Equality in Object Oriented Languages

Peter Grogono

Department of Computer Science, Concordia University
grogono@cs.concordia.ca

Philip Santas

Institute of Scientific Computation, ETH Zürich
santas@inf.ethz.ch

Abstract

Traditional programming languages provide a simplified view of equality. Many programmers have apparently inferred that equality itself is a simple concept — a matter of writing '=' in the right place and leaving the rest to the compiler. Equality tests involve semantic relationships, however, and it follows that a compiler cannot generate correct equality tests for user-defined types. In the sophisticated programming environments provided by modern object oriented languages, programmers must have a thorough understanding of equality if they are to implement it correctly and efficiently.

1 Introduction

There are concepts that appear so shallow and transparent that we do not expect to find hidden depths in them. We do not usually give much thought to equality, for example. But equality is a rich concept and, as the expressiveness of programming languages increases, it becomes increasingly important that programmers understand equality and use it correctly. The growing popularity of object oriented languages has actually led to widespread debate on what it means to say that two objects are 'equal'.

Programming languages typically provide an equality test of the form $x = y$ that compares the values of two expressions. Equality is usually defined only for built-in types, leaving the responsibility for comparing values of other types to the programmer. Features introduced in object oriented programming, especially inheritance and polymorphism, have raised the question of whether languages should provide default equality tests for all types or classes. We explain why it would not be correct to provide such defaults, and we discuss the features that languages should provide.

Programmers speak of 'overloading' symbols as though they invented the term. But overloading has a long history in mathematics, and few symbols are overloaded more than '='. The use of a single symbol suggests a universal meaning but, in fact, there is no such universal meaning. Equality in programming has the complications of equality in mathematics combined with the additional problems of efficient implementation.

2 Equality in Mathematics

Consider these symbols and strings:

$$7 \quad \text{vii} \quad 3+4 \quad \text{seven} \quad \text{sept} \quad \text{VII} \quad 111_2$$

If we agree on a universe of discourse (natural numbers, integers, or some other universe), then all of these symbols are *representations*, or *names*, for a single *value*. The representations are concrete and distinct: we can write them on paper, say them aloud, and, using yet other representations, store them in computer memory. The value, on the other hand, is an abstraction. The value corresponding to a representation is its *denotation*. To obtain the denotation of a name, we need an *interpretation*. For example, the denotation of '10' depends on whether we interpret it as a decimal integer, a string, or a hexadecimal integer.

In mathematics, we use representations to make statements about values. Consequently, it is important to define the permitted operations on representations in a way that maintains the correct relationships between values. We could ignore the difference between values and representations if there was a one-to-one correspondence between representations and values but, in general, there is not. We therefore need ways of deciding when two representations denote the same value, and this is where equality enters the picture. When we write '$3 + 7 = 10$' we are saying that the representations '$3 + 7$' and '10' denote the same abstract value. We must, of course, agree about the interpretation of the representations: the equation is true if 3, 7, and 10 represent decimal integers and + represents integer addition. If, however, + represents addition modulo 8, then $3 + 7 = 2$. If 3 and 7 represent strings, and + represents catenation, then $3 + 7 = 37$.

When mathematicians introduce a new structure, they provide a rule that determines whether two instances of the structure are equal. The rule depends on their interpretation of the structure. The following examples demonstrate some of the ways in which equality may be defined.

- Sets are equal iff their components are equal. For example, $\{1,2,3\}$ and $\{2,1,3\}$ are different representations of the same set, and so we write $\{1,2,3\} = \{2,1,3\}$.
- Ordered pairs are defined to be equal iff their corresponding components are equal. For example, $(3,4) = (3,4)$ but $(3,4) \neq (4,3)$ and $(3,4) \neq (6,8)$. We can compare tuples with three or more components in the same way.
- If we interpret the ordered pair (x,y) as the rational number x/y, then $(a,b) = (x,y)$ iff $a/b = x/y$. With this interpretation, $(3,4) = (6,8)$.
- If we define an equivalence relation on a set, we obtain a new set whose components are the equivalence classes of the original set. Rationals are conventionally defined in this way as ordered pairs of integers under the equivalence relation $(a,b) \sim (x,y)$ iff $a/b = x/y$.
- Rather than working with equivalence classes, we may choose a particular component of each equivalence class as the *canonical form* of a value. For instance, we might take $(3,4)$ as the canonical form of the equivalence class $\{(3,4),(6,8),(9,12),\dots\}$. Representations are considered equal if their canonical forms are equal.

