São Paulo (USP), São Paulo, Brazil

Abstract. Equality and ordering are relatable concepts we use daily. As they depend on the domain at hand, the programmer must eventually code them. Empirical evidence points towards defects on code that models comparison on open source software written in object-oriented (OO) languages. The main culprits are: comparison logic that spans multiple methods; and, shortcomings of the message dispatch scheme found on most OO languages. There are several proposals to mitigate these issues. None, however, deals with the ordering aspect and some forbid objects to be mutable. In this paper, we define a declarative, language-agnostic way of comparing objects that handles equality, ordering, and hashing aspects of comparison. We formalize it using Plotkin's style operation semantics and provide detailed proof of its soundness.

Keywords: Object-oriented programming · Computer languages · Comparison semantics

1 Introduction

Equality is an abstract concept that establishes "when" two entities are similar to each other. The natural ordering is a complementary concept that determines "how" they differ. They depend on the domain at hand and constitute the comparison semantics of an entity. However, writing code that compares objects is complex [7, 22, 23, 26]. Baker [1] argues that such a task should be simpler (recently, C++20 plans to add the spaceship operator with this intent [14]).

One of the culprits is that comparison usually spans several methods that must be consistent with each other [3]. The developer must enforce such consistency by hand. Automatic checking is not feasible, since it is reducible to the program-equivalence problem (and thus undecidable [25]). Static code analysis is also not an option, since it is notoriously prone to false positives [11]. Inheritance further escalates the issue. Bruce et al. [4] noted that it precludes the maintenance of symmetry in languages that employ simple message dispatch (which is the case of most mainstream OO languages). A valid equivalence relation must be symmetric.

The goal of this paper is to devise, in the theoretical level, a way of comparing objects where the developers use metadata to encode semantics. Our proposal

* Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

handles both aspects of comparison – equality and order. We specify how to use this metadata to compare objects and which information it contains. Both languages and libraries may use our proposal as a foundation on how to compare objects.

The benefits of our approach are two-fold: it shields the developer from the complexity of writing code that correctly compares objects in single-dispatch, OO languages; and, it restores orthogonality, as a single element - the metadata - is used instead of several methods.

There are proposals [10, 21, 26] in the literature that employ different degrees of declarative programming. None deals with the ordering aspect of equality. Also, some of them [1, 10, 26] push that objects must be immutable, since some collections fail when its items change after inclusion. We feel, however, that such constraints forbid common programming idioms and hinder adoption. For instance, ORMs[1] frameworks usually rely on mutability to track changes to entities. We think that it would be better if the collections themselves enforce such constraints. Discussing the intricacies of such an approach is out of the scope of this paper.

We employ Plotkin‘s [20] Structural operational semantics (SOS) to formalize our proposal. We also use both Featherweight Java (FJ) [12] and Cardelli and Mitchell's record calculus [5] in the formalization process (Section 7). This approach allows us to define our proposal without tying it to any language and also provides us a framework to prove its soundness.

The remainder of this paper is structured as follows. Section 2 lays down the mathematical aspects of comparison. Next, Section 3 outlines the impacts of inheritance on comparison. Section 4 describes how the comparison is coded in two distinct OO languages, while Section 5 introduces the hash concept. Section 6 describes our proposal informally and lays the foundations for the formal description on Section 7. Section 8 proves the the soundness of our proposal. Finally, in Sections 9 and 10, we review related work and provide some final insights.

2 Equality and Ordering Semantics

From a set-theoretic point of view, equality and order can both be expressed as binary relations between elements of a set. Two elements are said to be equal whenever an equivalence relation exists between them. Order is modeled by another kind of binary relation known as strict-total order (Zermelo [27] demonstrates that at least one ordering scheme can be found on every set, albeit sometimes without any meaningful semantics).

The OO paradigm is built upon the notion of modeling real-world elements onto constructs called objects. We access them via their references[2]. Multiple references can point to the same object. In this case, we say they have the same

[1] An object-relation mapping (ORM) framework maps objects into tables in a relational database. An example of ORM is the Hibernate framework.

[2] Through this paper, we use the terms object and reference interchangeably.

identity. The identity of an object can determine if two objects are similar[3]. In such a case, only references to the same physical object are equal. Often, only a subset of properties determines if two objects are equal (e.g., in a particular context, persons are equal when their Social security number match, regardless of any other properties). We call this minimum subset of properties the object's **equality state**.

Regardless of its semantics, equality must observe a well-known set of rules to be valid. Bloch [3] stated that failure to observe them leads to subtle bugs that are hard to pin-point. Definition 1 describes those rules.

Definition 1. *A well-formed equivalence relation must be:* ***reflexive*** – $\forall x[(x = x)]$ –, ***symmetric*** – $\forall x \forall y[(x = y) \iff (y = x)]$ – *and* ***transitive*** – $\forall x \forall y \forall z[(x = y) \wedge (y = z) \implies (x = z)]$.

Ordering relations also have well-formedness rules that must hold, regardless of the underlying ordering semantics. Definition 2 describes them.

