Handout B: Value Equality

(Revision 2)

Christoph Reichenbach

November 15, 2021

Comparing two values for equality is an essential operation in many programming languages. However, it is not always obvious what it means for two values to be equal. Most languages agree that $0 = 0$ and $0 \neq 1$. But what about $\frac{1}{2} = \frac{2}{4}$? Here we must distinguish between the intended meaning of a value and its representation.

1 Axioms for Equality

In Mathematics, equality takes the form of an *equivalence relation*, which means that it satisfies the following three axioms:

Reflexivity:	$x = x$
Symmetry:	$x = y \iff y = x$
Transitivity:	$x = y$ and $y = z \implies x = z$

Most programming languages take great care to ensure that their notion of equality satisfies these axioms. However, there are exceptions, as we discuss below.

2 Equality in Programming Languages

In the following, we look at some of the most common forms of equality.

2.1 Primitive (Structural) Equality

When we compare two integer values, such as 0 and 1, we can directly compare the (short) bit patterns that represent these values in memory. This is efficient, precise, and unambiguous — but unfortunately this technique only works for fixed-size integers, booleans, enumerations, and characters.

2.2 Floating-Point Equality

Floating point numbers, like integers, are represented as compact bit patterns in memory. In principle we can compare them directly for equality, and many languages permit that. However, arithmetic operations on floating-point numbers are inherently imprecise. Consider the following Python example (Python 2 only):

```
>>> 1.1 + 0.1 == 1.2
False
>>> 1.1 + 0.1
1.2000000000000002
```

Here, the addition of 1.1 to 0.1 produced a result that is slightly different from one that we expect arithmetically. This is because of the trade-off that floating point numbers make: they sacrifice precision (and hence robustness) for execution speed.

As a consequence, the programming language Standard ML forbids equality comparisons between floating point numbers. Programmers must instead use less-than and greater-than comparisons, forcing them to think about suitable epsilon environments in which their conditions might hold.

2.3 Reference Equality

Let us consider an example from the programming language Java:

```
1 String hello = "Hello ";
2 hello += "World";
3 return "Hello World" == hello;
```

This code will return false, even though the comparison in line 3 compares two strings that contain exactly the same characters. The underlying reason for this behaviour is how Java (and other programming languages) represent strings. Such strings may be (almost) arbitrarily long and thus take up arbitrary amounts of memory, so Java cannot store them as efficiently as a 32-bit number and must instead place the strings in some block of memory that contains each of the characters in the string. Java then represents each string with the memory address of that structure.

When Java's equality comparison operator ==, compares strings, it only compares these *addresses*.

However, it is perfectly possible to represent the same string at different memory addresses[1]. The Java Language Specification (JLS-8, Section 15.21.3) makes this explicit:

> *While* `==` *may be used to compare references of type* `String`*, such an equality test determines whether or not the two operands refer to the same String object. The result is* `false` *if the operands are distinct String objects, even if they contain the same sequence of characters (§3.10.5).*

(While we have not discussed *objects* yet, for our purposes the term "*String object*" is the memory structure that represents the string).

We call this form of equality *reference equality*, since it does not test whether the *contents* of two variables are identical, but rather whether they are *references to the same memory address*.

2.4 Structural Equality

Some languages also support a form of equality checking that works analogously to primitive equality, but generalises its idea and scales to objects of arbitrary size. For example, in the language Go, we can define and compare records like the record person in the program below:

```
type person struct {
    name string
    age  int
}

func main() {
    a := person{"A", 20};
    b := person{"A", 20};
    fmt.Println(a == b);
}
```

This program will print out true, since Go here performs structural equality checking. If we instead write

```
a := person{"A", 21};
```

or

```
a := person{"B", 20};
```

the program will print out false, since it considers records to be equal if and only if all fields are equal.

[1] Java provides some guarantees for *literal* strings, which always have the same address: `"x" == "x"` is always true, cf. JLS-8, Section 3.10.5

2.5 Programmer-defined Equality

To compare the *contents* of two strings, Java programmers must instead use

```
return s.equals("foo");
```

This mechanism hooks into techniques that we will discuss later; its essence is that it relies on *programmer-definable notions of equality*. In Java, the default mode of comparing strings compares them for equality character-by-character. Alternative forms of equality could be to compare strings irrespective of capitalisation, such that "foo" and "FOO" are equal.

Programmer-defined equality may execute arbitrary code that might violate the three equality relations from Section 1. For instance, in Java or Python we can easily define a custom notion of equality that claims that $o \neq o$ for some object o. This is a challenge for language designers, especially since it is mathematically impossible to check whether a custom equality test in a general-purpose programming language satisfies the three axioms in general.

2.6 Ad-Hoc Equality

When we are comparing two values of different *types*, some languages like PHP and JavaScript use custom, language-specific rules for comparison. Since there is no standard terminology for this mechanism, we here call it *ad-hoc equality*.

For example, when PHP compares an integer i against an array a, it first transforms i into the array `array(`i`)`, i.e., the array with the single element i, and then uses structural equality comparison to test if `array(`i`) =` a.

Below are examples of the '==' equality operator from the language JavaScript:

```
0 == "0"    // true
0 == ""     // true
[] == 0     // true
0 == [0]    // true
[] == [0]   // false
```

Note that the last three examples show that ad-hoc equality in JavaScript does not satisfy the transitivity axiom.

3 Outlook

Most languages use a combination of these equality methods. For example, Python provides two equality checking operators:

- 'is' performs primitive equality checks on built-in types, such as integers, and reference equality checks on other types, while
- '==' is a user-definable equality checking mechanism that defaults to reference equality checking, but for most of Python's built-in types (lists, tuples, etc.) does structural equality checking.

The language usually ensures that one of these mechanisms is more "fine-grained" than the other. For example, in Python we are guaranteed that for any a and b, whenever 'a is b' is true, then we also have 'a==b', at least as long as the programmer-defined equality satisfies the equality axioms.

4 Summary

We have discussed the following forms of equality:

- **Primitive Equality**, which directly compares two values by comparing their bit patterns in memory,

- **Structural Equality**, which performs the same kind of comparison, but recursively within potentially more complex structures,
- **Reference Equality**, which checks whether two pointers/references point to the same address in memory,
- **User-Defined Equality**, which can perform arbitrary equality checking but isn't provided by the language itself.
- **Ad-Hoc Equality**, which applies language-specific heuristics to check equality.

A Changes and Errata

A.1 Revision 2

- Section 2.3 updated to correct a wrong claim about string equality checking in Java, thanks to feedback by Anton Risberg Alaküla
- Changed section numbering for consistency.