- A graph G is an ordered pair of sets, (V, E). If we used the definitions given above for equality of sets and ordered pairs, two graphs would be equal only if their vertex sets and edge sets were equal. This rule would not be not useful in practice, because it does not often happen that the sets involved are equal. A more useful relation between graphs is *isomorphism.* G_1 is isomorphic to G_2 (written $G_1 \cong G_2$) iff there is a bijection $f : V_1 \to V_2$ which preserves edges: $(u, v) \in E_1$ iff $(f(u), f(v)) \in E_2$.

- Since a function is a set of ordered pairs, equality of functions is naturally defined using equality of sets. If the domain of a function is an infinite set, then the function is an infinite set of pairs; deciding whether two such functions are equal requires infinite time. Thus equality of functions is, in general, undecidable. Many functions, however, have finite representations from which we may be able to establish equality. For example, if

$$\begin{array}{rrcl} & f(x) & = & x^2 - 1 \\ \text{and} & g(x) & = & (x+1)(x-1) \end{array}$$

  then $f = g$ in any domain in which the identity $x^2 - 1 \equiv (x+1)(x-1)$ is true.

These examples make it clear that '=' is an overloaded symbol. The use of a single symbol with many meanings is convenient but misleading. In the examples above, we *define* equality in terms of abstract values, but we *decide* equality using concrete representations. Complications arise because there are usually many representations for each value.

Equality predicates are not arbitrary. The criteria that we use to construct them include the following.

- An equality predicate must be an equivalence relation on representations: it must be reflexive, symmetric, and transitive.

- If there are operators, the equality predicate should be consistent with them. We do not, for instance, define equality of rational numbers by the rule $(a, b) = (x, y)$ iff $a + b = x + y$, because this rule would conflict with the rules for adding and multiplying rational numbers. An equality predicate that is consistent with the operations of an algebra is called a *congruence.*

- An equality predicate that is also a congruence permits 'substitution of equals for equals': if $E(x)$ is a term, then $a = b$ implies $E(a) = E(b)$.

3 Equality in Programming

Programmers, like mathematicians, often need equality predicates for the structures that they introduce. Programming, however, introduces two additional difficulties.

- Programmers cannot ignore efficiency issues. Clever algorithms, or careful choice of representations, may be required to achieve fast equality testing.

- Programs often manipulate representations of objects in the real world. The equality predicate for these representations must correspond to a suitable interpretation of equality for real world objects.

Most programming languages provide an equality predicate for a small collection of built-in types such as `int`, `float`, `bool`, and `char`. Many languages also provide an equality test for strings, because almost everybody agrees about the meaning of string equality. Most languages, however, do not usually provide equality tests for user-defined data structures composed of arrays, records, and pointers. Three kinds of equality test are often mentioned in discussions:

- *Pointer equality* determines whether the arguments reside at the same place in memory.
- *Shallow equality* is implemented by comparing the bits of the arguments, ignoring their meaning.
- *Deep equality* compares pointer structures recursively, using shallow equality for non-pointer values.

These tests share two properties: they can easily be generated by a compiler; and they do not, in general, provide the desired semantics of equality. How, then, do we obtain correct semantics? We begin by examining a few simple cases.

Arrays If A and B are arrays with the same base type T, a reasonable equality predicate would be

$$A = B \quad \text{iff} \quad |A| = |B| = n \quad \text{and} \quad A_i \overset{T}{=} B_i \quad \text{for} \quad i = 1, \ldots, n$$

where $|A|$ stands for the number of components of A and $\overset{T}{=}$ is the equality predicate for the base type T. Such a test could be generated by the compiler as a default.

Records Records with fields that are not pointers can sometimes be compared field by field. Since some fields may not be relevant for comparison, most compilers do not generate such a test even as a default.

Binary Trees There are many ways of comparing pointer structures. For simplicity, we consider binary trees with labeled nodes. "$\overset{L}{=}$" is the equality test for labels. If X is a non-empty tree, $X.label$ is the label of its root node and $X.left$ ($X.right$) is its left (right) subtree. We define equality for binary trees as follows: $X \overset{T}{=} Y$ if either X and Y are both empty trees or

$$X.label \overset{L}{=} Y.label \quad \text{and} \quad X.left \overset{T}{=} Y.left \quad \text{and} \quad X.right \overset{T}{=} Y.right.$$

Note that we use $\overset{T}{=}$ recursively in the definition. This is an application of deep equality, as defined above.