Definition 2. *A well-formed order relation must be:* ***asymmetric*** – $\forall x \forall y[(x < y) \iff (y > x)]$ –, ***transitive*** – $\forall x \forall y \forall z[(x > y) \wedge (y > z) \implies (x > z)]$ – *and* ***trichotomous*** – $\forall x \forall y[(x \neq y) \iff (x > y) \vee (x < y)]$.

3 Symmetry with Single Dispatch and Inheritance

Coding comparison is complex. There is, however, a particular scenario presented by Bruce et al. [4] that challenges developers the most. Let us assume two classes: *Point* and *ColorPoint* (Figure 1). *Point* has two attributes – x and y – that holds its coordinates within a cartesian plane. It also defines the == method, which checks if both operands' coordinates match. *ColorPoint* includes a new attribute – *color* – that stores the color of a point. It also extends == to check colors (as such, two instances of *ColorPoint* are only equal if their coordinates and colors match).

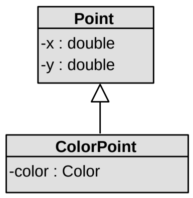

Fig. 1. Point/ColorPoint hierarchy (Bruce et al. [4])

Given two objects – $a : Point$ and $b : ColorPoint$ –, that shares the same coordinates. Invoking $a == b$ triggers the $Point's$ implementation of == (yielding *true*, since it only checks the coordinates of each operand). Invoking $b == a$,

[3] Matter of fact, this models the Leibniz's Identity of Indiscernibles Principle.

however, triggers the $ColorPoint's$ implementation, which requires the coordinates and color of both operands to match. Since a does not have a color, $b == a$ yields $false$, which is a violation of symmetry.

Therein lies the issue: languages based on single-dispatch are unable to maintain symmetry when inheritance is present [4]. There are techniques to simulate multiple dispatch on simple-dispatch languages [8, 13]. They, however, have negative impacts[4] on both encapsulation and modularity.

At first glance, this problem seems trivial: we could assume that equal objects must be instances of the same concrete class: Rupakheti and Hou [22] calls this **type-incompatible** equality. However, this is not the only kind of equality. Rupakheti and Hou describes another strategy – **type-compatible** – that does not require the objects to have the same concrete class.

To better explain the results of supporting only type-incompatible equality, let us imagine a drawing software that allows the user to draw, among other shapes, square and rectangles. Each of those shapes has its own class. A viable criterion to decide if two shapes are equal is the congruency: equal shapes must have the same form and size. By that rule, a square and a rectangle whose sides are the same are equal. Such modeling is only possible if we use type-comparison equality.

4 Implementing comparison in OO Languages

Eventually, the programmer must implement comparison semantics, since it varies according to the domain at hand. How to do it, however, depends on the language we are using, with each one having its quirks. Thus, to build an agnostic solution, we must first find common ground between them. To do that, we analyzed two languages: Java [9] and Ruby [6].

Many popular languages (e.g., Python and Smalltalk) follow the same approach of Java and Ruby. Two notable exceptions are JavaScript and C++. JavaScript does not have a standard way to compare objects. C++ does, but it misses a general-purpose hash function.

Java is a statically-typed OO language whose execution model allows it to run on a variety of devices without modifications. On Java, comparison spans three distinct methods: *equals*, *hashCode*, and *compareTo*. The equals method checks if two operands are equal (by default, that is only the case if they share the same identity). The *hashCode* method is a general-propose hash function that some kinds of collection use internally. According to Bloch [3], an informal contract requires both methods to be consistent with each other (Section 5).

In Java, objects are not comparable by default. They must realize the Comparable interface to be comparable. *Comparable* declares a single method – *compareTo* – that models the Trichotomy Law. The results of *compareTo* must be consistent with the results of *equals* (i.e., if *x.equals(y)* returns *true*, then

[4] According to Muschevici et al. [17], those impacts occur even in languages that support multiple-dispatch natively.

x.compareTo(y) yields zero). As with *hashCode*, the programmer must enforce this consistency themselves.

Ruby is an OO language that is heavily influenced by Smalltalk. Ruby is used as both a scripting language and in web development (alongside with the Rails framework). Like Java, Ruby defines[5] both an equality and a hash operation (== and *hash*). Their semantics are also identity-based by default. Ordering is also optional on Ruby. In such a case, it is required that the object includes the *Comparable* mixin, which requires the developer to implement the spaceship operator ($\Leftrightarrow$). The $\Leftrightarrow$ operator carries the same semantics as *compareTo* in Java.

Ruby's approach differs from Java's in the sense that the *Comparable* mixin incorporates implementations for every other comparison operation, minus the *hash* operation. They rely on the $\Leftrightarrow$ operation and are guaranteed to be consistent with each other. That frees the developer from the burden of ensuring their consistency (sadly, they still have to ensure that *hash* is consistent).

5 Hashing

Despite not being strictly necessary to model comparison, Java defines a general-purpose hash function. An important type of collection – the equality collections [18] – relies heavily on it. These collections usually provide better performance in selected operations, such as element lookup. Hash tables and sets are examples of them.