Now suppose that we do not want to distinguish the left and right subtrees. We must change the definition as follows: $X \overset{\mathrm{U}}{=} Y$ if either X and Y are both empty trees or

$$\begin{array}{lcccc} X.label \overset{\mathrm{L}}{=} Y.label & \text{and} & (X.left \overset{\mathrm{U}}{=} Y.left & \text{and} & X.right \overset{\mathrm{U}}{=} Y.right) \\ & \text{or} & (X.left \overset{\mathrm{U}}{=} Y.right & \text{and} & X.right \overset{\mathrm{U}}{=} Y.left) \end{array}$$

This small change has a drastic effect on efficiency: the number of comparisons required increase exponentially with the size of the tree. We can avoid the increase if we can define a total ordering on the labels. Using the ordering, we transform each tree X into a normal form $n(X)$ in which the left subtree of each node has a smaller label than the right subtree. We could then use $\overset{\mathrm{T}}{=}$ to compare the transformed trees. The correctness of the method depends on the fact that

$$n(X) \overset{\mathrm{T}}{=} n(Y) \text{ implies } X \overset{\mathrm{U}}{=} Y. \tag{1}$$

Reference Comparison The time spent comparing labels in trees is wasted if the trees share memory. Suppose that the trees are represented as pointer structures. We write &X to mean 'the address of the root node of X' and $\overset{\mathrm{R}}{=}$ to denote reference comparison. Then, clearly, $\&X \overset{\mathrm{R}}{=} \&Y$ implies $X \overset{\mathrm{T}}{=} Y$. If some of the nodes of trees are shared, the following test, applied recursively, will be more efficient than the first test above.

$$\begin{array}{lll} X = Y \quad \text{iff} & & X \text{ and } Y \text{ are both empty trees} \\ & \text{orelse} & \&X \overset{\mathrm{R}}{=} \&Y \\ & \text{orelse} & X \overset{\mathrm{T}}{=} Y. \end{array}$$

In this definition, "P orelse Q" abbreviates "if P then true else Q", meaning that Q should not be evaluated if P is true.

Hashing Hashing is a transformation that reduces a complicated structure to a simple value, typically an index to a hash table. A hashing function h is *perfect* if $X \neq Y$ implies $h(X) \neq h(Y)$ for all structures X and Y. Since perfect hashing functions are hard to find, except for small, fixed sets, it usually possible to have $h(X) = h(Y)$ even though $X \neq Y$. We can be certain, however, that

$$h(X) \neq h(Y) \text{ implies } X \neq Y. \tag{2}$$

Thus the equality test based on hashing is the converse of the equality test based on references:

$$X = Y \quad \text{iff} \quad h(X) \overset{\mathrm{H}}{=} h(Y) \quad \text{andthen} \quad X \overset{\mathrm{V}}{=} Y$$

in which $\overset{\mathrm{H}}{=}$ denotes equality of the hash indices, $\overset{\mathrm{V}}{=}$ denotes equality of data values, and "P andthen Q" abbreviates "if $\neg P$ then false else Q", meaning that Q should not be evaluated if P is false.

These examples use well-known programming tricks, but their purpose is to show that efficient equality testing depends on both the structure and the interpretation of objects compared, as rules (1) and (2) show.

4 Object Oriented Programming

One of the attractive features of object oriented programming is that it narrows the semantic gap between programs and the world. Insofar as object oriented programs use the same mathematical structures as other programs — integers, strings, arrays, trees, and so on — the issues of equality are also the same. But simulation raises new issues about equality in the 'real world'.

In programming we work with representations of the objects, not the objects themselves. Moreover, object oriented programming introduces not only objects, but classes, inheritance, and polymorphism. The following examples illustrate some of the issues raised by these features.

Immutable Objects Instances of a class are *immutable* if, after they have been created, they can never be changed. In many languages, instances of built-in classes such as `int` and `string` are immutable. If instances can be changed, they are *mutable*. Most object oriented languages allow programmers to define mutable objects; some also permit the definition of immutable objects. Immutable objects are similar to conventional data structures for comparison purposes: we can apply the examples of the previous section to immutable objects.