Hash functions are directly linked to equality. According to Bloch [3], a hash function h is correct if $\forall x \forall y[(x = y) \implies (h(x) = h(y))]$[6] (as a corollary, two different objects may have (or may have not) the same hash). An erroneous hash function may lead to subtle bugs that are hard to pin-point. Tantamount to their correctness is their efficiency. A badly written function severely hinders performance while still correct[7]. The efficiency of a hash function is based on how it yields distinct hashes to distinct objects. Achieving an optimal hash function can be a daunting task (and is often not needed).

6 Evaluating comparison-related operations with comparison records

We informally describe our solution before dwelling into its formal system. This informal description is useful to understand the formalization's intricacies.

[5] Ruby defines two more operations – === and *eql?*. The former is used within switch/case statements, whereas the latter is used inside hash tables to compare keys. For the sake of argument, let us consider their semantics the same as ==.

[6] At first, it looks like a tautology. Yet, this is true only when following a stricter sense of equality (i.e., identity-based equality), which is not the case. It is trivial to implement *Point* in such a way that only x is checked by *equals* while every attribute composes its hash. In that case, $a : Point$ and $b : Point$ having the same value of x and different values of y will be considered equal despite having distinct hashes.

[7] A constant hash function (e.g., $h(x) = 42$) is especially bad. It degrades the lookup of elements in a hash table from $\mathcal{O}(1)$ to $\mathcal{O}(n)$ (akin to a regular, linked list).

The process of evaluating comparison operation is divided into two phases. In the first phase, a comparison record is built. The comparison record encapsulates the context on which a comparison operation is evaluated. In the second phase, the newly-created comparison record is evaluated.

6.1 Comparison records

Comparison records are the backbone of our proposal. They are built upon the Cardelli and Mitchell [5] record calculus and provide context to compute comparison operations. There is a key difference between them and regular records. On regular records, fields are not traversed in any particular order. Fields on comparison records are ordered based on their order of evaluation (which is domain-specific).

Informally, a comparison record is a collection of fields. Each field has two components – x and y – that store the value associated with the first and second operand. Values are extracted via the extraction operation (e.g. $r.x$ extracts the value x from record r). Extraction operations can be chained.

6.2 Denoting comparison semantics

We made a conscientious choice of not committing to any particular form of denoting comparison semantics. Doing so would bias our proposal towards a language and hurt its agnosticism. Different languages have different ways of embedding metadata. Java, for instance, uses annotations (both Rayside et al. [21] and Grech et al. [10]. use them). A Ruby programmer, on the other hand, may employ metaprogramming techniques instead. A new language (such as the Java extension proposed by Vaziri et al. [26]) could use keywords to convey the same information.

Regardless of ”how” this information is embedded, we can define ”what” we must include. To be able to denote semantics, we must be able to specify: a) if a field belongs to the equality state of its underlying object; b) what is its evaluation order.

6.3 Record building phase

At first glance, it might appear that comparison records do not bring further development to the equality state concept. Their differences, however, become more clear as we describe how they are built. Firstly, we collect the fields that compose the equality state of *each* operand. Then, we fetch the values of each one of those fields within each operand, and include them in the newly-built comparison record.

Some scenarios pose an extra challenge. When there is a sub-type relation between the operands, a particular field may appear in just one of them. For instance, if we compare $a : Point$ and $b : ColorPoint$, looking up for field *color* on a would fail, as *Point* does not declare the *color* field. In that case, we expect the field lookup operation to yield a special element – the **unknown** element (henceforth denoted by $\sqcup$) – instead.

For example, a comparison record built to compare $a : Point$ and $b : Point$ has the fields x and y. Yet, a record built to compare $b : Point$ and $c : ColorPoint$ has three fields – x, y, and *color* (in that case, the component bound to b within *color* stores $\sqcup$). Notably, b yields different values, depending on the object against it is compared: a striking difference from the equality state concept.

At first glance, $\sqcup$ and *null* seems to share the same semantics. Yet, there is a fundamental difference: *null* usually represents the absence of value, whereas $\sqcup$ denotes that even such an assertion is not feasible at the time (if the field does not exist, we cannot determine whether it holds a value or not). In a sense, $\sqcup$ shares the same semantics as *unknown* on Kleene's K_3^S Logic [15] and is different to any other value (including itself). Concerning its order, $\sqcup$ should always be considered the lesser of two values.

A corner-case worth mentioning deals with how to use comparison records to compute hashes. Hashing is a unary operation, yet, two operands are required to create a comparison record. In that case, we use the hash operation's single operand as both the first and second operand of the record building procedure.

6.4 Record evaluation phase

After we build the comparison record, we can proceed to evaluate it. The evaluation is carried out by one of three specialized functions. These functions map directly to the three aspects of equality outlined in Section 4 – equality, order, and hashing.

Checking equality is straightforward. We iterate through every field within a comparison record, checking if their values are different. In that case, we deem the operands as different and stop the process. If no discrepancies were found after all fields have been checked, then we consider the operands equal.

Determining the order between two operands is similar. We iterate through every field, checking if one of the values within the current field is greater than the other. In that case, the operand bound to the greater value is deemed greater and the process stops. If the iteration ends without finding any discrepancies, the operands are equal.

The algorithm to compute hashes also iterates over every field of a comparison record. For each field, it calculates the field's hash and aggregates it on a single value, which will be the hash of the whole comparison record by the end of the process. Unlike the other operations, hashing does not allow short-circuiting.

Dwelling into the optimal form to aggregate hashes is out of the scope of this paper. Still, the rewriting rules described in Section 7 (and their proofs in Section 8) depend on the aggregation function $\oplus$ to follow a single well-formedness rule: $(\mathbb{Z}, \oplus)$ must be a semi-group[8].

[8] A semi-group is an algebraic structure composed of a set $\mathcal{S}$ and an associative binary operation $\oplus$.

Table 1. Notation summary

Construct	Description
$\langle\rangle$	Denotes an empty record
$r = \langle x = 0\rangle$	Creates a record r with field x, whose value is 0
$\langle r \| x = 0\rangle$	Adds field $x = 0$ to the record r, returning a new record
$r.x$	Fetches the value associated to field x on r
$r\backslash x$	Removes field x from record r, returning a new record
$A\backslash B$	Removes every element within set B from set A (e.g., $\{1,2,3\}\backslash\{1,4\} = \{2,3\}$)
$\overline{x}$	Shorthand to sequences such as $x_0 \dots x_n$ (x may be replaced by any other letter or symbol)
class $C \lhd D$ { $\overline{C}\ \overline{f}$; $K\ \overline{M}$}	Defines class C extending from class D. C is composed of fields $\overline{f}$, the constructor K and methods $\overline{M}$.
$o : C$	Denotes an object o which is an instance of class C
$fields(C)$	Obtains all fields defined on C and its super-classes
$es\_fields(C)$	Obtains all fields on C's (and its super-classes) equality state
$\top$	Represents **true**
$\bot$	Represents **false**
$\sqcup$	Represents **unknown** value (as described in Section 6.3)

7 Formal Semantics

The given formalization uses both Cardelli and Mitchell [5] record calculus (Table 1) and Featherweight Java [12] (FJ). One may argue why we used FJ, as it focuses on type safety (which is not a concern of our proposal). Shortly, we use FJ's definition of classes and its field lookup semantics. These constructs are similar to those found on the majority of OO languages. As such, we feel that they do not make our proposal less agnostic.

7.1 metadata access

Neither Cardelli‘s record calculus nor FJ defines the *es_fields* function. Its role is to obtain which fields compose the equality state of an object based on supplied metadata (Section 6.2). To prevent bias, we choose not to commit to any form of metadata. Instead, we treat *es_fields* as a black box. Given a class C, it returns the fields $\overline{f}$ that compose the equality state of C, ordered by their evaluation order (Definition 3).

7.2 Comparison record

Definition 3 formalizes comparison records, as described in Section 6.1.

Definition 3 (Comparison record). *A comparison record r is a record $\langle f_0 = \langle x = x_0, y = y_0\rangle, f_1 = \langle x = x_1, y = y_1\rangle \dots f_n = \langle x = x_n, y = y_n\rangle\rangle$. For each field $f_i = \langle x = x_i, y = y_i\rangle$, x stores the value bound to the first operand while y*

$$\text{CR-BUILD}\ \frac{o_1 : C \quad class\ C \lhd D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \quad o_2 : C' \quad class\ C' \lhd D'\ \{\overline{C}'\ \overline{f}';\ K'\ \overline{M}'\}}{build(o_1, o_2) \rightarrow cpBuild(o_1, o_2, es\_fields(C) \cup es\_fields(C'))}$$

$$\text{CR-EMPTY}\ \frac{f \rightarrow \emptyset}{cpBuild(o_1, o_2, f) \rightarrow \langle\rangle}$$

$$\text{CR-SINGLE}\ \frac{f \rightarrow \{f_0\}}{cpBuild(o_1, o_2, f) \rightarrow \langle x = fetch(f_0, o_1), y = fetch(f_0, o_2)\rangle}$$

$$\text{CR-MUTIPLE}\ \frac{f \rightarrow \{f_o \dots f_n\}}{cpBuild(o_1, o_2, f) \rightarrow \langle cpBuild(o_1, o_2, f \backslash \{f_n\}) \mid cpBuild(o_1, o_2, \{f_n\})\rangle}$$

$$\text{FETCH-EXISTS}\ \frac{o : C \quad class\ C \lhd D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \quad f \in \overline{f}}{fetch(f, o) \rightarrow o.f}$$

$$\text{FETCH-DOES-NOT-EXIST}\ \frac{o : C \quad class\ C \lhd D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \quad f \notin \overline{f}}{fetch(f, o) \rightarrow \sqcup}$$

Fig. 2. Record building semantics.

stores the value bound to the second operand. The iteration of the fields $\overline{f}$ within r follows the LIFO[9] *style (e.g. $f_n, f_{n-1} \dots f_0$).*

Fields of comparison records may store the $\sqcup$ value. As stated in Section 6.1, any comparison having $\sqcup$ as one of its operands states that the operands are different ($\sqcup$ being the lesser one). Definition 4 formalizes its semantics.