If instances of a class are immutable, we need only create one canonical copy of each distinct object required by the program. More time will be required to create instances, but less time will be required to compare them. Sometimes, this will lead to an overall saving of time. The overhead of creation can sometimes be reduced by hashing, as described in the previous section. Since the effectiveness of unique representation depends on the application, the programmer should be able to decide whether or not it is needed.

Object Identity As we noted in Section 2, classical mathematics typically evades issues of identity by working with values rather than references. Object oriented programs often simulate the real world, and objects in a program represent physical objects rather than abstractions.

In the simplest case, equality of objects is identity: $x = y$ iff x and y denote the same region of memory. But there are other possibilities. Consider, for example, the following two objects, which belong to different classes.

Name	`Fred Fish`
SIN	`2534 675 67`
ID	`8920437`
Status	`Undergraduate`
Enrolment Date	`910623`

Name	`Fred Fish`
SIN	`2534 675 67`
Status	`Part time`
Salary	`12.50`

The left object represents an undergraduate student at a university and the right object represents a part-time employee at the same university. We assume that it is inconvenient or inefficient to merge the corresponding classes. It is likely that the two objects represent the same person, although they are instances of different — and possibly unrelated — classes. An equality predicate might compare the name and SIN fields only.

The natural way to achieve the required effect in an object oriented language would be to abstract the fields to be compared into a superclass *Person*, and define the predicate in the superclass as follows:

```
class Person
    var name, SIN: String
    function eq (other: Person): Boolean
        return $self.name = other.name$ and $self.sin = other.sin$
```

There are other cirucmstances under which different data structures may represent the same object. This situation occurs when, for example, there is a copy of the record on backing store and another copy in memory. Another possibility is that it might be necessary to keep old versions of an object around. Two instances of a data structure, with unequal components, would then represent the same object at different times in its history and, for some purposes, might be considered equal.

Inheritance We use the familiar 'colored point' example to illustrate the connection between equality and inheritance. Points are equal if their coordinates are equal.

```
class Point
    var x, y: Float
    function (=)(other: Point): Boolean
        return $self.x = other.x$ and $self.y = other.y$
```

Colored points inherit coordinates from class *Point* and add a color field. If colored points inherited *eq* from class *Point*, only their positions would be compared. We therefore redefine $(=)$ in class *ColoredPoint* so that it compares colors as well.

```
class ColoredPoint inherits Point
    var x, y: Float
    var z: Color
    function (=)(other: Point): Boolean
        return $self.x = other.x$ and $self.y = other.y$ and $self.z = other.z$
```

A full discussion of equality in the presence of inheritance is beyond our scope. Our main point is that some issues must be understood and controlled by the programmer.

Coercion Some languages allow coercion from a class to its superclass. Using coercion, we can write $c = p$ to compare a colored point c and a plain point p. The colored point c will be coerced to *Point* (by ignoring its color field) and the points will be compared by the predicate in class *Point*. Suppose that

$$p \equiv \{x := 1, y := 2\} \tag{3}$$

$$c_1 \equiv \{x := 1, y := 2, z := red\} \tag{4}$$

$$c_2 \equiv \{x := 1, y := 2, z := green\} \tag{5}$$

Then we have $c_1 = p$ and $p = c_2$ (by comparison in class *Point*), but $c_1 \neq c_2$ (by comparison in class *ColoredPoint*). This suggests that we have lost transitivity but, in fact, all that has happened is that we have confused two different equality predicates. The criteria for comparing points and colored points are different, and problems are bound to occur if we mix them.

Message Passing In object oriented languages that implement method invocation by 'message passing', the expression $c = p$ is effectively converted to $c.\mathsf{eq}(p)$, which we may paraphrase as 'ask c to execute its method eq with argument p'. Using the values (3), (4), and (5) above, $p = c_1$ will return 'true' and $c_1 = p$ will cause a type error or, even worse, attempt to access the non-existent z-component of p and yield an unpredictable result. Thus the important property of reflexivity may be lost in a language with message passing.

Encapsulation If objects are fully encapsulated, all we can do is send them messages. (This observation is, of course, merely an extension of the note above on message passing.) If x and y keep their representations hidden, they must provide functions that allow an observer to determine whether their abstract values are equal.

5 Equality in Existing Languages

When we look at what existing languages actually provide, we find that most of them provide the facilities that we need for comparing objects.

In **C++**, the operator == can be used to compare built-in values, including pointers. The same operator can be overloaded in the definition of a member function, allowing users to define their own equality tests.

CLOS, which is based on Common LISP, provides several built-in functions to test equality [5]. The expression (eq x y) is true iff x and y are the same object. However, CLOS does not guarantee that strings and numbers have unique representations. Consequently, expressions such as (eq "ho" "ho") and (let ((n 19)) (eq n n)) might be true or false. These anomalies are mitigated by the function eql, which is like eq but uses the values of number and string arguments rather than their addresses.

The expressions (equal x y) is true if x and y have the same structure and the same leaves. Most implementations of equal do not detect cycles and may fail to terminate if either of the arguments is a cyclic structure. The function equalp is similar to equal but it ignores alphabetic case and certain attributes of numbers. For example, (equal 5 5.0) and (equal "A" "a") are both false, but (equalp 5 5.0) and (equalp "A" "a") are both true. In summary:

$$(\mathsf{eq\ x\ y}) \Rightarrow (\mathsf{eql\ x\ y}) \Rightarrow (\mathsf{equal\ x\ y}) \Rightarrow (\mathsf{equalp\ x\ y}).$$

CLOS provides all that is necessary for efficient comparison, but assumes that programmers know what they are doing.

Although **CLU** is not an object oriented language, it comes close, especially with its approach to data abstraction [3]. The definition of equality for a user-defined type is the programmer's responsibility, but there are certain conventions. One of the conventions is that, if $x = y$, it must be impossible to distinguish between x and y. A consequence of this convention is that equality for mutable objects *must* be implemented as identity, because otherwise we could change one object but not another "equal" object and thereby distinguish between them. The function *similar* is used to determine whether two distinct objects have similar or equal components.

Eiffel provides a number of options for object comparison [4]. The predicate *equal* returns true if its arguments: are both void; both point to the same object; are equal values of a simple type; or are objects with identical components. The name *equal* is statically bound to a method: calls have the form $equal(x, y)$. The message $x.is\_equal(y)$ has the same meaning by default, although it fails if x is void. But *is_equal* can be redefined, and dynamic binding ensures that the implementation provided by the class of x will be used to perform the comparison.

Eiffel also provides a recursive equality test called *deep_equal*, which is carefully defined ([4, page 306]) to provide correct results for cyclic structures.

Smalltalk-80 provides several primitives from which programmers can construct appropriate equality predicates [1]. Value comparison is built-in: $x = y$ is true if x and y have the same representation. Reference comparison is also built-in: $x == y$ is true if x and y denote the same object. Smalltalk also provides support for hash comparison: the method `hash` applied to an object yields a small integer that depends on the value of the object, and the method `identityHash` applied to an object yields a small integer that depends on the address of the object. Thus Smalltalk provides default comparisons for any two objects, but also provides programmers with the tools necessary to construct their own comparisons.

6 Conclusion

In order to decide whether two objects are equal, we must know the meaning of the objects as well as their structure. Thus equality is a semantic concept. In this paper, we have suggested the following consequences of this observation.

- Identity is an *a priori* concept: an object is equal to itself.
- Equality is *not* an *a priori* concept: given two distinct objects, we need a rule that depends on the type of the objects to decide whether they are equal.
- Equality predicates are not, in general, obtained by induction over the type constructors of the language. Thus a programming language cannot provide an equality test for an arbitrary user-defined type.
- Programming languages must provide a collection of primitive operations from which equality predicates for arbitrary structures can be built.
- Special case must be taken in coding equality tests in the presence of object oriented features such as inheritance, polymorphism, and coercion.

Acknowledgements This paper was motivated partly by a discussion in the internet newsgroup `comp.object`. Some of the ideas presented here were drawn from an earlier paper [2]. Discussions with Richard Fateman, Michael Monagan, and Adam Steele were very helpful.

This paper appears in the proceedings of EastEurOOPe'93, a conference on object oriented programming held at Bratislava, Slovakia in November 1993.

References

[1] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[2] Peter Grogono. Issues in the design of an object oriented programming language. *Structured Programming*, 12(1):1–15, January 1991.

[3] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[4] Bertrand Meyer. *Eiffel: the Language*. Prentice Hall International, 1992.

[5] G.L. Steele Jr. et al. *Common LISP: the Language*. Digital Press, second edition, 1990.