Definition 4 ($\sqcup$ comparison semantics). *The value $\sqcup$ denotes an unknown value and is considered to be different from any other value (including itself). As such, $(\forall x)(x \neq \sqcup)$ holds. When deciding the order of two values, $\sqcup$ will always be the lesser*[10] *one. Hence, $(\forall x)(x > \sqcup)$ and $(\forall x)(\sqcup < x)$ hold.*

7.3 The Record Building Semantics

The first phase of computing comparison operations is the record building stage. The *build* function is the entry point to the record build process. It receives two objects as operands and returns a corresponding comparison record. Rule `CR-BUILD` formalizes the *build* function. As shown in Figure 2, `CR-BUILD` uses *es_fields* to fetch the fields that are part of the equality state of each operand. It calls then *cpBuild*, which recursively builds comparison records.

Rules `CR-EMPTY`, `CR-SINGLE`, and `CR-MULTIPLE` formalize *cpBuild*. `CR-SINGLE` calls $fetch$ to obtain the value associated with a given field f in the context of an object o. `FETCH-EXISTS` and `FETCH-DOES-NOT-EXIST` formalize $fetch$. The former fires when the class hierarchy of o defines the field f. In that case, it returns $o.f$. Otherwise, the latter fires, and it returns $\sqcup$ instead.

[9] Acronym for last in, first out. LIFO is the form of iteration of stacks.

[10] At first, this may seem like an oversight, as it blatantly violates asymmetry. However, having a field f storing $\sqcup$ on each component is not possible. It implies that f does not exists on both operands. In that case, f would not be part of the comparison record in the first place.

$$\text{EQ-EMPTY}\ \frac{r \to \langle\rangle}{cr\_eq(r) \to \top} \qquad \text{EQ-SINGLE}\ \frac{r \to \langle f_0 = \langle x = x_0, y = y_o\rangle\rangle}{cr\_eq(r) \to eq(x_0, y_0)}$$

$$\text{EQ-MULTIPLE-EQ}\ \frac{r \to \langle f_0 = \langle x = x_0, y = y_0\rangle \ldots f_n = \langle x = x_n, y = y_n\rangle\rangle \qquad cr\_eq(r.f_n) \to \top}{cr\_eq(r) \to cr\_eq(r \backslash f_n)}$$

$$\text{EQ-MULTIPLE-NEQ}\ \frac{r \to \langle f_0 = \langle x = x_0, y = y_0\rangle \ldots f_n = \langle x = x_n, y = y_n\rangle\rangle \qquad cr\_eq(r.f_n) \to \bot}{cr\_eq(r) \to \bot}$$

$$\text{EQ-UNKNOWN}\ \frac{x = \sqcup \vee y = \sqcup}{eq(x, y) \to \bot} \qquad \text{EQ-KNOWN}\ \frac{x \neq \sqcup \wedge y \neq \sqcup}{eq(x, y) \to x = y}$$

$$\text{CP-MULTIPLE-EQ}\ \frac{r \to \langle f_0 = \langle x = x_0, y = y_0\rangle \ldots f_n = \langle x = x_n, y = y_n\rangle\rangle \qquad cr\_cp(r.f_n) \to 0}{cr\_cp(r) \to cr\_cp(r \backslash f_n)}$$

$$\text{CP-MULTIPLE-NEQ}\ \frac{r \to \langle f_0 = \langle x = x_0, y = y_0\rangle \ldots f_n = \langle x = x_n, y = y_n\rangle\rangle \qquad cr\_cp(r.f_n) \neq 0}{cr\_cp(r) \to cp(r.f_n)}$$

$$\text{CP-SINGLE}\ \frac{r \to \langle f_0 = \langle x = x, y = y\rangle\rangle}{cr\_cp(r) \to cp(x, y)} \qquad \text{HC-SINGLE}\ \frac{r \to \langle f_0 = \langle x = x_0, y = y_0\rangle\rangle}{cr\_hc(r) \to hc(x_0)}$$

$$\text{CP-UNKNOWN-LT}\ \frac{x = \sqcup}{cp(x, y) \to -1} \qquad \text{CP-UNKNOWN-GT}\ \frac{y = \sqcup}{cp(x, y) \to 1}$$

$$\text{CP-EMPTY}\ \frac{r \to \langle\rangle}{cp(r) \to 0} \qquad \text{CP-KNOWN}\ \frac{x \neq \sqcup \qquad y \neq \sqcup}{cp(x, y) \to x \Leftrightarrow y} \qquad \text{HC-EMPTY}\ \frac{r \to \langle\rangle}{cr\_hc(r) \to 0}$$

$$\text{HC-MULTIPLE}\ \frac{r \to \langle f_0 = \langle x = x_0, y = y_0\rangle \ldots f_n = \langle x = x_n, y = y_n\rangle\rangle}{cr\_hc(r) \to cr\_hc(r \backslash f_n) \oplus hc(r.f_n)}$$

Fig. 3. Record evaluation semantics.

7.4 The Record Evaluation Semantics

The evaluation stage is the second phase of computing comparison operations. A distinct set of rules evaluates each comparison operation. Figure 3 describes them.

The function *cr_eq* evaluates the equality of the comparison record r, using `EQ-MULTIPLE-EQ` and `EQ-MULTIPLE-NEQ` to iterate recursively through the fields of r. Having different values $f_n.x$ and $f_n.y$ on the last field f_n triggers `EQ-MULTIPLE-NEQ`, ending the process and returning $\bot$. Otherwise, rule `EQ-MULTIPLE-EQ` is triggered. It acts as the recursive step of the equality checking procedure. Eventually, `EQ-SINGLE` is triggered when only a single field f_0 remains on r. Its result depends on whether $f_0.x$ and $f_0.y$ are equal. The function *eq* determines if $f_0.x$ and $f_0.y$ are equivalent. `EQ-UNKNOWN` and `EQ-KNOWN` model the semantics of *eq*. If one of the given operands is $\sqcup$, `EQ-UNKNOWN` is triggered and $\bot$ is obtained (Definition 4). Conversely, `EQ-KNOWN` is triggered if neither is $\sqcup$. If the given operands are primitives[11] , their equivalence is checked by their native implementation (otherwise, a comparison record is built from $f_0.x$ and $f_0.y$ and is subsequently evaluated). Regardless of the approach, either $\top$ or $\bot$ is returned, depending on whether $f_0.x$ and $f_0.y$ are equal.

The function *cr_cp* evaluates the order of a comparison record r. Its main rules are `CP-MULTIPLE-EQ` and `CP-MULTIPLE-NEQ`, which operate under the same

[11] We use the term *primitive* in a broader sense. Here, any type that provides comparison operations is primitive. According to that, Java's strings are primitive.

rationale as `EQ-MULTIPLE-EQ` and `EQ-MULTIPLE-NEQ`. `CP-EMPTY` handle empty comparison records while `EQ-SINGLE` uses the *cp* function to determine the order of two operands (in this case, $f_0.x$ and $f_0.y$). Rules `CP-UNKNOWN-LT`, `CP-UNKNOWN-GT`, and `CP-KNOWN model` the semantics of *cp*. `CP-UNKNOWN-LT` and `CP-UNKNOWN-GT` deal with the scenarios described by Definition 4 and fire, respectively, when $\sqcup$ is the first or the second operand. In the former case, the first operand is the lesser one whereas, in the latter, it is the greater one. `CP-KNOWN` fires when neither operand is $\sqcup$. To obtain the order of the given operands, it uses the $\Leftrightarrow$ operator. Its implementation can be either native (if the operands are primitive) or based on comparison records.

The function *cr_hc* evaluates the hash of a comparison record r. If r has more than one field, `HC-MULTIPLE` triggers. `HC-MULTIPLE` divides r into two sub-records – r_1 and r_2 – containing the last field and the remainder fields. The hash of both sub-records are obtained and then combined by the $\oplus$ operator (as stated in Section 6.4, $(\mathbb{Z}, \oplus)$ must be a semi-group). Eventually, only a single field f_0 remains on r. That triggers `HC-SINGLE`, which entails that r's hash is the same as of $f_0.x$. Function *hc* computes the hash of $f_0.x$.

8 Soundness of comparison-related properties

After we formalize our proposal, we can proceed to prove its correctness based on the rules outlined by Definitions 1 and 2 and the rules defined in Section 5, regarding hash functions. The proofs have two premises: the operands do not change while the operation executes; and, the values on their fields have themselves correct comparison procedures.

Lemma 1 (Equality symmetry). *Given two objects – $x : C$ and $y : C'$ –, then $x = y \Leftrightarrow y = x$.*

Proof. By applying `CR-BUILD`*,* `EQ-MULTIPLE-EQ`*,* `EQ-SINGLE`*, and* `EQ-EMPTY`*, we infer that $(x = y) \implies [(\forall f \in \overline{f})(x.f = y.f)]$, where $\overline{f} = (es\_fields(C) \cup es\_fields(C'))$. Assuming $y \neq x$ implies that either the set union is not commutative or $[(\exists f \in \overline{f})((x.f = y.f) \wedge (y.f \neq x.f))$. The latter case implies that the values on each field (primitives or objects) are not symmetric themselves.*

Corollary 1 (Equality reflexivity). *Given one object $o : C$, then $o = o$.*

Proof. `CR-BUILD` *yields $r = \langle f_0 = \langle x = fetch(f_0, o), y = fetch(f_0, 0)\rangle \ldots f_n = \langle x = fetch(f_n, o), y = fetch(f_n, o)\rangle\rangle$. Since eq is reflexive and assuming that $fetch$ returns the same result when given the same parameters. We conclude that $eq(fetch(f, o), fetch(f, o))$ will always return $\top$ for every field f. By applying* `EQ-MULTIPLE-EQ` *repeatedly, we eventually trigger* `EQ-SINGLE`*, returning $\top$.*

Corollary 2 (Equality transitiveness). *Given three objects – $x : C$, $y : C'$ and $z : C''$ –, then $(x = y \wedge y = z) \Leftrightarrow x = z$.*

Proof. Similar to the proof of Lemma 1.

Lemma 2 (Order transitiveness). *Given three objects – $x : C$, $y : C^{'}$ and $z : C^{''}$ –, then $(x > y \wedge y > z) \Leftrightarrow x > z$.*

Proof. *Assuming that $x > y$ and $y > z$ and analyzing* ***`CP-MULTIPLE-NEQ`***, *we infer that two fields – $f^{'}$ and $f^{''}$ – are determinant to conclude that indeed $x > y$ and $y > z$. Based on that, we divide this proof into three scenarios.*

When $f^{'} = f^{''}$, transitiveness is trivially proven as $\Leftrightarrow$ is transitive itself.

When $f^{'}$ is evaluated before $f^{''}$, we conclude that $y.f^{'} = z.f^{'}$ (since otherwise $f^{'}$ would have greater evaluation order). By replacing $y.f^{'}$ on $x.f^{'} > y.f^{'}$ we have $x.f^{'} > z.f^{'}$, thus, $x > z$.

Finally, when $f^{'}$ is evaluated after $f^{''}$, we conclude that $x.f^{''} = y.f^{''}$ (otherwise, $f^{'}$ would have a lesser evaluation other). Replacing $y.f^{''}$ on $y.f^{''} > z.f^{''}$ yields $x.f^{''} > z.f^{''}$, thus, $x > z$.

Corollary 3 (Order asymmetry). *Given two objects – $x : C$ and $y : C^{'}$ –, then $(x < y) \Leftrightarrow (y > x)$.*

Proof. *Having $x < y$ implies that a field f where $x.f < y.f$ exists. Having $y > x$ implies either: cp is not asymmetric; the traversal order of $f^{'} = (es\_fields(C) \cup es\_fields(C^{'}))$ or $f^{''} = (es\_fields(C^{'}) \cup es\_fields(C))$ is not based on the evaluation order of their fields; or, $f^{'}$ and $f^{''}$ do not have the same elements. Any of which is contradictory.*

Corollary 4 (Order trichotomy). *Given two objects – $x : C$ and $y : C^{'}$ –, they must be either $x = y$, $x < y$ or $x > y$.*

Proof. *By analyzing* ***`CP-EMPTY`***, ***`CP-UNKNOWN-LT`***, ***`CP-UNKNOWN-GT`***, *and* ***`CP-KNOWN`***, *alongside with the premise that $\Leftrightarrow$ is trichotomous. We conclude that the computation ends and yields either 0, -1, or 1, and it is thus trichotomous.*

Lemma 3. *Given two objects – $x : C$ and $y : C^{'}$ –, $(x = y) \implies hc(x) = hc(y)$.*

Proof. *By analyzing* ***`CR-BUILD`***, ***`EQ-MULTIPLE-EQ`***, ***`EQ-SINGLE`*** *and* ***`EQ-EMPTY`***, *we deduce that $(x = y) \implies [(\forall f \in \overline{f})(eq(x.f, y.f))]$, being $\overline{f} = (es\_fields(C) \cup es\_fields(C^{'}))$. Inspecting* ***`HC-MULTIPLE`*** *and* ***`HC-SINGLE`***, *we conclude that $hc(x) \neq hc(y)$ entails that either $(\exists f \in \overline{f})((x.f = y.f) \wedge (hc(x.f) \neq hc(y.f))$ or the $\oplus$ is not associative, and thus $(\mathbb{Z}, \oplus)$ is not a semi-group, which is a contraction.*

9 Related Work

Common LISP (CLOS) [13] follows the functional paradigm. However, some of its issues also plague OO languages. Among them is the impact of the lack of orthogonality on the programmers. Baker highlights its importance and proposes the *EGAL* predicate: a uniform way of comparing values in CLOS.

Vaziri et al. [26] extend FJ to include a new construct called Relation Type (RT). An RT may declare a special kind of field – key field – that tags which information forms its equality state. A key field must not change after the initialization of an RT. Two instances of RT with the same equality state always have the same identity. That makes it possible to compare them by their memory addresses. Vaziri et al. argue that RTs could replace about 70% of comparable objects. Contrary to Vaziri et al., our proposal does not require language changes and allows objects to be mutable.

Like our proposal, Rayside et al. [21] fully embrace mutability. Unlike ours, it is not fully declarative and requires imperative code when inheritance is present. That makes it prone to the same issues of coding comparison by hand. After testing, correctness and easiness of use improved, whereas performance degraded by about 21%.

Grech et al. [10] describe a fully declarative approach to object comparison. It presents a novel approach that sensibly improves performance: statements are reordered based on how well they detect distinct objects. Sadly, we could not employ such an approach, as order evaluation is non-commutative. Grech et al. also requires objects to remain immutable after their initialization.

Modern IDEs provide wizards to create the source-code of comparison operations. Project Lombok [24] uses metadata to generate such code in compile-time. Due to their static nature, it is not possible to model type-compatible comparisons and maintain symmetry at the same time using them. The dynamic nature of our proposal mitigates that issue.

It is worth noting that, unlike ours, none of the proposals we analyzed deals with the ordering aspect of comparison.

10 Conclusion

Implementing comparison semantics on an OO language is tricky. It usually spans multiple methods that depend on each other. Failure to observe such dependencies leads to bugs that are hard to track. Since program equivalence is uncomputable, it is impossible to create a procedure that checks if these dependencies hold. Another issue is that most OO languages rely on simple message dispatch, on which symmetry (a mandatory property of equivalence relations) is not attainable. Our proposal restores orthogonality by embedding the comparison semantics on metadata instead of writing it by hand. We formalized the comparison procedure and demonstrated that it is correct. We showed that it is also possible to compute order using the same metadata.

The next step of this research is to build a prototype to check the impacts of our proposal on readily-available software. We plan to use the corpus defined by the DaCapo Benchmark Suite [2] and implement it in Java.

Cycle handling is a shortcoming of our proposal. Parnas [19] argues that cycles are a bad practice. Still, they occur on production code [16]. Preliminary analysis shows it is possible to extend our proposal to handle cycles transparently. More work is, however, necessary in its formal aspects.

Another research opportunity is to check if a solution that prevents the inclusion of mutable elements into equality collection at compile-time is viable. At first glance, it seems that an alternative collection framework combined with a type capable of identifying mutable objects, would suffice.

Bibliography

[1] Baker, H.G.: Equal rights for functional objects or, the more things change, the more they are the same. ACM SIGPLAN OOPS Messenger 4(4), 2–27 (1993)

[2] Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications. pp. 169–190. ACM Press, New York, NY, USA (Oct 2006)

[3] Bloch, J.: Effective Java. Addison-Wesley, Boston, MA, 3 edn. (2018), https://www.safaribooksonline.com/library/view/effective-java-third/9780134686097/

[4] Bruce, K., Cardelli, L., Castagna, G., Group, H.O., Leavens, G.T., Pierce, B.: On binary methods. Theory and Practice of Object Systems 1(3), 221–242 (1995)

[5] Cardelli, L., Mitchell, J.C.: Operations on records. Mathematical structures in computer science 1(1), 3–48 (1991)

[6] Flanagan, D., Matsumoto, Y.: The Ruby Programming Language: Everything You Need to Know. " O'Reilly Media, Inc." (2008)

[7] Fraser, G., Arcuri, A.: 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. Empirical Software Engineering 20(3), 611–639 (2013)

[8] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1994)

[9] Gosling, J., Joy, B., Steele, G.L., Bracha, G., Buckley, A.: The Java language specification. Pearson Education (2014)

[10] Grech, N., Rathke, J., Fischer, B.: Generating correct and efficient equality and hashing methods using jequalitygen (2010)

[11] Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Not. 39(12), 92–106 (Dec 2004), https://doi.org/10.1145/1052883.1052895

[12] Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight java: a minimal core calculus for java and gj. ACM Transactions on Programming Languages and Systems (TOPLAS) 23(3), 396–450 (2001)

[13] INCITS, A.: Incits 226-1994 (r2004). Information Technology: Programming Language: Common Lisp (2004)

[14] ISO: ISO/IEC 14882:N4778 Information technology – Programming languages – C++. International Organization for Standardization, Geneva, Switzerland (2018)

[15] Kleene, S.C., de Bruijn, N., de Groot, J., Zaanen, A.C.: Introduction to metamathematics, vol. 483. van Nostrand New York (1952)

[16] Melton, H., Tempero, E.: An empirical study of cycles among classes in java. Empirical Software Engineering 12(4), 389–415 (2007)

[17] Muschevici, R., Potanin, A., Tempero, E., Noble, J.: Multiple dispatch in practice. Acm sigplan notices 43(10), 563–582 (2008)

[18] Nelson, S., Pearce, D., Noble, J.: Understanding the impact of collection contracts on design. Objects, Models, Components, Patterns pp. 61–78 (2010)

[19] Parnas, D.L.: Designing software for ease of extension and contraction. IEEE transactions on software engineering pp. 128–138 (1979)

[20] Plotkin, G.D.: Structural operational semantics. Aarhus University, Denmark (1981)

[21] 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: Proceedings of the 31st International Conference on Software Engineering. pp. 342–352. IEEE Computer Society (2009)

[22] Rupakheti, C.R., Hou, D.: An empirical study of the design and implementation of object equality in java. In: Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds. p. 9. ACM (2008)

[23] Silva, T.M., Serey, D., Figueiredo, J., Brunet, J.: Automated design tests to check hibernate design recommendations. In: Proceedings of the XXXIII Brazilian Symposium on Software Engineering. pp. 94–103 (2019)

[24] Spilker, R., Zwitserloot, R.: Project Lombok. https://projectlombok.org/ (2019), [Online; accessed 24-April-2019]

[25] Strichman, O.: Special issue: program equivalence. Formal Methods in System Design 52(3), 227–228 (Mar 2018), https://doi.org/10.1007/s10703-018-0318-y

[26] Vaziri, M., Tip, F., Fink, S., Dolby, J.: Declarative object identity using relation types. In: ECOOP. vol. 7, pp. 54–78. Springer (2007)

[27] Zermelo, E.: Beweis, daß jede menge wohlgeordnet werden kann. Mathematische Annalen 59(4), 514–516 (1904)